Reasonable Space for the Lambda-Calculus, Logarithmically

Beniamino Accattoli, Ugo Dal Lago, Gabriele Vanoni

Logical Methods in Computer Science
Volume 20, Issue 4, 2024, pp. 15:1–15:57
https://lmcs.episciences.org/

Submitted Feb. 01, 2023
Published Nov. 20, 2024

REASONABLE SPACE FOR THE λ-CALCULUS, LOGARITHMICALLY*

BENIAMINO ACCATTOLI [a], UGO DAL LAGO [b], AND GABRIELE VANONI [c]

[a] Inria & LIX, École Polytechnique, UMR 7161, France
e-mail address: beniamino.accattoli@inria.fr

[b] Università di Bologna, Italy & Inria, France
e-mail address: ugo.dallago@unibo.it

[c] IRIF, CNRS, Université Paris Cité, France
e-mail address: gabriele.vanoni@irif.fr

Abstract. Can the λ-calculus be considered a reasonable computational model? Can we use it for measuring the time *and* space consumption of algorithms? While the literature contains positive answers about time, much less is known about space. This paper presents a new reasonable space cost model for the λ-calculus, based on a variant over the Krivine abstract machine. For the first time, this cost model is able to accommodate logarithmic space. Moreover, we study the time behavior of our machine, which is unreasonable but it can be turned into a reasonable one using known techniques. Finally, we show how to transport our results to the call-by-value λ-calculus.

1. Introduction

Bounding the amount of resources needed by algorithms and programs is a fundamental problem in computer science. Here we are concerned with space. In many applications, say, stream processing or web crawling, *linear* bounds on computing space are not satisfactory, given the enormous amount of data processed. Therefore *logarithmic* bounds become the standard of reference. Theoretically, complexity classes such as the class L of logarithmic space, although apparently small, are very interesting, and it is not known whether they are distinct from P, see for instance Hopcroft and Ullman [HU79].

Space and the λ-Calculus. Dealing with space bounds in the λ-calculus, or in functional programming languages, has always been considered a challenge.

The first reason is related to the special role of garbage collection in functional languages and in the λ-calculus. Space usage that is linearly related to time is the worst possible usage in sequential models, since space is always bounded by time, given that—intuitively—using a unit of space requires a unit of time. In a purely functional setting without garbage collection, space is indeed linearly related to time. Therefore, to properly studying space requires explicitly taking into account garbage collection, which instead is exactly one of the

Key words and phrases: lambda-calculus, abstract machines, complexity, space.

*This is the extended version of the paper appeared with the same title at LICS2022 [ADLV22b].

LOGICAL METHODS IN COMPUTER SCIENCE
DOI:10.46298/LMCS-20(4:15)2024

aspects that functional programming aims at *hiding*, leaving it to the meta-level. The case of the λ-calculus is slightly different. The β-reduction rule can erase sub-terms, but there is not much control over this form of erasure, because it is not asynchronous as in functional languages, since—technically—erasing β-steps cannot be postponed. Consider indeed the following sequence:

$$(\lambda x.\lambda y.y)tu \quad \to_\beta \quad (\lambda y.y)u \quad \to_\beta \quad u$$

The first step is erasing but it *cannot* be postponed after the second one, because the second step is *created* by the first one.

The second reason behind the challenge in studying space is that the abstract notions of time and space in the λ-calculus have some puzzling properties, as we shall discuss at length. In particular, there are families of terms where the abstract notion of space seems to be *exponential* in the abstract notion of time, a phenomenon known as *size explosion*. This puzzling fact, which roughly means that *one does not need a unit of time to use a unit of space*, eclipses the issues related to garbage collection, and historically was the main reason why λ-calculus was not considered a good setting for computational complexity.

Logarithmic Space and the λ-Calculus. But logarithmic space is special, because it adds a further difficulty to an already challenging topic: it requires *log-sensitivity*, that is, to distinguish between *input space* and *work space*—without such a distinction one cannot even measure sub-linear space. Since the λ-calculus does not distinguish between *programs* and *data*, it is *log-insensitive* and, apparently, at odds with logarithmic space.

The literature about the λ-calculus does nonetheless offer results about space complexity, but they are all *partial*, as they either concern logarithmic space for *variants* of the λ-calculus, as for Dal Lago and Schöpp [Sch07, DLS16], Mazza [Maz15] and Ghica [Ghi07], or they do deal with the λ-calculus but apply only to linear space and above, as it is the case of Forster et al. [FKR20].

Contribution. The main result of this paper is the first fully fledged space reasonability result for the pure, untyped λ-calculus, accounting for logarithmic space. Precisely, we retrieve log-sensitivity by representing the *input space* as λ-terms, and the *work space* as the space used by a new variant of the well-known Krivine abstract machine (KAM) [Kri07] that we dub *Space KAM*. We then prove that such a notion of work space is *reasonable* and it accounts for logarithmic space. Reasonable, roughly, means equivalent to the notion of space of Turing machines (shortened to TMs). More accurately, we show that there are encodings of TMs into the λ-calculus and vice-versa inducing simulations with a linear space overhead[1]. For space, the tricky simulation is the one of TMs into the λ-calculus (for time, it is the opposite one), which is studied in great detail in this paper. The other simulation is only outlined, as it does not present any difficulty. Section 2 contains an original perspective on the theory of reasonable cost models for the λ-calculus and of how our result fits in.

[1]The notion of *linear* space overhead with respect to sub-linear space might be confusing: if a TM uses work space $\mathcal{O}(\log n)$, where n is the size of the input, then the simulation in the Space KAM must use $k \cdot \log(n) \in \mathcal{O}(\log n)$ work space, and not $\mathcal{O}(n)$ space. In other words, what is linear is the overhead, not the function describing the space consumption.

Key Ingredients. Our result follows from a careful dissection and refinement of the KAM and of the simulation of TMs into the λ-calculus. A peculiar aspect is that the result does not rest on a *single* innovation or idea. It rests instead on the simultaneous addressing of *numerous* critical points of the encoding of TMs and of the KAM. None of them is in itself difficult or striking—apart perhaps from the disabling of environment sharing discussed below—but all of them are mandatory for the result to hold. We identify six critical points, of which we provide an overview in Section 3. Four of them are of a high-level nature:

(1) *Eager garbage collection*: environment-based abstract machines such as the KAM are usually presented without garbage collection, assuming that the meta-level shall take care of it, as it is customary in functional programming. For a parsimonious use of space, it is instead essential to re-use space as much as possible, thus having a first-class treatment of garbage collection. From the point of view of cost models, this change disentangles space from time (more precisely, from time considered as the number of β-steps). Concretely, we introduce a new variant of the KAM with eager garbage collection (plus another optimization) dubbed Collecting KAM.
(2) *Disabling data structure sharing*: there are two different forms of sharing at work in the KAM that are usually not distinguished, namely the *sharing of sub-terms* provided by environments, and the *sharing of environments* themselves. These aspects, actually, are not explicit in the specification of the KAM, they are left to the concrete implementation of the KAM. The Space KAM—which is a specific concrete implementation of the Collecting KAM— adopts the former but forbids the latter. Abstractly, this is needed to turn the data structures of the KAM into sort of *tapes* of TMs. Tapes are special in that they are *flat*, that is, cells are juxtaposed without using space, rather than linked via pointers, which would add a space overhead. Similarly, then, the data structures of the Space KAM are flat, which has the consequence of forbidding the sharing of environments. This is probably the most surprising point of our work, and—to our knowledge—the first time that such an approach is adopted in the literature on abstract machines for the λ-calculus.
(3) *Encoding and moving over tapes*: designing the Space KAM is only half of the story. The other half is the refinement of the encoding of TMs into the λ-calculus. Our reference is the encoding by Dal Lago and Accattoli [DLA17], which uses a linear amount of extra space to simulate the moving over TMs tapes. This is particularly bad for the input tape, for which a logarithmic overhead is required. Therefore, we change the encoding of input tapes, exploiting their read-only nature, to achieve the required overhead.
(4) *Low-level complexity analysis and left addresses*: the complexity analysis of the space used by the Space KAM to execute the encoding of TMs is not *abstract*, that is, it is not simply given by the maximum number of pointers to the code times the logarithm of the code. It is *low-level* in the sense that one has to inspect the size of pointers, and the reasonable bound holds only because some of them turn out to have constant size. This is obtained via a specific addressing scheme for pointers to the code, what we call *left addresses*.

Looking Back. Understanding the whole picture, at both the high-level of the theory of cost models and the low-level of machines simulations, is far from obvious. The following sections shall strive to provide such a picture. The long-standing riddle of logarithmic reasonable space for the λ-calculus, however, turns out to have a relatively simple solution. Roughly, it is enough to add two simple space optimizations (eager garbage collection plus

environment unchaining) to the KAM, obtaining the Collecting KAM, and to further disable its sharing of environments, obtaining the Space KAM, if the new encoding of TMs is taken for granted. Our Space KAM, indeed, is not much more complex than the KAM itself.

The difficulty behind the quest for a logarithmic reasonable work space is better understood by considering that it required dismissing two widespread intuitions about the problem, as we shall now explain. Additionally, the literature about abstract machines of the λ-calculus is mainly concerned with time, for which the issues for space are irrelevant, and are therefore often treated in an ambiguous and inaccurate way.

Somewhat unusually, the main obstacle behind the solution of the long-standing problem turned out to be reasoning without the preconceptions and the inaccuracies of the established knowledge about space for the λ-calculus and the theory of abstract machines.

A Wrong Positive Belief: the Geometry of Interaction. For 15 years, logarithmic reasonable space was believed to be connected to the alternative execution schema offered by Girard's *geometry of interaction* [Gir89]. Mackie's and Danos & Regnier's *interaction abstract machine* (shortened to IAM) [Mac95, DR95], recently reformulated by Accattoli et al. in [ADLV20], is a machine rooted in the geometry of interaction and in Abramsky et al.'s game semantics [AJM00]. It is based on a log-sensitive approach, and—apparently—it is parsimonious with respect to space. Schöpp [Sch06, Sch07] (with later developments with Dal Lago [DLS10]) was the first one to show how IAM-like mechanisms can be used for dealing with logarithmic space. It was since then conjectured that the space of the IAM were a reasonable cost model. The belief in the conjecture was reinforced by further uses of IAM-like mechanisms for space parsimony related to circuits, by Ghica [Ghi07], and for characterizing L, by Mazza [Maz15]. In 2021, however, Accattoli et al. essentially *refuted* the conjecture: the space used by the IAM to evaluate the reference encoding of TMs is unreasonable [ADLV21b] (as well as time inefficient [ADLV21a]). While one might look for different encodings, the unreasonable behavior of the IAM concerns the modeling of recursion via fix-point combinators (or self application) which is a cornerstone of the λ-calculus, hardly avoidable by any alternative encoding.

A Wrong Negative Belief: The Space Cost of Environments. Another misleading belief was that environment-based abstract machines could not provide reasonable notions of space. Environments are data structures used to achieve time reasonability. According to Fernandez and Siafakas [FS08], there are two main styles of environments, *local* and *global*, studied in-depth by Accattoli and Barras [AB17]. Global environments (as in the Milner Abstract Machine [ABM14]) are log-insensitive because they work over the input space. Local environments (as in the KAM) are log-sensitive. There are two reasons why their usual presentation is space unreasonable.

Firstly, garbage collection is not usually accounted for, which leads to ever-increasing space usage, while reasonable space should be re-usable. This issue is easily solved by adding garbage collection. One actually needs an eager form, in order to maximize space re-usability, and to implement it in a naive, time-ineffiecient way, as time efficient techniques such as reference counters would add an unreasonable space overhead.

Secondly, and more subtly, environments are usually space unreasonable because of the use of pointers for sharing. To be precise, local environments use *two* types of pointers, handling two forms of sharing: sub-term pointers, which serve to avoid copying sub-terms, and environment pointers, which both realize their linked list structure and the sharing

of environments. Sub-terms pointers are a key aspect of logarithmic space computations, and are thus crucial. Environment pointers are instead what makes environments space unreasonable (despite being, according to Douence and Fradet [DF07], the *essence* of the KAM): they introduce a logarithmic *pointer overhead* that, at best, gives simulations of TMs with a $\mathcal{O}(n \log n)$ overhead in space, instead of the required $\mathcal{O}(n)$ for reasonability. It was then generally concluded that environments cannot provide reasonable space.

We here show that, instead, environments make perfect sense also without environment pointers, and so without sharing of environments, by implementing them simply as strings of adjacent symbols and copying their whole content instead of copying only pointers to them—these shall be referred to as *flat environments*. Such an unusual approach is one of the critical ingredient for space reasonability. At the same time, adopting flat environments has two correlated consequences. Firstly, it breaks *time reasonability* (with respect to time considered as the number of β-steps), as we show in Section 10.1. In Section 11, we discuss how to recover simultaneous reasonability for both time and space.

Secondly, for some terms it leads to extreme inefficiencies for both time and space. Namely, this happens for terms that are *not* encodings of TMs and for which environment sharing provides a speed-up, that can even be exponential for both time and space. While at first sight this fact might seem startling, it is in fact well known that *reasonable* does not mean *efficient*: it only means that the theory can *simulate and be simulated by TMs with negligible overhead*. See Accattoli [Acc17] for discussions of this delicate point with respect to time.

Pointers, Abstract Machines, and Abstract Implementations. The literature on abstract machines for the λ-calculus usually assumes that pointers are used in implementations. Still, pointers are not usually explicitly accounted for in abstract machine specifications. Such an ambiguity can be both positive and negative. On the positive side, it allows one to omit details that might be irrelevant for the intended result, obtaining simpler machines. On the negative side, it makes such specifications ambiguous and prevents precise cost analyses. Different implementations of the machine can indeed be possible, sometimes with very different asymptotic complexities, and forms of sharing can be unhygienically hidden behind the meta-level assumption that some of the machine components are represented via pointers. This is for instance the case of the KAM, which is time reasonable (for time = # of β steps) *only if* environments are shared—as we here show for the first time, providing an example of unreasonable time overhead in absence of environment sharing (Prop. 10.8)—even if such sharing is not explicit in the specification of the KAM.

Taking into account pointers and their size is mandatory for the study of space, and even more so for logarithmic space. Therefore, we refine abstract machines by adding specifications of the time and space cost for each component, what we dub *abstract implementations*. It is a methodological contribution of this work to the theory of abstract machines. It is required for the space reasonability result, but we believe that its value is independent of it.

Encoding of TMs and Call-by-Value. Beyond the design of the Space KAM, our other main contribution is a new encoding of TMs in the λ-calculus. A critical point, as already mentioned, is that we change the representation of the input tape, in order to achieve the required logarithmic overhead. A further point is that the new encoding is carefully designed so as to retain the *indifference property* of the reference one, i.e. the fact that it behaves the

same under both call-by-name and call-by-value evaluation. We then build over this design choice by showing that our results smoothly transfer to call-by-value evaluation.

This is in contrast to what happens for time. The study of reasonable time for the λ-calculus is also based on a strategy-indifferent encoding of TMs, but in that case the difficult direction is the other one, that is, the simulation of the λ-calculus on TMs. To obtain such reasonable simulations, different strategies require different treatments. It turns out, then, that reasonable space can be studied more uniformly than reasonable time.

Sub-Term Property. The techniques for reasonable time and reasonable space seem to be at odds, as they make essential but opposite uses of linked data structures. Both techniques, however, crucially rely on the *sub-term property* of abstract machines, that is, the fact that duplicated terms are sub-terms of the initial one. For time, it allows one to bound the cost of duplications, while for space it allows one to see sub-terms as (logarithmic) pointers to the input. The sub-term property seems to be the *unavoidable ingredient for reasonability in the λ-calculus*. For extensive discussions about the sub-term property, see Accattoli [Acc23], particularly Section 3 therein.

Related Work: Safe for (Reasonable Logarithmic) Space. Disabling sharing environments also plays a crucial role in a work about space by Paraskevopoulou and Appel [PA19]. They study *closure conversion*, a program transformation at work in compilers for functional languages, turning abstractions into *closures*, that is, into pairs of transformed abstractions and *environments*. In the compilers literature, *closures* and *environment* refer to concepts that are similar and yet different with respect to those used in the abstract machine literature. Despite the differences (not discussed here), one can see some analogies between [PA19] and our work. The problem studied in [PA19] is which data structure for the (compiler) closures/environments of transformed programs is *safe for space*, that is, preserves the space used by the source program. They show that *flat environments* are safe for space, where *flat* means without environment sharing. Playing with their slogan, one might then say that *flat environments are safe for reasonable logarithmic space*.

In [PA19], it is stressed that linked environments are *not* safe for space because different environments might share sub-environments, preventing some garbage to be collected. Our study stresses a different danger, namely the pointer overhead introduced by the linked representation, which is unreasonable for logarithmic space. By removing pointers, we also remove the sharing of sub-environments. In [PA19], flat environments are *records*, that are assumed to not be linked via pointers, so the two approaches agree. Simply, the speech in [PA19] does not mention the danger of the pointer overhead, which is instead crucial for us. To avoid misunderstandings, we stress that both works study flat environments but the studied problems and the used techniques are incompatible: closure converted terms (with flat environments) can have size quadratically bigger than source λ-terms, so that closure conversion cannot be used to study reasonable space.

Related Work. The space inefficiency of environment machines is also observed by Krishnaswami et al. [KBH12], who propose techniques to alleviate it in the context of functional-reactive programming and based on linear types. A characterization of PSPACE in the λ-calculus is given by Gaboardi et al. [GMR12], but it relies on alternating time rather than on a notion of space. The already-mentioned works by Dal Lago and Schöpp [Sch07, DLS16] and Mazza [Maz15] characterize L in variants of the λ-calculus, while Jones characterizes L using a programming language but not based on the λ-calculus [Jon99]. Blelloch and

coauthors study in various papers how to profile (that is, measure) space consumption of functional programs [BG95, BG96, SBHG10], also done by Sansom and Peyton Jones [SJ95]. They do not study, however, the reasonability of the cost models, that is, the equivalence with the space of TMs, which is the difficult part of our work. Finally, there is an extensive literature on garbage collection, as witnessed by the dedicated handbook [JHM11]. We here need a basic eager form, that need not be time efficient, as the Space KAM is time unreasonable anyway.

2. The Theory of Reasonable Cost Models for the λ-Calculus, and How Our Result Fits in It

Reasonable Cost Models. According to the seminal work by Slot and van Emde Boas [SvEB88, vEB90], the adequacy of space and time cost models is judged in relationship to whether they reflect the corresponding cost models of TMs, the computational theory[2] from which computational complexity stems. Namely, a cost model for a computational theory T is *reasonable* if there are mutual simulations of T and TMs (or another reasonable theory) working within:

- for *time*, a polynomial overhead;
- for *space*, a linear overhead.

In many cases, the two bounds hold simultaneously for the same simulation, but this is not a strict requirement. The aim is to ensure that the basic hierarchy of complexity classes

$$\mathsf{L} \subseteq \mathsf{P} \subseteq \mathsf{PSPACE} \subseteq \mathsf{EXP} \tag{2.1}$$

can be equivalently defined on any reasonable theory, that is, that such classes are *robust*, or theory-independent. Note a slight asymmetry: while for time the complexity of the required overhead (polynomial) coincides with the smallest robust time class (P), for space the smallest robust class is logarithmic (L) and not linear space. In particular, a simulation within a linear space overhead for logarithmic space implies that one needs to preserve logarithmic space.

A typical example of reasonable theory is *random access machines* (RAMs), which are simulated by Turing machines within a *quadratic* time overhead—which justifies the requirement for a polynomial (rather than linear) time overhead—needed for simulating random access on sequential access tapes. The very concept of reasonable cost model was introduced to study the relationship between the space consumption of RAMs and of Turing machines [SvEB88].

Locked Time and Space. On TMs, space cannot be greater than time, because using space requires time—we shall then say that space and time are *locked*. If both the time and space cost models of a computational theory are reasonable, are they also necessarily locked? This seems natural, but it is not what happens in the λ-calculus, at least with respect to its abstract cost models.

[2]We prefer *computational theory* to *computational model* in order to avoid the overloading of the word *model* already in use for *cost model*.

(Unreasonable) Abstract Machines. Before diving into the subtleties of cost models for the λ-calculus, we clarify concepts and terminology that might be confusing for the unacquainted reader. The λ-calculus is an abstract setting relying on a single rule, β, which is non-deterministic (but confluent), and that is a non-atomic operation involving three meta-level aspects:

(1) The search of the β-redex, and
(2) Capture-avoiding substitution, itself based on
(3) On-the-fly α-renaming.

In order to study cost models for the λ-calculus, one usually fixes a deterministic evaluation strategy (typically call-by-name or call-by-value) and some micro-step formalism, typically an abstract machine, which simulates β and explicitly accounts for the three meta-level aspects. Therefore, abstract machines are intermediate settings used to study simulations of the λ-calculus into TMs (or other reasonable theories).

We shall often say that a certain abstract machine is unreasonable for time or space. The use of *unreasonable* in these cases is different than when referred to a computational theory (such as the λ-calculus or TMs). It means that the simulation realized by the machine, *and only that simulation*, works in bounds that exceed those for reasonable time or space. In contrast, a theory is unreasonable if *all* possible simulations exceed the bounds.

Why Studying Reasonability for the λ-Calculus? As another clarifying preliminary, let us answer such a question. There are *two* reasons. The *theoretical* and *external* motivation is being able to define the hierarchy (2.1) in the λ-calculus, hoping to help resolving the decades-long separation between the λ-calculus and mainstream computer science. The *concrete* and *internal* motivation is instead to better understand how to evaluate λ-terms, for which there are many different approaches and no general theory.

Cost Models for the λ-Calculus. For the λ-calculus there are a few natural candidates as cost models. Fix an evaluation strategy $\to_{\mathsf{str}}$. Then, we have three candidates for the time cost of a $\to_{\mathsf{str}}$ evaluation $\rho : t_0 \to_{\mathsf{str}} t_1 \to_{\mathsf{str}} t_2 \to_{\mathsf{str}} \dots \to_{\mathsf{str}} t_n$ of length n:

Ink time:	the time taken by printing out all the terms t_i for $i \in \{0, \dots, n\}$;
Abstract time:	the number n of $\to_{\mathsf{str}}$ steps;
Low-level time:	the time taken by an abstract machine implementing ρ.

For space, the ink and abstract notions coincide:

Ink/abstract space:	the size of the largest λ-term among the t_i;
Low-level space:	the maximum space used by an abstract machine implementing ρ.

Let us first discuss time. Ink time is locked with ink space and easily shown to be reasonable. The problem with it is twofold: on the one hand, it is too generous a notion, since the cost of functional programs is not usually estimated in this way (in functional programming practice), on the other hand, it is difficult to reason with such a notion, as it is not abstract enough.

Low-level time is a better measure, which is locked with low-level space and easily proved to be reasonable. It differs from ink time in that abstract machines usually rely on some form of sharing to avoid managing the ink representation of all the terms of the evaluation sequence. The obvious drawback is that low-level time depends on the details of the implementation and on which optimizations are enabled. It is thus not abstract, nor fixed once for all, not even when the evaluation strategy is fixed, and not even when the abstract machine is fixed, because the choice of data structures for a concrete implementation

usually affects the complexity. It is rather a family of cost models. In particular, it does not have the *distance-from-implementative-details* that is distinctive of the λ-calculus. Similar arguments apply to low-level space.

Abstract time is the best notion, since it does not depend on an implementation and it is close to the practice of cost estimates, which does count the number of *function calls*, that, roughly, is the number of β-steps. The puzzling point is that it is *not locked* with ink space: ink space can be *exponential* in abstract time (independently of the strategy), a degeneracy known as *size explosion*—we shall say that time and space are *explosive*.

Is the λ-calculus reasonable? It certainly is, with respect to unsatisfying cost models. The question rather is whether abstract time is a reasonable cost model. This was unclear for a while, because of the intuition that reasonable cost models have to be locked.

The next paragraphs shall discuss the reasonability of abstract time and then explain the delicate aspects of reasonable space, but let us anticipate a point that shall seem to contradict what we just explained. About logarithmic space, there is an unresolvable tension. On the one hand, the principle of *distance-from-implementative-details* would require abstract space—that is, ink space—to be reasonable in the logarithmic case. On the other hand, ink space is log-insensitive and thus cannot be sub-linear. The tension shall be resolved in this paper by abandoning the abstract principle and turning to a low-level notion of logarithmic space. While not ideal, this is the only available solution, at present. A more abstract cost model for logarithmic space would be preferable, in principle, but it cannot be ink space—because of the mentioned tension—and it is far from clear that an alternative abstract notion of space is possible, given the many difficulties discussed in this paper. In fact, before our work, even the existence of a low-level space cost model accounting for logarithmic space was a longstanding open problem. Implementations of the λ-calculus, indeed, are tuned for time-efficiency and inevitably use space in an unreasonable way. Therefore, simply adopting low-level space is *not enough*.

Abstract Time is Reasonable. In the study of abstract time, what is delicate is the simulation of the λ-calculus into a reasonable theory, which typically is the one of random access machines rather than TMs. The difficulty stems from the explosiveness of abstract time, and requires a slight paradigm shift. To circumvent the exponential explosion in space, λ-terms are usually evaluated *up to sharing*, that is, in abstract machines with sharing that compute shared representations of the results. These representations can be exponentially smaller than the results themselves: explosiveness is then encapsulated in the sharing unfolding process (which itself has to satisfy some reasonable properties, see [ADL16, CAC19]). The number of β steps (according to various evaluation strategies) then turns out to be a reasonable time cost model (up to sharing), despite explosiveness. Resorting to sharing amounts to studying abstract time assuming that the underlying notion of space is *low-level space* rather than ink space (which forbids sharing). It is important to point out that the adopted notion of low-level space is *not* proved to be space reasonable.

The first such result is for weak evaluation by Blelloch and Greiner [BG95], then extended to strong call-by-name evaluation by Accattoli and Dal Lago [ADL16], and very recently transferred to strong call-by-value by Accattoli et al. [ACC21] and to a variant of strong call-by-value and to strong call-by-need by Biernacka et al. [BCD21, BCD22].

Ink Space. For space, the difficult direction is, instead, the simulation of TMs in the λ-calculus. TMs are space-minimalist, as their only data structure, the tape, is a flat data

structure that *juxtaposes* cells rather then linking them via pointers—this is one of the key points. Motivated by time-efficiency, all abstract machines for the λ-calculus rely instead on linked data structures, and—as already pointed out in the introduction—the linking pointers add a logarithmic factor to the overhead for the simulation of TMs that is space unreasonable. Therefore, reasonable space requires to evaluate without using linked data structures when they are not needed, as it is the case for the encoding of TMs.

It is a recent insight by Forster et al. [FKR20] that evaluating without *any* data structure (via plain rewriting, without sharing) is reasonable for linear ink space even if unreasonable for abstract time (because of explosiveness). An interesting aspect of this result is that it establishes that the rigid (i.e. non-postponable) management of garbage collection provided by the λ-calculus is enough for reasonable linear space.

Pairing up Abstract Time and Ink Space. Forster et al. [FKR20] also show a surprising fact. Given two simulations, one that is reasonable for ink space but not abstract time, and one that is reasonable for abstract time but not ink space, there is a smart way of interleaving them as to obtain reasonability for abstract time and ink space *simultaneously*. Their result therefore shows that, surprisingly, a computational theory can be reasonable for *unlocked* and explosive notions of time *and* space. Whenever their interleaving technique applies, however, it also induces a second (low-level) reasonable cost model for space that is locked with the time cost model.

Forster et al.'s is a remarkable contribution to the external interest in reasonable cost model (that is, for defining complexity classes) but their interleaving machine is not a machine that one would use for concrete implementations. That is, the result does not help in the internal (i.e. implementation-oriented) understanding of reasonable space.

Work vs Ink Space. A puzzling fact is that sub-linear space *cannot* be measured using the ink space cost model, and is then *not* covered by Forster et al.'s result. The reason is that if space is the maximum size of terms in an evaluation sequence, the first of which contains the input, then space simply *cannot be* sub-linear. How could one accommodate for *logarithmic* reasonable space? As already explained in the introduction, one needs *log-sensitivity*, that is, a distinction between an immutable *input space*, which is not counted for space complexity, and a (smaller) mutable *work space*, that is counted. It is then natural to switch, again, from ink space to a form of low-level space. Namely, one considers the work space as the low-level space used by an input-preserving abstract machine.

Low-Level Space. Turning to low-level space, however, does not immediately provide a solution. As we already mentioned, most abstract machines rely on sharing mechanisms which allow one to prove that abstract time is reasonable while, unfortunately, also make those simulations space unreasonable, as they rely on pointers which add an unreasonable space overhead. It was thus unclear whether low-level space could be reasonable *at all*. The machines for time reasonability realize sharing via environments. The community believed that abstract machines that rely on so-called *tokens* (related to the geometry of interaction), rather than on environments, might provide reasonable notions of low-level space, but it was showed by Accattoli et al. that this is not the case [ADLV21b].

The intuition that one could use environments and yet disable their sharing, as in the Space KAM, is a contribution of the present work. Our main result is that the low-level space of the Space KAM—from now on referred to as *work space*—is indeed reasonable.

Space KAM and Time. We also study the time behavior of the Space KAM. Adopting flat environments implies giving up environment sharing, which—we show with an example—makes the Space KAM unreasonable for abstract time. The situation is then a familiar one: abstract time and work space are explosive. Work space is in this respect a conservative refinement of ink space. On the other hand, we prove that the *low-level time* of the Space KAM is an alternative *reasonable* cost model, obviously locked with work space.

Is Work Space a Good Cost Model? It might be argued that work space, being a low-level notion of space, is unsatisfactory. While this is partly true, we believe that it would be an unfair assessment. A first argument against this criticism is that it is unclear what would be the alternative, given that ink space is ruled out by its log-insensitivity. A second argument is that the requirements for reasonable logarithmic space are so strict that they almost *dictate* how the Space KAM has to be. There does not seem to be much room for alternative designs. That is, it is a cost model given by an abstract machine, but it is a quite special machine: the criticism to low-level cost models, amounting to the fact that different machines would provide different notions of cost, does not seem to apply here. Moreover, we also prove that the same cost model works also for call-by-value.

A Notion of Abstract Space. The space of the Space KAM is obtained by taking the maximum number of closures during its execution, and weighing every closure with the size of its sub-term pointer (which is not necessarily logarithmic in the size of the input). Taking only the number of closures—ignoring the size of pointers—provides a more abstract notion of *closure space* that is not the actual cost model (if adopted as space cost model, the simulation of the λ-calculus in TMs has an additional and unreasonable logarithmic overhead) and yet it provides a useful abstraction. For instance, this closure space is stable by η-equivalence, as we show. Additionally, in a companion paper we show how to measure closure space via multi types [ADLV22a], thus abstracting away the low-level details of the machine.

What Does Our Result Say for Concrete Implementations? The Space KAM is a realistic machine as long as logarithmic space is the main concern. The example showing that abstract time and work space are explosive also confirms that—as for time—*space reasonable* and *space efficient* are in some sense different concepts: the example uses exponentially less space (and it is thus more efficient) if environment pointers (which are space unreasonable) are enabled. The key point is that the λ-terms of the example are not in the image of the encoding of TMs: for them, environments sharing provides an exponential speed-up (for both time and space), while on the image of the encoding it only provides a slow-down. In other words, the Space KAM is efficient for first-order, that is, TMs-like logarithmic space computations, but beyond them it can be *desperately* space inefficient.

3. Bird's Eye View of the Problems and Their Solutions

There shall be six critical points which—only when are all solved simultaneously—shall allow us to achieve the main result of this paper, namely the fact that the space of the Space KAM is a reasonable space cost model for the λ-calculus accounting for logarithmic space. None of them is particularly difficult to deal with, but each one of them is critical, and solving all of them at once does make the solution somewhat involved. In particular, a reader can easily get lost and lose sight of where the essence of the result is. Here we give an

overview of these critical points and of how we address them. One of our main contributions is the careful identification of these points.

Let's start by listing some key points of the simulation of TMs by an abstract machine.

- *Pointers*: TMs do not use pointers, while in the λ-calculus and its execution via abstract machines, three kinds of pointers play a role. Some are essential and some are dangerous for logarithmic reasonable space.
  - *Variable pointers*: variables are pointers to their binders. Such pointers are unavoidable. Different term representations induce pointers with different properties.
  - *Sub-term pointers*: abstract machines often manipulate pointers to sub-terms rather than the sub-terms themselves. This is essential for logarithmic space.
  - *Data pointers*: data structures are often implemented as pointer-based structures, such as pointer-based linked lists. These pointers are also often responsible for forms of data-structures sharing, which is essential in time-reasonable computations. They are dangerous for reasonable space.
- *Tapes*: the input tape of the TM is meant to be represented as part of the initial code fed to the abstract machine, the rest of which is dedicated to represent the transition function of the TM. The work tape is instead represented by the data structures of the abstract machine, namely the applicative stack and the environments.
- *Overhead*: there are three points in which the encoding and the simulation incur some overhead.
  (1) *Input representation*: this is due to the representation of input strings s as λ-terms. One expects such an overhead to be $\mathcal{O}(|s|)$, that is, linear in the length $|s|$ of the input string. It depends on the fixed encoding of strings *and* on the adopted notion of variable pointer, that is, on the fixed representation of λ-terms.
  Technically speaking, the linear overhead here is not mandatory: a reasonable simulation is possible also if the input representation overhead is not linear (it can be polynomial, usually $\mathcal{O}(|s| \log |s|)$, but not exponential), as long as sub-term pointers and the overhead for scrolling the input tape (discussed below) are both logarithmic in the size of s (rather than in the size of the encoding of s as a λ-term). We shall however show how to obtain a linear input representation overhead, as it seems natural and it reinforces the trust in the correctness of the result.
  (2) *(Work) tape representation*: tapes juxtapose cells using zero space for such a juxtaposition. For the input tape, in fact it is not mandatory that the representation overhead is linear, as discussed in the previous point. The representation of the work tape, instead, has to be linear. Since it is given by the data structures of the abstract machine, it usually rests on data pointers, which add an unreasonable logarithmic overhead. It is important to point out that here the additional logarithmic factor is not in the size of the input: data pointers are generated along the execution and thus they depend on the amount of space currently in use. Thus data pointers add a bureaucratic space factor that is logarithmic on the space used by TMs.
  (3) *Tape scrolling*: this is the space overhead needed to simulate in the λ-calculus the moving of the TM head over the tape, with respect to the size of the tape. It is a notion that is tricky to define precisely because, in the reference encoding of TMs, a single move over the tape incurs only a constant space overhead. It is better intended as the space overhead generated by scrolling the *whole* tape from, say, left to right. Most TM executions never do such a mono-directional scrolling, but they nonetheless incur such an overhead during their continuous moving over the tape.

For our result, we shall need a space overhead for scrolling the input tape that is logarithmic in its size, while for the work tape a linear space overhead (in its size) is enough. Clearly, the requirement for the input tape is specific to the study of logarithmic space. We shall see, however, that (independently of logarithmic space) the adoption of a low-level space cost model (instead of ink space) forces us to be very careful with respect to the linear space scrolling overhead for the work tape, since the use of machine pointers can easily add an unreasonable logarithmic factor.
Here various ingredients play a role: the way in which tapes are encoded as λ-terms, the space optimizations of the abstract machine as well as the way in which sub-term pointers are organized.

We now list the ingredients that allow us to meet the requirements for the three overheads.

(1) *Input representation*. Here the optional linear space overhead is achieved by:
   (a) Using the Scott encoding of strings, which is also used in the standard enconding of TMs;
   (b) Adopting a representation of λ-terms for which the Scott encoding uses variable pointers of size proportional to the size of the *alphabet* Σ of the input string s, rather than to the size of s itself (which would occur if one would represent variables via textual names and enforce Barendregt's convention, or if one would represent λ-terms as proof nets or string diagrams). The actual representation is left unspecified, but for instance de Bruijn indices would do. This is discussed at the beginning of Section 9.

   If the input representation has a more than linear overhead (typically $\mathcal{O}(|s| \log |s|)$ for input string s), the logarithmic size of sub-terms pointers is obtained by adopting a specific address scheme for the Space KAM, what we call *left addresses*. This is discussed in Section 4.
(2) *Work tape representation*: in this case the linear space overhead is achieved by totally disabling the use of data pointers in the abstract machine. The stack and the environments are then represented via contiguous cells, as strings, without linking the different cells via data pointers. This is discussed in Section 7 and it is unusual for abstract machines. It has the consequence of removing the sharing of data structures needed for time reasonability. As already mentioned, our Space KAM shall be time unreasonable, with respect to the abstract cost model for time.
(3) *Tape scrolling*. For scrolling tapes with the required overhead, both the encoding and the abstract machine have to be modified.
   - *Encoding*. The standard encoding of (single tape) TMs is based on a representation of tapes, which comes with costant-time read operations but linear space scrolling overhead. This would simply forbid our result, as the input tape scrolling overhead has to be logarithmic. Therefore, we modify the encoding and adopt a different representation for the input tape—dubbed *mathematical representation* following van Emde Boas [vEB12]—coming with polynomial time read operations (thus worse for time) and logarithmic space scrolling overhead (but better for space). For the work tape, we keep the standard representation because the work tape requires only a linear space scrolling overhead, and for write operations the standard representation is better suited. This aspect is discussed in Section 9.
   - *Abstract machine*. There are two critical points here.

(a) *Sub-term pointer addressing*. We mentioned that the standard representation of tapes comes with linear space tape overhead. This is indeed true, but it requires a low-level analysis of the size of sub-term pointers, showing that some of them are of constant rather than logarithmic size, and it holds only if sub-term pointers satisfy some properties. Such properties are satisfied by the *left addressing scheme* that we adopt for the Space KAM. This aspect is discussed in Section 9.
(b) *Space awareness*. Environment-based abstract machines are usually developed having time efficiency in mind and neglecting space. For achieving the required tape scrolling overheads, they have to be tuned so as to be parsimonious with respect to space. There are two optimizations that have to be realized, and that are formally defined in Section 8:
    (i) *Eager garbage collection*. In absence of garbage collection, abstract machines have an inflationary, ever increasing use of space. In particular, they allocate space at every β-step, which is the *tick* of their abstract notion of time. Therefore, their use of space is entangled with their use of time, which is unreasonable for space. Garbage collection is needed in order to disentangle space from time. Eager collection is needed to maximize such disentanglement. In addition, garbage collection has to be implemented in a naive and time-inefficient way, because smart techniques such as reference counters would incur a space overhead—for counters—that would be unreasonable.
    (ii) *Unchaining*. Disentangling space from time by freeing garbage is not enough. One also needs to ensure that non-garbage space is not redundant, by avoiding silly indirections, sometimes referred to as *space leaks*. This is achieved by the *environment unchaining* optimization, which at the moment of creating a closure checks whether it is simply referring to another closure (which happens when the term part of the closure is a variable), in which case it is short-cut.

Summing up, there are three main issues, the last of which actually composed by four sub-problems, for a total of six critical points. We tame them as follows:

(1) *Input representation overhead*: handled by the Scott encoding and the representation of λ-terms;
(2) *Work tape overhead*: handled by the disabling of data pointers;
(3) *Tape scrolling overhead*, composed by the following sub-points:
    (a) *Tape representation overhead*: handled by the mathematical representation of the input tape;
    (b) *Sub-term pointer addressing*: handled by the left addressing of sub-term pointers;
    (c) *Space awareness*, composed by the following sub-points:
        (i) *Space entangled with time / re-usability of space*: handled by eager garbage collection;
        (ii) *Space leaks / indirections*: handled by environment unchaining.

4. The λ-Calculus, Term Sizes, and Addresses

In this section, we define the λ-calculus and discuss the delicate aspects of how to measure to size of terms and notions of addresses for constructors in terms.

λ-Calculus. Let $\mathcal{V}$ be a countable set of variables. Terms of the *λ-calculus* Λ are given by:

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

Free and *bound variables* are defined as usual: $\lambda x.t$ binds x in t. Terms are considered modulo α-equivalence. Capture-avoiding (meta-level) substitution of all the free occurrences of x for u in t is noted $t\{x{\leftarrow}u\}$. The computational rule is *β-reduction*:

$$(\lambda x.t)u \quad \to_\beta \quad t\{x{\leftarrow}u\}$$

which can be applied anywhere in a λ-term. Here, a strategy $\to$ shall be a sub-relation of $\to_\beta$. Given a relation $\to$, its reflexive-transitive closure is noted $\to^*$, and a λ-term t is $\to$-normal if there are no u such that $t \to u$. A $\to$-sequence is a pair of $\to^*$-related terms, often noted $\rho : t \to^* u$, and it is *complete* if u is $\to$-normal.

The Constructor and Code Sizes of λ-Terms. The *(constructor) size* of a λ-term is defined as follows:

$$|x| := 1 \qquad |tu| := |t| + |u| + 1 \qquad |\lambda x.t| := |t| + 1$$

The *code size* $\|t\|$ of a λ-term t is instead bounded by $\mathcal{O}(|t| \log |t|)$. The idea is that, when terms are explicitly represented, variables are some abstract kind of pointer (de Bruijn indices/levels, names, or actual pointers to the syntax tree), of size logarithmic in the number $|t|$ of constructors of t. Then a term with n constructors requires space $\mathcal{O}(n \log n)$ to be represented. For our study, it is important to stress the difference between the constructor size $|t|$ and the code size $\|t\|$, because given a binary input string i, at first sight its encoding t_i as a λ-term satisfies $|t_i| = \Theta(|i|)$ and $\|t_i\| = \mathcal{O}(|i| \log |i|)$, and so $\|t_i\|$ has an additional (unreasonable) logarithmic factor. We refer to this issue as *the code-constructor gap*.

Notions of Space and Size. The relevance of the code-constructor gap depends on the notion of space under consideration. If one adopts ink space (that is, the max size of λ-terms) as cost model, and thus is concerned with at least *linear* space, the gap is relevant, because the size of terms can indeed have an additional unreasonable logarithmic factor.

In Section 9, we shall encode strings in the λ-calculus using the Scott encoding, which has the property that, with respect to some concrete representations of terms (such as de Bruijn indices), variable pointers have *constant size*, so that $\|t_i\| = |t_i| = \Theta(|i|)$, thus removing the code-constructor gap.

If instead one studies logarithmic space, and thus adopts a low-level cost model where the input is separated from the work space, the gap is—perhaps surprisingly—less relevant. It is indeed perfectly possible that the encoding of the input string i has size $\mathcal{O}(|i| \log |i|)$, because the size of the input is not counted for the space usage, and still the space cost is logarithmic in $|i|$. What is important, indeed, is that pointers to the input—that are inevitably used if the space complexity is logarithmic—have size $\mathcal{O}(\log |i|)$ (rather than $\mathcal{O}(\log(|i| \log |i|))$), which is easy to obtain, as we explain in the next paragraph.

Since log-sensitivity allows us to use term representations also when the encoding of strings suffers from the code-constructor gap, we do not fix a concrete representation of terms. Such a relaxed approach is harmless, because, as we mentioned, there is at least one term representation (namely, de Bruijn indices) for which strings are encoded with no gap. It is however convenient because it allows us to use names rather than indices, improving readability, while obtaining more general results at the same time.

Addresses in Terms. We shall need addresses in terms for two purposes: comparing constructors (the reason is explained in the next section) and sub-term pointers. We then adopt two address schemes, as the two purposes need different properties. For constructor comparisons we adopt *tree addresses*, while for sub-term pointers we rely on *left addresses*, as we now explain.

It is standard that a constructor in a term can be identified via a path in the syntax tree of the term, and that such a path can be described as a binary address. We shall use this notion of tree address for comparing constructors in terms. For that, we also need to fix a way of comparing variables. We choose it to be de Bruijn indices. Note that this choice does not necessarily force the term representation itself to be de Bruijn indices, because given another representation one can usually compute the de Bruijn index of a variable in logarithmic space.

Definition 4.1 (Tree addresses)**.** A *tree address* a is a binary string. The constructor $t|_a$ of a closed term t at a is given by the following partial function defined by structural induction:

$$\begin{array}{rclcrcl} x|_\varepsilon & := & \mathsf{dB}(x) & \qquad & x|_{b\cdot a} & := & \bot \\ (tu)|_\varepsilon & := & @ & & (tu)|_{0\cdot a} & := & t|_a \\ (\lambda x.t)|_\varepsilon & := & \lambda & & (tu)|_{1\cdot a} & := & u|_a \\ & & & & (\lambda x.t)|_{b\cdot a} & := & t|_a \end{array}$$

where $b \in \{0,1\}$, $\mathsf{dB}(x)$ is the de Bruijn index corresponding to x (written in binary) if the whole term t were written using de Bruijn indexes, and $\bot$ denotes that $t|_a$ is undefined.

Tree addresses are a convenient way to point at a constructor or a sub-term. Unfortunately, they are not in general logarithmic in the size of the term. It is enough to consider terms the tree structure of which is linear (such as n applications of the identity to itself), so that the structural address of some constructors is linear in the size of the term.

We then consider also a second notion, *left addresses*.

Definition 4.2 (Left address)**.** Let t be a λ-term and c be a constructor of t (identified by a context or by a tree address). The left address of c in t is simply the index (written in binary) of the constructor in the enumeration of the nodes generated by an in-order visit of its syntactic tree, that is, a visit that:

(1) On applications rw, it first enumerates the constructors in the left sub-tree r, then the application @, and last the constructors in the right sub-tree w, recursively;
(2) On abstractions $\lambda x.r$, it first enumerates the abstraction λx and then the constructors in r.

For instance, the constructors of $x((\lambda y.z)w)$ are enumerated in the following order x, @, λy, z, @, and w.

Left addresses are always logarithmic in the constructor size of a term (even when the code size is bigger than the constructor size), and they are nothing else than the left-to-right position of the constructor in the string representation of the λ-term, whence their name.

Another property of left addresses that shall be crucial in our complexity analysis is that for an application ur, the size of the addresses in u is independent from r. Note that this would hold also if one enumerates the constructors according to a visit in pre-order, but not in post-order.

We avoid left addresses for comparisons because we shall have to compare constructors extracted from machine states, and the extraction from states is considerably simpler if done with respect to tree addresses.

5. Reasonable Preliminaries

In the study of reasonable cost models for the λ-calculus, it is customary to show that the λ-calculus simulates TMs reasonably, and conversely that the λ-calculus can be simulated reasonably by TMs[3] *up to sharing*. Since space is more delicate than time, we fix the involved theories and their cost measures carefully.

Turing Machines. We adopt TMs working on the boolean alphabet $\mathbb{B} := \{0, 1\}$. For a study of logarithmic space complexity, one has to distinguish *input* space and *work* space, and to *not* count the input space for space complexity. On TMs, this amounts to having *two* tapes, a read-only *input tape* on the alphabet $\mathbb{B}_{\mathsf{I}} := \{0, 1, \mathsf{L}, \mathsf{R}\}$, where L and R are delimiter symbols for the start and the end of the input binary string, and an ordinary read-and-write *work tape* on the boolean alphabet extended with a blank symbol $\mathbb{B}_{\mathsf{W}} := \{0, 1, \square\}$. To keep things simple, we use TMs without any output tape. The machine rather has two final states q_0 and q_1 encoding a boolean output—there are no difficulties in extending our results to TMs with an output tape.

Let us call these machines *log-sensitive TMs*. A log-sensitive TM M computes the function $f : \mathbb{B}^* \to \mathbb{B}$ by a sequence of transitions $\rho : C^M_{\mathtt{in}}(i) \to^n C^M_{\mathtt{fin}}(f(i))$ where $i \in \mathbb{B}^*$, $C^M_{\mathtt{in}}(i)$ is an initial configuration of M with input i and $C^M_{\mathtt{fin}}(f(i))$ is a final configuration of M on the final state $q_{f(i)}$. We define the time of the sequence ρ as $T_{\mathrm{TM}}(\rho) := n$ and the space $S_{\mathrm{TM}}(\rho)$ of ρ as the maximum number of cells of the work tape used by ρ.

While we shall study in detail the encoding of TMs in the λ-calculus, which is the difficult direction, we are not going to lay out the details of the simulation of the λ-calculus on TMs, as it is conceptually simpler. We shall provide an abstract machine for the λ-calculus and study its complexity using standard considerations for algorithmic analysis, but without giving the details of the simulation.

Reasonable Cost Models. We give the simulations and the bounds we shall consider for reasonable cost models for the λ-calculus. The case of the λ-calculus is peculiar because its simulation on TMs is up to sharing, that is, it computes a *compact representation* of the result, not the encoding of the result itself. Therefore, some further conditions about such a representation are required, expressed via a decoding function. The one about space is new and motivated by the paragraphs after the definition.

Definition 5.1 (Reasonable cost model for the λ-calculus)**.** A *reasonable time (resp. space) cost model* for the λ-calculus is an evaluation strategy $\to$ together with a function T_λ (resp. S_λ) from complete $\to$-sequences $\rho : t \to^* u$ to $\mathbb{N}$ such that:

- *TMs to λ*: there is an encoding $\bar{\cdot}$ of binary strings and TMs into the λ-calculus such that if the run σ of a TM M on input i ends on a state q_b with $b \in \mathbb{B}$, then there is

[3] In the study of reasonable time, random access machines (RAMs) rather than TMs are usually the target of the encoding of the λ-calculus, because RAMs are reasonable and easier to manage for the time analysis of algorithms. For our study, it shall be simpler to use TMs. Therefore, we avoid references to RAMs in the paper.

a complete sequence $\rho : \overline{M}\,\overline{i} \to^* \overline{b}$ such that $T_\lambda(\rho) = \mathcal{O}(poly(T_{\mathrm{TM}}(\sigma), |i|))$ (resp. space $S_\lambda(\rho) = \mathcal{O}(S_{\mathrm{TM}}(\sigma) + \log|i|)$). Moreover:
  - *Complexity of the encoding:* computing $\overline{i}$ is done in time $\mathcal{O}(poly(|i|))$, and space $\mathcal{O}(\log(|i|))$ (measured on a reasonable model).
- λ *to TMs*: there is an encoding $\underline{\cdot}$ of λ-terms into binary strings, a TM M, and a decoding $\cdot\!\downarrow$ of final configurations for M such that if $\rho : t \to^* u$ is a complete sequence, then the execution σ of M on input $\underline{t}$ produces a final configuration C such that $C\!\downarrow = u$ in time $T_{\mathrm{TM}}(\sigma) = \mathcal{O}(poly(T_\lambda(\rho), |t|))$ (resp. space $S_{\mathrm{TM}}(\sigma) = \mathcal{O}(S_\lambda(\rho) + \log|t|)$). Moreover:
  - *Complexity of the encoding:* computing $\underline{t}$ is done in time $\mathcal{O}(poly(|t|))$, and space $\mathcal{O}(\log(|t|))$ (measured on a reasonable model).
  - *Polytime result equality*: for all final configurations C' of M, testing whether $C\!\downarrow = C'\!\downarrow$ can be done in time $\mathcal{O}(poly(|C|, |C'|))$ (measured on a reasonable model).
  - *Logarithmic space constructor equality*: given the initial term t, the final configuration C, and a tree address a, computing $(C\!\downarrow)|_a$ has space complexity $\mathcal{O}(\log|t| + \log|C| + \log|a|)$ (measured on a reasonable model).

Explaining the Complexity of the Encoding Conditions. The two requirements on the complexity of the encodings ensure that no unreasonable overhead is hidden inside the enconding functions. In particular, in going from λ to TMs, we do not want the encoding function to "execute" the encoded λ-term.

Note however that these conditions are somewhat vague, and necessarily so: they ask complexity bounds for the encoding of a computational theory into another, but in which theory are those bounds to be taken? To be precise, one should be able to express both theories inside a third reasonable theory... but how has this third theory been shown reasonable? There is no real way out, because the study of reasonable cost models is *preliminary* to the definition of notions of complexity valid *across theories*. We added the conditions anyway to intuitively point out that encodings doing more than just translating from one formalism to the other should be ruled out.

Explaining the Equality Conditions. Polytime result equality ensures that compact results (i.e. final configurations) can be compared for equality of the underlying unshared result without having to unshare them, which might take exponential time. The requirement is with respect to another compact result because polynomiality in $|q|$ and $|u|$ is useless if $|u|$ is exponential in $|q|$. For sharing as explicit substitutions/environments, polytime result equality was first proved by Accattoli and Dal Lago [ADL12], and then showed linear (on random access machines with constant-time pointers manipulation) by Condoluci et al. [CAC19]. The *logarithmic space constructor equality* requirement is new, and ensures that the compact representation allows one to access atomic parts of the result as if the result were unshared. To motivate it, consider a result u that is exponentially bigger than its compact representation as a final configuration C, which is what happens with size exploding families. The requirement ensures that to read atomic parts of u out of C there is no need to unfold the sharing in C, which might require space exponential in $|C|$. Moreover, the requirement essentially states that, to compute $(C\!\downarrow)|_a$, one has to use only a constant amount of pointers to t, C, and a. In particular, the requirement is independent of the cost of evaluation. This aspect rules out degenerated simulations where a part of the work is hidden in the representation of the result (think of the simulation that does nothing and leaves all the work to the de-compactification of the result). In our case, the proof

that constructor equality can be tested in logarithmic space shall be straightforward, which contrasts with polytime result equality, that requires non-trivial algorithms.

Single Inputs, not Input Lengths. Note that our cost assignments concern *runs*, thus a single input of a given length, rather than the max over all inputs of the same length, as it is usually done in complexity. The study of cost models is somewhat finer, the max can be considered afterwards.

6. λ-Calculus and Abstract Machines

A term is *closed* when there are no free occurrences of variables in it. The operational semantics—that is, the evaluation strategy—that we adopt in most of the paper is *weak head evaluation* $\to_{wh}$, defined as follows:

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

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

Abstract Machines Glossary. In this paper, an *abstract machine* $\mathrm{M} = (Q, \to, \mathsf{init}(\cdot), \cdot\!\downarrow)$ is a transition system $\to$ over a set of states, noted Q, together with two functions:

- *Initialization.* $\mathsf{init}(\cdot) : \Lambda \to Q$, turning λ-terms into states;
- *Decoding.* $\cdot\!\downarrow : Q \to \Lambda$, turning states into λ-terms and such that $\mathsf{init}(t)\!\downarrow = t$ for every λ-term.

A state $q \in Q$ is composed by the *(immutable) code* t_0, the *active term* t, and some data structures. Since the code never changes, it is usually omitted from the state itself, focussing on *dynamic states* that do not mention the code. A state q is *initial* for t if $\mathsf{init}(t) = q$. In this paper, $\mathsf{init}(t)$ is always defined as the state having t as both the code and the active term and having all the data structures empty. Additionally, the code t shall always be *closed*, without further mention. A state is *final* if no transitions apply. A *run* $\rho : q \to^* q'$ from state q to state q' is a possibly empty sequence of transitions, the length of which is noted $|\rho|$. If a and b are transitions labels (that is, $\to_a \subseteq \to$ and $\to_b \subseteq \to$) then $\to_{a,b} := \to_a \cup \to_b$, $|\rho|_a$ is the number of a transitions in ρ, and $|\rho|_{\neg a}$ is the size of transitions in ρ that are not $\to_a$. An *initial run* is a run from an initial state $\mathsf{init}(t)$, and it is also called *a run from* t. A state q is *reachable* if it is the target state of an initial run. A *complete run* is an initial run ending on a final state.

Abstract Machines and Abstract Implementations. Abstract machines do not specify how the (abstract) data structures of the machine are meant to be realized. In general an abstract machine can be implemented in various ways, inducing different, possibly incomparable performances. Therefore, it is not really possible to study the complexity of the machine without some assumptions about the implementation of its data structures. The study of reasonable space requires to take into account the use, and especially the *size*, of pointers, which is instead usually omitted in the coarser study of reasonable time. In that context, indeed, pointers are assumed to be manipulable in constant time, which is safe because the omitted logarithmic factors are irrelevant for the required polynomial overhead. The more constrained study of space instead requires to clarify the cost of pointers.

Closures	Environments	Stacks	States
$c ::= (t, e)$	$e ::= \epsilon \mid [x \leftarrow c] \cdot e$	$\pi ::= \epsilon \mid c \cdot \pi$	$q ::= (t, e, \pi)$

Term	Env	Stack		Term	Env	Stack	
tu	e	π	$\to_{\mathsf{sea}}$	t	e	$(u, e) \cdot \pi$	
$\lambda x.t$	e	$c \cdot \pi$	$\to_{\beta}$	t	$[x \leftarrow c] \cdot e$	π	
x	e	π	$\to_{\mathsf{sub}}$	u	e'	π	if $e(x) = (u, e')$

Figure 1: Data structures and transitions of the Outlined KAM.

Switching to such a level of detail, apparently innocent gaps between the specification of a machine and how it is going to be implemented suddenly become relevant.

To account for these subtleties, we specify for every construct of the abstract machine the space that it requires, and for every transition the time that it takes, both asymptotically. The adoption of such an *abstract implementations* is in our opinion a contribution of this paper towards a more solid theory of abstract machines.

Definition 6.1 (Abstract implementation)**.** Let M be an abstract machine and $\rho : \mathsf{init}(t_0) \to^* q$ an initial run for M. An abstract implementation I for M is the assignment of asymptotic space costs $|\cdot|^I_{\mathsf{sp}}$ for every component of q and of asymptotic time costs $|\cdot|^I_{\mathsf{tm}}$ for every transition from q.

Assigning costs to the state components provides the space cost $|q|^I_{\mathsf{sp}}$ of each state q, by summing over all components.

Definition 6.2 (Space and time of runs)**.** Let $\rho : q_0 \to^k q_k$ be an initial run of an abstract machine M and I an abstract implementation for M.

(1) The I-space cost of ρ is $|\rho|^I_{\mathsf{sp}} := \max_{q \in \rho} |q|^I_{\mathsf{sp}}$.
(2) The I-time cost of ρ is $|\rho|^I_{\mathsf{tm}} := \sum_{i=0}^{k-1} |q_i \to q_{i+1}|^I_{\mathsf{tm}}$.

A Technical Remark. Note that abstract implementations do not specify the space cost of transitions $q \to q'$. According to the space cost for a run, such a cost has to be the difference $|q'|^I_{\mathsf{sp}} - |q|^I_{\mathsf{sp}}$ in space between the two involved states, which can be inferred by the size of the states. Therefore, it need not be specified by an abstract implementation. Note also a subtlety: implementing a transition might require auxiliary space temporarily exceeding both $|q|_{\mathsf{sp}}$ and $|q'|_{\mathsf{sp}}$, which we are not accounting for. The point is that for all the machines considered in this paper, such a temporary extra space is bounded by the current space (that is, $|q|^I_{\mathsf{sp}}$), and taking it into account would affect the globally used space only linearly, which is reasonable for space. Therefore, the auxiliary use of space can, and shall, be safely omitted.

7. The KAM and its Implementations

The Krivine abstract machine [Kri07] is a standard environment-based machine for Closed CbN, often defined as in Fig. 1. We refer to it as *the Outlined KAM*, to distinguish it from forthcoming variants and to stress that its specification is too abstract, as the role of low-level

aspects such as the immutable initial code, pointers, and how the abstract data structures are concretely implemented is not made explicit, while it is essential for complexity analyses.

The machine evaluates closed λ-terms to weak head normal form via three transitions, the union of which is noted $\to_{\mathrm{OutKAM}}$:

- $\to_{\mathsf{sea}}$ looks for redexes descending on the left of topmost applications of the active term, accumulating arguments on the stack;
- $\to_{\beta}$ fires a β redex (given by an abstraction as active term having as argument the first entry of the stack) but delays the associated meta-level substitutions, adding a corresponding explicit substitution to the environment;
- $\to_{\mathsf{sub}}$ is a form of micro-step substitution: when the active term is x, the machine looks up the environment and retrieves the delayed replacement for x.

The abstract data structures used by the Outlined KAM are *local environments*, *closures*, and a *stack*. *Local environments*, which we shall simply refer to as *environments*, are defined by mutual induction with *closures*. The idea is that every (potentially open) term t in a dynamic state comes with an environment e that *closes* it, thus forming a closure $c = (t, e)$, and, in turn, environments are lists of entries $[x{\leftarrow}c]$ associating to each open variable x of t, a closure c i.e., essentially, a closed term. The *stack* simply collects the closures associated to the arguments met during the search for β-redexes.

A dynamic state q of the Outlined KAM is the pair (c, π) of a closure c and a stack π, but we rather see it as a triple (t, e, π) by spelling out the two components of the closure $c = (t, e)$. Initial dynamic states of the Outlined KAM are defined as $\mathsf{init}(t_0) := (t_0, \epsilon, \epsilon)$ (where t_0 is a closed λ-term, and also the code). The decoding of closures and states is as follows:

$$\begin{array}{rrcl@{\qquad}rcl} \textsc{Closures} & (t,\epsilon)\!\downarrow & := & t & (t,[x{\leftarrow}c]{\cdot}e)\!\downarrow & := & (t\{x{\leftarrow}c\!\downarrow\},e)\!\downarrow \\ \textsc{States} & (t,e,\epsilon)\!\downarrow & := & (t,e)\!\downarrow & (t,e,\pi{\cdot}c)\!\downarrow & := & (t,e,\pi)\!\downarrow c\!\downarrow \end{array}$$

Basic Qualitative Properties. Some standard facts about the Outlined KAM follow. Let $\rho : \mathsf{init}(t_0) \to_{\mathrm{OutKAM}}{}^{*} q$ be a run.

- *Closures-are-closed invariant*: if the code t_0 is closed (that is the only case we consider here) then every closure (u, e) in q is *closed*, that is, for any free variable x of u there is an entry $[x{\leftarrow}c]$ in e, and recursively so for the closures in e. Thus $(u, e)\!\downarrow$ is a closed term, whence the name *closures*.
- *Final states*: the previous fact implies that the machine is never stuck on the left of a $\to_{\mathsf{sub}}$ transition because the environment does not contain an entry for the active variable. Final states then have shape $(\lambda x.u, e, \epsilon)$.

Theorem 7.1 (Implementation)**.** *The Outlined KAM implements Closed CbN, that is, there is a complete $\to_{wh}$-sequence $t \to_{wh}^{n} u$ if and only if there is a complete run $\rho : \mathsf{init}(t) \to_{\mathrm{NaKAM}}{}^{*} q$ such that $q\!\downarrow = u$ and $|\rho|_{\beta} = n$.*

The proof of the facts and of the theorem are standard and omitted. Similar statements hold for all the variants of the KAM that we shall see, with similar proofs which shall be omitted as well (we shall also omit the decoding of the variants of the KAM).

The key point is that there is a bijection between $\to_{wh}$ steps and $\to_{\beta}$ transitions, so that we can identify the two. Moreover, the number of $\to_{wh}$ steps is a reasonable cost model for time, as first proved by Sands et al. [SGM02].

Quantitative Properties. We recall also some less known *quantitative* facts, for runs as above, from papers by Accattoli and co-authors [ADL12, ABM14, AB17]. The aim is to bound quantities relative to the run ρ and the reachable state q. The bounds are given with respect to two parameters: the size $|t_0|$ of the code and the number $|\rho|_\beta$ of β-transitions, which, as mentioned, is an abstract notion of time for Closed CbN.

- *Number of transitions*: the number $|\rho|_{\mathsf{sub}}$ of sub transitions in ρ is bounded by $\mathcal{O}(|\rho|_\beta^2)$, and there are terms on which the bound is tight. The number $|\rho|_{\mathsf{sea}}$ of sea transitions is bounded by $\mathcal{O}(|\rho|_\beta^2 \cdot |t_0|)$, but on complete runs the bound improves to $\mathcal{O}(|\rho|_\beta)$.
- *Sub-term invariant*: every term u in every closure (u, e) in every reachable state is a literal (that is, *not* up to α-renaming) sub-term of the code t_0 . Therefore, in particular $|u| \leq |t_0|$.
- *The length of a single environment*: the number of entries in a single environment is bounded by the size $|t_0|$ of the code.
- *The number of environments*: the number of distinct environments in q is bounded only by $|\rho|_\beta$.
- *The length of the stack*: the length of the stack in q is bounded by $\mathcal{O}(|\rho|_\beta^2 \cdot |t_0|)$.

Sub-Term Pointers and Data Pointers: the Linked KAM. The Outlined KAM is usually implemented using an immutable initial code and *two* forms of pointers, obtaining what here refer to as the *Linked KAM*:

(1) *Sub-term pointers* p_t, p_u: the initial term t_0 provides the initial immutable code. The essential sub-term invariant mentioned above allows one to represent the active term and the terms u in every closure (u, e) of every reachable state with a pointer p_u to t_0, instead that with a copy of u.
(2) *Data (structure) pointers* $p_e, p_{e'}$: to ensure that the duplication of the environment e in transition $\to_{\mathsf{sea}}$ can be implemented efficiently (in time), environments are shared so that what is duplicated is just a pointer to an environment, and not the environment itself. This means that environments entries are stored in the *heap* (or global environment), a new data structure which is simply a store, and that environments are pointers to heap entries.

The Linked KAM is in Fig. 2; explanations and comments follow.

- *Environment entries and look-up*. Note that environment entries now have shape $[p_{\lambda x.t} \leftarrow c]$ instead of $[x \leftarrow c]$, since they pair the pointer $p_{\lambda x.t}$ to the binder of x (with the closure) rather than the variable x. Consequently, the function $e(x)$ looking up environments is now replaced by a function $e(p_x)$ acting on pointers, which first retrieves the pointer $p_{\lambda x.t}$ of the binder of p_x and then looks up the list structure of e in the heap for the closure $(p_u, p_{e'})$ associated to $p_{\lambda x.t}$.
- *Stack pointers*. The representation of stacks can also be (data-)pointer-based or flat. In the Linked KAM, we assume that stacks are flat. In extensions of the λ-calculus with control operators (which are not treated in this paper), stacks can be duplicated and so therein it would be more natural to have pointer-based representations of stacks, to enable their sharing.
- *Sub-term pointers*. Note that having made pointers explicit is still not concrete enough for precise complexity analyses, in the case of sub-term pointers. Using tree addresses (Def. 4.1), one can obtain p_t and p_u from p_{tu} (as required by transition $\to_{\mathsf{sea}}$) in constant time, but tree addresses in general have size $\mathcal{O}(|t_0|)$. Using left addresses (Def. 4.1),

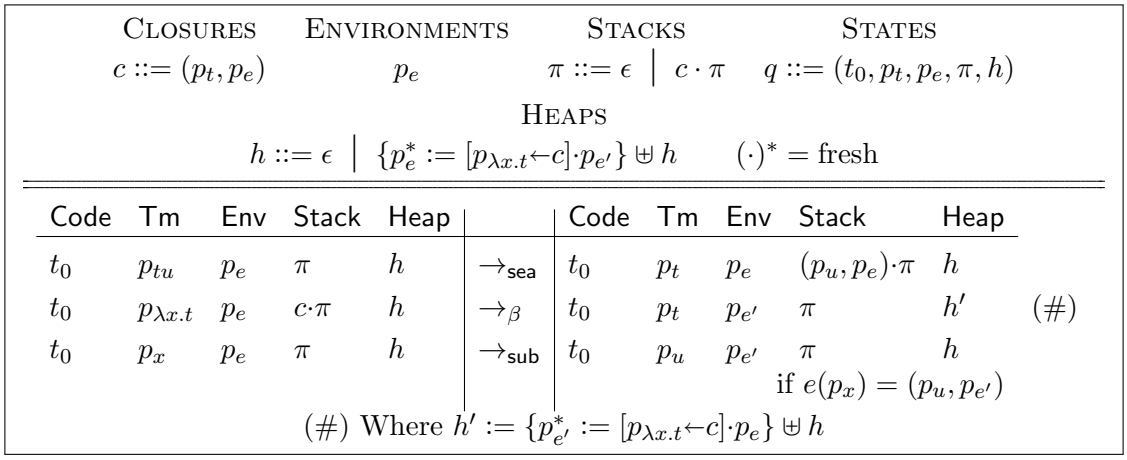


Figure 2: Data structures and transitions of the Linked KAM.

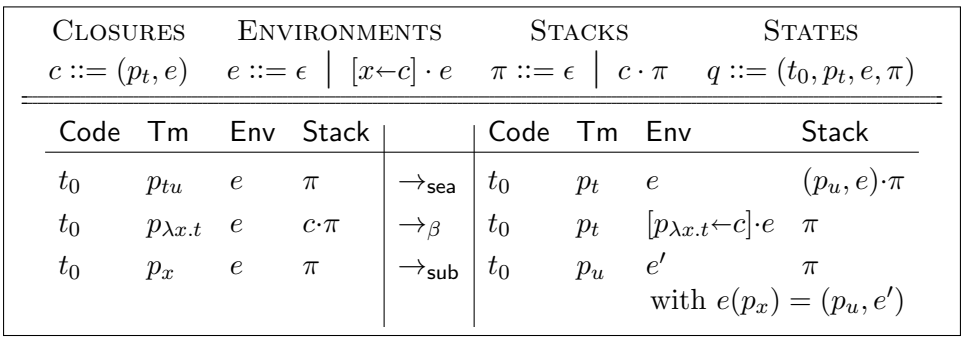


Figure 3: Sub-Term KAM.

obtaining p_t and p_u requires instead more time, namely $\mathcal{O}(|t_0|)$, but left addresses use less space, namely $\mathcal{O}(\log(|t_0|))$.

There is a big difference between sub-term and data pointers. As already mentioned, sub-term pointers can be crafted as to have size $\mathcal{O}(\log |t_0|)$. For the present discussion they are *space-friendly*, because their size does not depend on the length of the run. Data pointers, on the other hand, are *space-hostile*, because (as recalled above) the number of environments is bounded only by $|\rho|_\beta$, that is, *(abstract) time*. Data pointers have thus size $\mathcal{O}(\log |\rho|_\beta)$, entangling space with time, which is unreasonable for space. In the next section we shall add garbage collection, which disentangles space from time, but data pointers shall still add a $\mathcal{O}(\log E)$ overhead, where E is the number of environment entries, which is excessive for space reasonability. Therefore, we now remove data pointers altogether, turning to a *flat* representation of environments (and stacks), as explained in the next paragraph.

Sub-Term and Naive KAM. Summing up, as a reference we want a KAM adopting sub-term pointers but avoiding data pointers. Such a KAM can be presented in two ways. The first one is the Sub-Term KAM in Fig. 3, where the immutable code and sub-terms pointers are explicit, while environments are presented as in the Outlined KAM, to suggest that they are flat rather than linked via data pointers.

We shall however adopt a different presentation, for two reasons. Firstly, the Space KAM of the next section needs further tweaks of the KAM (namely garbage collection and environment unchaining) and, to avoid a too heavy treatment, we prefer to hide sub-term

Outlined KAM

Closures	Environments	Stacks	States
$c ::= (t, e)$	$e ::= \epsilon \mid [x \leftarrow c] \cdot e$	$\pi ::= \epsilon \mid c \cdot \pi$	$q ::= (t, e, \pi)$

Term	Env	Stack		Term	Env	Stack	
tu	e	π	$\to_{\mathsf{sea}}$	t	e	$(u, e) \cdot \pi$	
$\lambda x.t$	e	$c \cdot \pi$	$\to_{\beta}$	t	$[x \leftarrow c] \cdot e$	π	
x	e	π	$\to_{\mathsf{sub}}$	u	e'	π	if $e(x) = (u, e')$

Naive Abstract Implementation (with respect to initial code t_0)

Terms	Closures	States
$\lvert u \rvert := \log \lvert t_0 \rvert$	$\lvert (u, e) \rvert := \lvert u \rvert + \lvert e \rvert$	$\lvert (u, e, \pi) \rvert := \lvert u \rvert + \lvert e \rvert + \lvert \pi \rvert$

Stacks	Environments	Transitions $q \to q'$
$\lvert \epsilon \rvert := 0$	$\lvert \epsilon \rvert := 0$	$\lvert \to \rvert_{\mathsf{tm}} := \mathit{poly}(\lvert q \rvert)$
$\lvert c \cdot \pi \rvert := \lvert c \rvert + \lvert \pi \rvert$	$\lvert [x \leftarrow c] \cdot e \rvert := \lvert x \rvert + \lvert c \rvert + \lvert e \rvert$	

Figure 4: Naive KAM = Outlined KAM + Naive Abstract Implementation.

pointers and the immutable code. We hope that the discussions of this section have clarified how these two points are managed. Secondly, as already mentioned, the Sub-Term KAM is still not detailed enough for unambiguous complexity analysis. Therefore, we rather revert to the Outlined KAM *but* we pair it with an abstract implementation specifying the cost of the involved components and transitions as intended for the Sub-Term KAM (with left sub-term addresses). The pair of the Outlined KAM and the naive abstract implementation is referred to as the *Naive KAM*, defined in Fig. 4, the transition relation of which is noted $\to_{\mathrm{NaKAM}}$.

Naive Abstract Implementation. Implementing the Naive KAM without data pointers means that environments and stacks are implemented as unstructured *strings*, in a linear syntax. We abstract from the actual encoding, what we retain is the abstract implementation in Fig. 1, which captures its essence. Sub-terms are assumed to be of size $\log(|t_0|)$, which is obtainable by adopting left addresses.

The time cost of all $\to_{\mathrm{NaKAM}}$ transition depends polynomially on the size of the whole source state $|q|$, because the lack of data sharing forces to use a new string for the new stack and the new environment; in particular, transition $\to_{\mathsf{sea}}$ requires to copy the whole string representing e. To be precise, one could develop a finer analysis, thus obtaining slightly better bounds, but this would require entering in the details of the implementation and would not give a substantial advantage. As we shall see, indeed, the Naive KAM is unreasonable for abstract time (Prop. 10.7). Via an analysis of the Naive KAM execution of the encoding of TMs, it shall turn out that also the space usage of the Naive KAM is unreasonable (Prop. 9.1). A space-reasonable refinement of the Naive KAM is the topic of the next section.

Term	Env	Stack		Term	Env	Stack	
tx	e	π	$\to_{\mathsf{sea_v}}$	t	$e\|_t$	$e(x)\cdot\pi$	
tu	e	π	$\to_{\mathsf{sea}_{\neg\mathsf{v}}}$	t	$e\|_t$	$(u, e\|_u)\cdot\pi$	if $u \notin \mathcal{V}$
$\lambda x.t$	e	$c\cdot\pi$	$\to_{\beta_\mathsf{w}}$	t	e	π	if $x \notin \mathsf{fv}(t)$
$\lambda x.t$	e	$c\cdot\pi$	$\to_{\beta_{\neg\mathsf{w}}}$	t	$[x\leftarrow c]\cdot e$	π	if $x \in \mathsf{fv}(t)$
x	e	π	$\to_{\mathsf{sub}}$	u	e'	π	if $e(x) = (u, e')$

where $e|_t$ denotes the restriction of e to the free variables of t.

Figure 5: Transitions of the Collecting KAM, which is the abstract layer of the Space KAM.

8. The Space KAM

Here we define an abstract space optimization of the Outlined KAM, dubbed *Collecting KAM*, which when paired with the same abstract implementation of the Naive KAM shall give the Space KAM. The Collecting KAM is derived from the Outlined KAM by adding two modifications aimed at space efficiency: namely *unchaining* and *eager garbage collection*.

Unchaining. Environment unchaining is a folklore optimization for abstract machines bringing speed-ups with respect to both time and space, used for instance by Sands et al. [SGM02], Wand [Wan07], Friedman et al. [FGSW07], and Sestoft [Ses97]. Its first systematic study is by Accattoli and Sacerdoti Coen in [ASC17], with respect to time. The optimization prevents the creation of chains of *renamings* in environments, that is, of delayed substitutions of variables for variables, of which the simplest shape in the KAM is:

$$[x_0\leftarrow(x_1, [x_1\leftarrow(x_2, [x_2\leftarrow \ldots])])]$$

where the links of the chain are generated by β-redexes having a variable as argument. On some families of terms, these chains keep growing, leading to the quadratic dependency of the number of transitions from $|\rho|_\beta$.

Eager Garbage Collection. Besides the malicious chains connected to unchaining, the Outlined KAM is not parsimonious with space also because there is no garbage collection (shortened to GC). In transition $\to_{\mathsf{sub}}$, the current environment is discarded, so something is collected, but this is not enough. It is thus natural to modify the machine as to maximize GC and space re-usage, that is, as to perform it *eagerly*.

The Collecting KAM. The Outlined KAM optimized with both eager GC and unchaining (both optimizations are mandatory for space reasonability) is here called Collecting KAM and it is defined in Fig. 5. The data structures, namely closures and (local) environments, are defined as before—the novelty concerns the machine transitions only. Unchaining is realized by transition $\to_{\mathsf{sea_v}}$, while eager garbage collection is realized by transition $\to_{\beta_\mathsf{w}}$, which collects the argument if the variable of the β redex does not occur, and by transitions $\to_{\mathsf{sea_v}}$ and $\to_{\mathsf{sea}_{\neg\mathsf{v}}}$, by restricting the environment to the occurring variables, when the environment is propagated to sub-terms. As a consequence, we obtain the following invariant.

Lemma 8.1 (Environment domain invariant)**.** *Let q be a Collecting KAM reachable state. Then $\mathsf{dom}(e) = \mathsf{fv}(t)$ for every closure (t, e) in q.*

Because of the invariant, which concerns also the closure given by the active term and the local environment of the state, the substitution transition $\to_{\mathsf{sub}}$ simplifies as follows:

Term	Env	Stack		Term	Env	Stack
x	$[x{\leftarrow}(u,e)]$	π	$\to_{\mathsf{sub}}$	u	e	π

Sub-Term Pointers and Abstract Implementation: the Space KAM. We now add an abstract implementation to the Collecting KAM, finally obtaining the Space KAM. The abstract implementation to be added is the same of the Naive KAM, but for a crucial point. For the Naive KAM, the size of a term $|u|$ is given by $\log|t_0|$, and we said that this is obtainable via left addresses. For the Space KAM, we need a finer approach, or rather a finer definition.

Definition 8.2 (Space KAM)**.** The Space KAM is defined as the Collecting KAM together with the abstract implementation I defined as for the Naive KAM except for the size of sub-terms, which is redefined as $|u|^I_{\mathsf{sp}} := |\mathtt{ladd}(u,t_0)|$, where $\mathtt{ladd}(u,t_0)$ is the left address of u in the initial code t_0.

In the complexity analysis of the encoding of TMs we shall use the fact that, for some sub-terms, $|\mathtt{ladd}(u,t_0)|$ is of size $\mathcal{O}(1)$ rather than $\mathcal{O}(\log|t_0|)$, and this shall be crucial for the space reasonability result. In fact, other addressing schemes might work as well. What is important for our result is that the size of sub-term pointers is always $\mathcal{O}(\log|t_0|)$ and that, for an application ur, the size of the addresses in u is independent of r, which shall be used to infer that some pointers have size $\mathcal{O}(1)$ because of how the encoding of TMs is built.

Note that left addresses do not respect the *tree locality* of terms, in the sense that in tu the first constructors of t and u are not the address of the root application ± 1. Their addresses can however be retrieved in time polynomial and space logarithmic in $|t_0|$, which is what is important. Note that such a non-locality of left addresses is one of the ways in which time is traded for space in the Naive/Space KAM: manipulating terms via left addresses requires a (polynomial) time overhead with respect to use actual pointers to the code.

For the abstract implementation, it is fine to keep the same time bounds used for the Naive KAM, because the garbage collector has a time cost which however stays within the polynomial (in the size of the states) cost of the transitions. It is mandatory that it is implemented by naively and repeatedly checking whether variables occur, and *not* via pointers or counters, as they would add an unreasonable space overhead. This fact is implicit in using the same abstract implementation of the Naive KAM, as a less naive GC would alter the space requirements.

Space Cost and Closure Space. When we defined the space cost of a generic machine run (Def. 6.2), we considered the max over the size of states. The size of states is a very concrete notion. Now that we have defined the Collecting KAM and the Space KAM, it is possible to define also a second, more abstract notion of space, here dubbed *closure space*. For ease of language, we define it on the Space KAM, but the abstract aspect of closure space is evident from the fact that it can already be defined on the Collecting KAM.

Definition 8.3 (Closure space)**.** Let $\rho : q_0 \to^k q_k$ be an initial run of the Collecting KAM and use $|q|_{\mathsf{cl}}$ for the number of closures in a state q. Then the closure space of ρ is defined as $|\rho|_{\mathsf{clsp}} := \max_{q\in\rho}|q|_{\mathsf{cl}}$.

Since concrete space is obtained by considering for each closure also the size of its sub-term pointer, one has $|\rho|_{\mathsf{clsp}} \leq |\rho|_{\mathsf{sp}} \leq \mathcal{O}(|\rho|_{\mathsf{clsp}} \cdot \log|t_0|)$. This is similar to what happens with abstract time (that is, the number of β-steps), where the actual time is more precisely bounded by abstract time times the size of the initial term. There is however an important difference. On the encoding of TMs, it shall turn out that $|\rho|^I_{\mathsf{sp}} \neq \Theta(|\rho|_{\mathsf{clsp}} \cdot \log|t_0|)$, because some sub-term pointers shall have size $\mathcal{O}(1)$ rather than $\mathcal{O}(\log|t_0|)$. Additionally, $\Theta(|\rho|_{\mathsf{clsp}} \cdot \log|t_0|)$ would not be a reasonable use of space. It is nonetheless useful to consider such an abstract notion of closure space, as the next paragraph shows.

Closure space and η-equivalence. We point out that the space consumption of the Space KAM is almost invariant with respect to η-expansion, thanks to unchaining. In particular, η-expansion preserves closure space, but not the concrete one: η-expanding t for n times preserves the number of closures but causes the size of sub-term pointers in each closure to grow by no more than $\log(n)$. Formally, let us define η-expansion as the function $\eta : \Lambda \to \Lambda$ such that:

$$\eta(t) := \lambda x.tx \quad \text{with } x \notin \mathsf{fv}(t)$$

Lemma 8.4. *Let t be a closed λ-term, and ρ_n the complete run from $(\eta^n(t), e, c \cdot \pi)$. Then $|\rho_n|_{\mathsf{clsp}} = |\rho_0|_{\mathsf{clsp}}$ and $|\rho_n|_{\mathsf{sp}} \in \mathcal{O}(\log(n) \cdot |\rho_0|_{\mathsf{sp}})$.*

Proof. We first prove that $(\eta^n(t), e, c \cdot \pi) \to^{2n}_{\mathrm{SpKAM}} (t, e, c \cdot \pi)$. We proceed by induction on n executing the Space KAM. The case $n = 0$ is trivial. Then, we consider the case $n = m + 1$.

Term	Env	Stack	
$\eta^{m+1}(t) := \lambda x.\eta^m(t)x$	e	$c \cdot \pi$	$\to_{\beta_{\neg \mathsf{w}}}$
tx	$[x \leftarrow c] \cdot e$	π	$\to_{\mathsf{sea_v}}$
$\eta^m(t)$	e	$c \cdot \pi$	$\to^{2n}_{\mathrm{SpKAM}}$ $(i.h.)$
t	e	$c \cdot \pi$	

We observe that no space is consumed during this transitions. About pointer size, let us call t_n the initial code. We have that the number of constructors of t_n is $|t_n| = 3n + |t_0|$ and thus its pointer size is $\log(3n + |t_0|)$. Then, let us call k the maximum number of closures stored in ρ_n (we have already observed that this is independent of n). Then

$$\begin{aligned} |\rho_n|_{\mathsf{sp}} &= k \cdot \log(3n + |t_0|) \leq k \cdot (\log(3n) + \log(|t_0|)) = k \cdot \log(3n) + k \cdot \log(|t_0|) \\ &\leq k \cdot \log(3n) \cdot \log(|t_0|) + k \cdot \log(|t_0|) = \log(3n) \cdot |\rho_0|_{\mathsf{sp}} + |\rho_0|_{\mathsf{sp}} \in \mathcal{O}(\log(n) \cdot |\rho_0|_{\mathsf{sp}}) \end{aligned}$$

□

9. Encoding and Moving over Strings

We now turn to the analysis of the encoding of TMs, taking as reference the one by Dal Lago and Accattoli based over the Scott encoding of strings [DLA17]. The first key step is understanding how to scroll Scott strings.

Encoding alphabets. Let $\Sigma = \{a_1, \dots, a_n\}$ be a finite alphabet. Elements of Σ are encoded in the λ-calculus in accordance to a fixed (but arbitrary) total order of the elements of Σ as follows:

$$\lceil a_i \rceil^{\Sigma} := \lambda x_1 . \dots . \lambda x_n . x_i \ .$$

Note that the representation of an element $\lceil a_i \rceil^{\Sigma}$ requires a number of constructors that is linear (and not logarithmic) in $|\Sigma| = n$. Since the alphabet Σ shall not depend on the input of the TM, however, the cost in space is actually constant.

Encoding strings. A string in $s \in \Sigma^*$ is represented by a term $\overline{s}^{\Sigma^*}$. The encoding exploits the fact that a string is a concatenation of characters *followed by the empty string* ε (which is generally omitted). For that, the encoding uses $|\Sigma| + 1$ abstractions, the extra one (x_ε in the definition below) being used to represent ε. The encoding is defined by induction on the structure of s as follows:

$$\begin{aligned} \overline{\varepsilon}^{\Sigma^*} &:= \lambda x_1 . \dots \lambda x_n . \lambda x_\varepsilon . x_\varepsilon \ , \\ \overline{a_i r}^{\Sigma^*} &:= \lambda x_1 . \dots \lambda x_n . \lambda x_\varepsilon . x_i \overline{r}^{\Sigma^*} . \end{aligned}$$

Note that the representation depends on the cardinality of Σ. As before, however, the alphabet is a fixed parameter, and so such a dependency is irrelevant. As an example, the encoding of the string aba with respect to the alphabet $\{a, b\}$ ordered as $a < b$ is

$$\overline{aba}^{\{a,b\}} \quad = \quad \lambda x_a . \lambda x_b . \lambda x_\varepsilon . x_a (\lambda x_a . \lambda x_b . \lambda x_\varepsilon . x_b (\lambda x_a . \lambda x_b . \lambda x_\varepsilon . x_a (\lambda x_a . \lambda x_b . \lambda x_\varepsilon . x_\varepsilon)))$$

Linear Space Representation Overhead. As announced in Section 5, we now explain how to remove the code-constructor gap, that is, how to make the size of variable pointers irrelevant for space (for the Scott encoding of strings). As already mentioned this is not mandatory for our result (it would be mandatory in a log-insensitive approach to linear space), but it is interesting to see that it is possible.

Note that in $\overline{s}^{\Sigma^*}$ every variable occurrence is bound inside the list of binders immediately preceding the occurrence. If de Bruijn indices are used to represent λ-terms, one needs only indices—that is, variable pointers—between 1 and $|\Sigma| + 1$, that is, of *constant size*. Note that, similarly, if variables are represented with textual names, again having only $|\Sigma| + 1$ distinct names is enough *if* one permits that different sequences of abstractions re-use the same names, that is, if one accepts Barendregt's convention to be broken. Remarkably, a notable folklore property of the (Space) KAM is that its implementation theorem does not need Barendregt's convention to hold. Therefore, de bruijn indices are not the only possible approach that removes the code-constructor gap on the Scott encoding of strings.

In their result about reasonable space, Forster et al. [FKR20] also rely on the Scott encoding of strings and they represent λ-terms using de Bruijn indices. Since they study (super-)linear space in a log-insensitive way, their choice of de Bruijn indices is crucial for their result to hold, even if they do not stress it.

Recursion and Fix-Points. The encoding of TMs crucially relies on the use of a fix-point operator to implement recursion. Precisely, fix-points are used to model the transition function, making a copy of the (sub-term encoding the) transition table at each step. It is the only point of the encoding where duplication occurs, and it is thus where the expressive power is encapsulated. The rest of the encoding is affine—note that the representation of strings is affine.

Fix-Points and Toy Scrolling Algorithms. To understand the delicate interplay between the space of the KAM and fix-points, we analyze it via simple toy algorithms on strings. The first, simplest one is the *consuming scrolling algorithm*: going through an input string s doing nothing and accepting when arriving at the end of the string, without having to preserve the string itself—the aim is just to see the space used for scrolling a string. The toy algorithm is a very rough approximations of the moving of TMs over a tape, which is the most delicate aspect of the space reasonable simulation of TMs in the λ-calculus that we shall develop. It is used to illustrate the key aspects of the problems that arise and of their solutions, without having to deal with all the details of the encoding of TMs at once. On TMs, scrolling a string obviously runs in constant space, and on log-sensitive TMs the consuming aspect cannot be modeled—we shall consider non-consuming scrolling later in this section.

We encode the algorithm as a λ-term over Scott strings, where a fix-point combinator is used to iterate over the (term t_s encoding) the input string s. Since the input string s is consumed in the process, the normal form would be the encoding of the accepting state q_1 of the TM, which for simplicity here is simply given by the identity combinator I.

We use Turing's fix-point combinator and the boolean alphabet $\mathbb{B} := \{0, 1\}$. Let $\mathsf{fix} := \theta\theta$, where $\theta := \lambda x.\lambda y.y(xxy)$. Given a term u, $\mathsf{fix}\, u$ is a fix-point of u.

$$\begin{array}{rcccccl} \mathsf{fix}\, u & = & (\lambda x.\lambda y.y(xxy))\theta u & & & & \\ & \to_\beta & (\lambda y.y(\theta\theta y))u & \to_\beta & u(\theta\theta u) & = & u(\mathsf{fix}\, u) \end{array}$$

Algorithms moving over binary Scott strings always follow the same structure. They are given by the fix-point iteration of a term that does pattern matching on the leftmost character of the string and for each of the possible outcomes (in our case, the first character is 0, 1, or the empty string ε) does the corresponding action. The general term is $\mathsf{fix}\,(\lambda f.\lambda z.zA_0A_1A_\varepsilon)$, where f is the variable for the recursive call and A_0, A_1, and A_ε represent the three actions, which in our case are simply given by $A_0 = A_1 = f$ and $A_\varepsilon = \mathsf{I}$, using the identity I as encoding of the accepting state. Formally, we have the following Proposition, the proof of which is in Appendix A.1.

Proposition 9.1. *Let* $s \in \mathbb{B}^*$ *and* $\mathtt{toy} := \mathsf{fix}\,(\lambda f.\lambda z.zff\mathsf{I})$.

(1) $\mathtt{toy}\,\overline{s}^{\mathbb{B}} \to_{wh}^{\Theta(|s|)} \mathsf{I}$.
(2) *The Naive KAM evaluates* $\mathtt{toy}\,\overline{s}^{\mathbb{B}}$ *in space* $\Omega(2^{|s|})$.
(3) *The Space KAM evaluates* $\mathtt{toy}\,\overline{s}^{\mathbb{B}}$ *in space* $\Theta(\log|s|)$.

We can see that the Naive KAM is desperately inefficient for space, while the Space KAM works within reasonable space bounds. It turns out, however, that the Space KAM is still not enough in order to obtain a space reasonable simulation of TMs. The problem now concerns the standard encoding of TMs and its managing of the tapes, rather than the use of space by the abstract machine itself. The issues can be explained using further toy algorithms.

String-Preserving Scrolling. Consider the same scrolling algorithm as above, except that now the input string s is *not* consumed by the moving over s, that is, it has to be given back as output of the λ-term implementing the algorithm. This variant is a step forward towards approximating what happens to the tapes of TMs *during* the computation: the TM moves over the tapes *without consuming them*, it is only at the end of the computation that

the TM can be seen as discarding the tapes. There are two ways of implementing the new algorithm:

(1) *Local copy*: moving over the string s while accumulating in a new accumulator string r the characters that have already been visited, returning r.
(2) *Global copy*: making a copy r of the string s, and then moving over s in a consuming way, returning r.

Local Copy. The local approach is the one underlying the reference encoding of TMs. In particular, it is almost affine, as duplication is isolated in the fix-point. The λ-term `loCpy` realizing it uses the same fix-point schema as before, but with different, more involved action terms A_0, A_1, and A_ε. We provide the following Proposition (and the next one) without proof, since `loCpy` is just a fragment of the encoding of TMs, for which we detail the execution by the Space KAM in Appendix B.

Proposition 9.2. *Let $s \in \mathbb{B}^*$.*

(1) $\texttt{loCpy}\,\overline{s}^{\mathbb{B}} \to_{wh}^{\Theta(|s|)} \overline{s}^{\mathbb{B}}$.
(2) *The Space KAM evaluates* $\texttt{loCpy}\,\overline{s}^{\mathbb{B}}$ *in space* $\mathcal{O}(|s|\log|s|)$.

The $\mathcal{O}(|s|\log|s|)$ bound in point 2 is problematic for the space reasonable modeling in the λ-calculus of both the input and the work tapes, for different reasons.

Work Tape and Left Addresses. For a space-reasonable managing of the work tape, a scrolling algorithm should rather work in space $\mathcal{O}(|s|)$. This improvement can be obtained by keeping the same algorithm and refining the complexity analysis. In Prop. 9.2.2, the cost comes from the use of $\mathcal{O}(|s|)$ sub-term pointers to the code $\texttt{loCpy}\,\overline{s}^{\mathbb{B}}$ used by the Space KAM. These pointers have size $\mathcal{O}(\log|s|)$ because $|\texttt{loCpy}\,\overline{s}^{\mathbb{B}}| = \mathcal{O}(|s|)$, that is, the size of `loCpy` is independent of $|s|$ and thus constant. A close inspection of the Space KAM run in Prop. 9.2.2 shows that, of the used $\mathcal{O}(|s|)$ pointers, only $\mathcal{O}(1)$ of them actually point to $\overline{s}^{\mathbb{B}}$, while all the others (that is, an $\mathcal{O}(|s|)$ amount) point to `loCpy`. Since `loCpy` is of size independent from $|s|$, if one admits separate address spaces for `loCpy` and $\overline{s}^{\mathbb{B}}$, as it is done using left addresses, then the pointers to `loCpy` have size $\mathcal{O}(1)$. And indeed the size of the left addresses that the Space KAM uses for sub-terms pointers give addresses to the sub-terms in `loCpy` is independent from $\overline{s}^{\mathbb{B}}$. Therefore, one obtains that the space cost is given by

$$\underbrace{\mathcal{O}(|s|)\cdot\mathcal{O}(1)}_{\text{pointers to } \texttt{loCpy}} \quad + \quad \underbrace{\mathcal{O}(1)\cdot\mathcal{O}(\log|s|)}_{\text{pointers to } \overline{s}^{\mathbb{B}}} = \mathcal{O}(|s|).$$

Proposition 9.3 (Linear Space Local-Copy Scrolling)**.** *Let $s \in \mathbb{B}^*$. The Space KAM evaluates* $\texttt{loCpy}\,\overline{s}^{\mathbb{B}}$ *in space* $\mathcal{O}(|s|)$.

Input Tape and Global Copy. For the input tape, a linear space bound for scrolling is unreasonable, if one aims at preserving logarithmic space complexity. For meeting the required $\mathcal{O}(\log|s|)$ bound, we need a more radical solution, which shall be possible because the tape is read-only (and thus the solution does not directly apply to the work tape).

The first step is the straightforward modification of the consuming scrolling algorithm into a global-copy string-preserving algorithm: it is enough to capture the input at the beginning with an extra abstraction λx and to give it back at the end with the action A_ε, that is, having $A_\varepsilon := x$. Namely, let $\texttt{glCpy} := \lambda x.(\mathsf{fix}\,(\lambda f.\lambda s'.s' f f x)x)$. Clearly, this

approach breaks the almost affinity of the encoding, as copying is no longer encapsulated only in the fix-point. Formally, we have the following Proposition, the proof of which is in Appendix A.2.

Proposition 9.4. *Let* $s \in \mathbb{B}^*$.
(1) $\texttt{glCpy}\,\overline{s}^{\mathbb{B}} \to_{wh}^{\Theta(|s|)} \overline{s}^{\mathbb{B}}$.
(2) *The Space KAM evaluates* $\texttt{glCpy}\,\overline{s}^{\mathbb{B}}$ *in space* $\Theta(\log|s|)$.

Interestingly, the space cost stays logarithmic, because the global copy of the input in point 1 (in fact there actually is a copy for every iteration of the fix-point) is not performed by the Space KAM, which instead only copies a *pointer* to it. The second step is refining this scheme as to implement a read-only tape, rather than just scrolling the tape. A slight digression is in order.

Intrinsic and Mathematical Tape Representations. A TM tape is a string plus a distinguished position, representing the head. There are two tape representations, dubbed *intrinsic* and *mathematical* by van Emde Boas in [vEB12].

- The *intrinsic* one represents both the string s and the current position of the head as the triple $s = s_l \cdot h \cdot s_r$, where s_l and s_r are the prefix and suffix of s surrounding the character h read by the head. This is the representation underlying the local-copy scrolling algorithm as well as the reference encoding of TMs. In this approach, reading from the tape costs $\mathcal{O}(1)$ time but the reading mechanism comes with a $\mathcal{O}(|s|)$ space overhead, as showcased by Prop. 9.3.
- The *mathematical* representation, instead, is simply given by the index $n \in \mathbb{N}$ of the head position, that is, the triple $s_l \cdot h \cdot s_r$ is replaced by the pair $(s, |s_l| + 1)$. The index $|s_l| + 1$ has the role of a pointer, of logarithmic size when represented in binary.

Mathematical Input and Global Copy. Given a *mathematical* read-only tape (s, n), one can use the global-copy scrolling scheme for a simulation in the λ-calculus in space $\mathcal{O}(\log|s|)$. The idea is to represent n as a binary string $\lfloor n \rfloor$. Since $n \leq |s|$, we have $|\lfloor n \rfloor| \leq \log|s|$. Moreover, it is possible to pass from $\lfloor n \rfloor$ to $\lfloor n+1 \rfloor$ or $\lfloor n-1 \rfloor$—which is needed to move the position of the head—in $\mathcal{O}(\log|s|)$ space. Finally, reading from the tape, that is, given a tape (s, n), returning (s, n) plus the n-th character s_n of s, is doable in space $\mathcal{O}(\log|s|)$ via the following composite operation:

- making a global copy of the tape (returned at the end),
- scrolling the current copy of n positions,
- extracting the head s_n of the obtained suffix, and
- discarding the tail.

Two remarks. First, this approach works because the tape is read-only, so that one can keep making global copies of the same immutable tape, and only changing the index of the head. Second, there is a (reasonable) time slowdown, because at each read the simulation has to scroll sequentially the input tape to get to the n-th character. Such a scrolling has time cost $\mathcal{O}(n \log n)$ because it is done by moving of (the encoding of) a cell at a time, and decrementing of one the index, until the index is 0. Therefore, accessing the right cell requires to decrement the index n times, each time requiring time $\log n$, because the index is represented in binary. Therefore, the time cost of a read operation is $\mathcal{O}(|s| \log|s|)$.

10. The Space KAM is Reasonable for Space

We are ready for our main result. It is based on a new variant over Dal Lago and Accattoli encoding of TMs into the λ-calculus [DLA17] which is defined in a separate document [ADLV23] (including also the encoding in the λ-calculus of the binary arithmetic needed for the mathematical representation of the input tape), to spare the tedious details of the encoding to the reader. The key points are:

- *Refined TMs*: the notion of TM we work with is log-sensitive TMs with mathematical input tape and intrinsic work tape (the formal definition of TMs is in [ADLV23]).
- *CPS and indifference*: following [DLA17], the encoding is in *continuation-passing style*, and carefully designed (by adding some η-expansions) as to fall into the *deterministic λ-calculus* $\Lambda_{\mathtt{det}}$, a particularly simple fragment of the λ-calculus where the right sub-terms of applications can only be variables or abstractions and where, consequently, call-by-name and call-by-value collapse on the same evaluation strategy $\to_{det}$. We shall exploit this *indifference* property in Section 12.
- *Duplication*: duplication is isolated in the unfolding of fix-points and in the managing of the input tape, all other operations are affine.

Theorem 10.1 (TMs are simulated by the Space KAM in reasonable space)**.** *There is an encoding $\bar{\cdot}$ of log-sensitive TMs into $\Lambda_{\mathtt{det}}$ such that if the run ρ of the TM M on input $i \in \mathbb{B}^*$:*

(1) Termination*: ends in q_b with $b \in \mathbb{B}$, then there is a complete sequence $\sigma : \overline{M}\,\overline{i} \to_{det}^{n} \overline{q_b}$ where $n = \Theta((T_{TM}(\rho)+1)\cdot|i|\cdot\log|i|)$.*
(2) Divergence*: diverges, then $\overline{M}\,\overline{i}$ is $\to_{det}$-divergent.*
(3) Space KAM*: the space used by the Space KAM to simulate the evaluation of point 1 is $\mathcal{O}(S_{TM}(\rho)+\log|i|)$.*

The previous theorem provides the subtle and important half of the space reasonability result. The first two points, proved in [ADLV23], establish the qualitative part of the simulation in the λ-calculus, together with the time bound (with respect to the number of β steps). They are connected to the Space KAM by the fact that the Space KAM implements closed call-by-name (that coincides with $\to_{det}$ in $\Lambda_{\mathtt{det}}$), via an omitted minor variant of Theorem 7.1. The third point is the important one, as it provides the space-reasonable simulation of the λ-calculus by the Space KAM. It is proved by directly executing on the Space KAM the λ-terms which are the image of the encoding of TMs. The proof is a tedious analysis of machine executions, and it is thus developed in Appendix B. The main ingredient is an invariant stating that, during the execution of the encoding on the Space KAM, configurations of the encoded TM are represented by Space KAM configurations of the same size.

The other half of the reasonability result amounts to showing that the Space KAM can be simulated on TMs within the space costs claimed in Section 8. The idea is that it can clearly be simulated reasonably by a multi-tape TM using one work tape for the active term (as a pointer to the fixed initial code), one for the environment, one for the stack, plus one for auxiliary pointers manipulations. Note that to encode the λ-calculus we use a notion of TM which is different from the one that we encoded in the λ-calculus, as there is no binary output, rather there is an output tape which, for the simulation, at the end of the execution is filled in with the content of the tape for the active term and the tape for the environment, which provide a shared representation of the result λ-term.

Proposition 10.2 (Space KAM is simulated by TMs in reasonable space)**.** *Let t be a λ-term. Every Space KAM run $\rho : \mathsf{init}(t) \to_{\mathrm{SpKAM}}{}^{*} q$ is implemented on TMs in space $\mathcal{O}(|\rho|_{\mathsf{sp}})$.*

Logarithmic Space Constructor Equality. The definition of reasonable space cost model (Def. 5.1) also requires that final configurations of the TM used in the simulation can be inspected in logarithmic space. For that, we provide a pseudo-code algorithm, presented as an inductive definition, that, given the initial term t_0, a closure $c_0 = (t_{\mathsf{fin}}, e_{\mathsf{fin}})$ (meant to be the closure of the final state of the Space KAM run), and a tree address a_0 (with respect to the decoded final term $c{\downarrow}$), returns the constructor of the term $c_0{\downarrow}$ at a_0. The algorithm navigates through c_0 using a notion of current closure $c = (u, E)$ which is either c_0 or a closure somewhere in e_{fin}. The navigation process is realized via three pointers:

- *Term pointer*: a pointer inside u, which we represent abstractly as a decomposition of u into a context C and a sub-term r, thus using a triple (r, C, e) to represent the closure with pointer $(C\langle r\rangle, e)$. Since u is a sub-term of the initial code, the term pointer actually moves over the initial code t_0.
- *Address pointer*: a pointer inside a_0, which, similarly to the term pointer, is represented as a pair of addresses (a, a') such that $a \cdot a' = a_0$.
- *Closure pointer*: a pointer inside c_0 to the current closure. For the sake of simplicity, we do not represent the closure pointer explicitly (it would require to introduce closure contexts, and the technicality is not worth it).

Given the initial closure $c_0 = (t_{\mathsf{fin}}, e_{\mathsf{fin}})$ and the initial address a_0, the algorithm is invoked as $(t_{\mathsf{fin}}, \langle\cdot\rangle, E_{\mathsf{fin}})|_{(\varepsilon, a_0)}$. It is defined as follows.

$$
\begin{array}{llll}
(tu, C, e)|_{(a, 0\cdot a')} & := & (t, C\langle\langle\cdot\rangle u\rangle, e)|_{(a\cdot 0, a')} & \qquad (tu, e)|_{(a,\varepsilon)} := @ \\
(tu, C, e)|_{(a, 1\cdot a')} & := & (u, C\langle t\langle\cdot\rangle\rangle, e)|_{(a\cdot 1, a')} & \qquad (\lambda x.t, C, e)|_{(a,\varepsilon)} := \lambda \\
(\lambda x.t, C, e)|_{(a, b\cdot a')} & := & (t, C\langle\lambda x.\langle\cdot\rangle\rangle, e)|_{(a\cdot b, a')} &
\end{array}
$$

$$
(x, C, e)|_{(a,a')} := \begin{cases} \mathsf{dB}(x) & \text{if } x \notin \mathsf{dom}(e) \text{ and } a' = \varepsilon, \\ (t, \langle\cdot\rangle, e')|_{(a,a')} & \text{if } x \in \mathsf{dom}(e) \text{ and } e(x) = (t, e') \\ \bot & \text{if } x \notin \mathsf{dom}(e) \text{ and } a' \neq \varepsilon \end{cases}
$$

The algorithm moves the pointer inside a sub-term according to the address a'. The key case is the one of a variable x, for which there are three possible outcomes:

- *Return*: if the address a' is empty, the de Bruijn index of the variable is returned (the initial code is closed by hypothesis). Note that terms are not necessarily represented with de Bruin indices: it is a convenient representation, but also with other representations, one can usually compute the de Bruijn index of a variable occurrence in logarithmic space;
- *Jump*: if the address a' is non-empty and x is bound by the environment, that is, $x \in \mathsf{dom}(e)$, then the algorithm moves to navigate the closure (t, e') associated to x by e. Here the implicit closure pointer changes, and it is the only point of the algorithm where it changes;
- *Error*: if the address a' is non-empty and x is not bound by the environment then it means that a_0 is an address that it is not compatible with the structure of the result term $c{\downarrow}$ of the computation. Thus the algorithm returns the undefined symbol $\bot$.

Note that the algorithm never needs to backtrack. The space complexity of the algorithm is easily verified to be $\mathcal{O}(\log|a| + \log|t| + \log|c|)$, because essentially one only needs to

manipulate the term, address, and closure pointers (plus possibly a constant number of auxiliary pointers to implement the algorithm). Therefore, we obtained the following result.

Proposition 10.3 (Logarithmic space constructor equality)**.** *Let t be a closed λ-term, $\rho : \mathsf{init}(t) \to_{\mathrm{SpKAM}}{}^{*} q$ be a Space KAM run, and a be a tree address. Then computing $(q\downarrow)|_a$ has space complexity $\mathcal{O}(\log|a| + \log|q| + \log|t|)$.*

Then, from the two simulations (Theorem 10.1 and Proposition 10.2) and the logarithmic space constructor equality test (Prop. 10.3), our main result follows.

Theorem 10.4 (The Space KAM is reasonable for space)**.** *Closed CbN evaluation $\to_{wh}$ and the space of the Space KAM provide a reasonable space cost model for the λ-calculus.*

10.1. **The Space KAM is not Reasonable for Abstract Time.** We complete our study of the Space KAM by analyzing its time behavior. For abstract time (in our case, the number of Closed CbN β steps), the Space KAM is unreasonable, because simulating Closed CbN at times requires exponential overhead. The number of transitions of the Space KAM is reasonable, while it is the cost of single transitions, thus of the manipulation of data structures, that can explode. The failure stems from the lack of data sharing, which on the other hand we showed being mandatory for space reasonability. Essentially, there are size exploding families such that their Space KAM run produces environments of size exponential in the number of β steps/transitions.

In order to prove this result we need to define the family of λ-terms $\{t_n\}_{n\in\mathbb{N}}$ and prove two auxiliary lemmas. We first define the following data structures:

$$\begin{array}{rclcrcl} e_0 & := & [x_0{\leftarrow}(\mathsf{I},\epsilon)] & & e_{n+1} & := & [x_{n+1}{\leftarrow}\pi_n]\cdot e_n \\ \pi_0 & := & (x_0x_0, e_0) & & \pi_{n+1} & := & (x_0..x_{n+1}, e_{n+1}) \end{array}$$

Note that the size of e_n is exponential in n.

Lemma 10.5. *For each $n \in \mathbb{N}$, $|e_n| \geq 2^n$.*

Proof. Since $e_{n+1} := [x_{n+1}{\leftarrow}\pi_n]\cdot e_n = [x_{n+1}{\leftarrow}(x_0..x_n, e_n)]\cdot e_n$, we have $|e_{n+1}| \geq 2|e_n|$ and thus $|e_n| \geq 2^n$. □

Now, we define the family of contexts $\{C_n\}_{n\in\mathbb{N}}$ as follows:

$$\begin{array}{rcl} C_0 & := & \lambda x_0.\langle\cdot\rangle(x_0x_0) \\ C_{n+1} & := & \lambda x_{n+1}.\langle\cdot\rangle(x_0\dots x_{n+1}) \end{array}$$

The execution of $C_n\langle t\rangle$ generates the stacks π_n and environments e_n defined above, provided that $x_0,\dots,x_n$ appear free in t.

Lemma 10.6. *For each λ-term t, if the variables $x_0,\dots,x_n$ appear free in t, then $(C_0\langle C_1\langle\dots C_n\langle t\rangle\dots\rangle\rangle\mathsf{I},\epsilon,\epsilon) \to^{\Theta(n)}_{\mathrm{SpaceKAM}} (t, e_n, \pi_n)$.*

Proof. We proceed by simply executing the Space KAM.

- Case $n = 0$.

Term	Env	Stack	
$C_0\langle t\rangle\mathsf{I}$	ϵ	ϵ	$\to_{\mathsf{sea}_{\neg\mathsf{v}}}$
$C_0\langle t\rangle := \lambda x_0.t(x_0x_0)$	ϵ	(I,ϵ)	$\to_{\beta_{\neg\mathsf{w}}}$
$t(x_0x_0)$	$[x_0{\leftarrow}(\mathsf{I},\epsilon)]$	ϵ	$\to_{\mathsf{sea}_{\neg\mathsf{v}}}$
t	$[x_0{\leftarrow}(\mathsf{I},\epsilon)]$	$(x_0x_0, [x_0{\leftarrow}(\mathsf{I},\epsilon)])$	

- Case $n \geq 1$. We observe that $C_0\langle C_1\langle \dots C_n\langle C_{n+1}\langle t\rangle\rangle \dots\rangle\rangle|$ can be rewritten to $C_0\langle C_1\langle \dots C_n\langle u\rangle \dots\rangle\rangle|$, where $u := C_{n+1}\langle t\rangle$. Of course $x_0, \dots, x_n$ appear free in u. We can thus immediately apply the *i.h.*

Term	Env	Stack	
$C_0\langle C_1\langle ..C_n\langle C_{n+1}\langle t\rangle\rangle ..\rangle\rangle\mathsf{I}$	ϵ	ϵ	$\to^{\Theta(n)}_{\text{SpaceKAM}}$ *(i.h.)*
$\underbrace{\lambda x_{n+1}.t(x_0..x_{n+1})}_{C_{n+1}\langle t\rangle}$	e_n	π_n	$\to_{\beta_{\neg\mathsf{w}}}$
$t(x_0..x_{n+1})$	$\underbrace{[x_{n+1}\leftarrow\pi_n]\cdot e_n}_{=:e_{n+1}}$	ϵ	$\to_{\mathsf{sea}_{\neg\mathsf{v}}}$
t	e_{n+1}	$(x_0..x_{n+1}, e_{n+1}) =: \pi_{n+1}$	

□

Finally, we prove that the Space KAM is not reasonable with respect to abstract time.

Proposition 10.7 (Space KAM abstract time overhead explosion)**.** *There is a family $\{t_n\}_{n\in\mathbb{N}}$ of closed λ-terms such that there is a complete evaluation $\rho_n : t_n \to^n_{wh} u_n$ simulated by Space KAM runs σ_n taking both space and time exponential in n, that is, $|\sigma_n|_{\mathsf{sp}} = |\sigma_n|_{\mathsf{tm}} = \Omega(2^n)$.*

Proof. Define t_n as $t_n := C_0\langle C_1\langle \cdots C_n\langle \lambda y.\mathsf{I}\rangle \cdots\rangle\rangle|$. Its Space KAM execution follows.

Term	Env	Stack	
$C_0\langle C_1\langle \cdots C_{n-1}\langle C_n\langle \lambda y.\mathsf{I}\rangle\rangle \cdots\rangle\rangle\mathsf{I}$	ϵ	ϵ	$\to^{\Theta(n)}_{\text{SpaceKAM}}$ (Lemma 10.6)
$\underbrace{\lambda x_n.(\lambda y.\mathsf{I})(x_0 \dots x_n)}_{=C_n\langle \lambda y.\mathsf{I}\rangle}$	e_{n-1}	π_{n-1}	$\to_{\beta_{\neg\mathsf{w}}}$
$(\lambda y.\mathsf{I})(x_0 \dots x_n)$	e_n	ϵ	$\to_{\mathsf{sea}_{\neg\mathsf{v}}}$
$\lambda y.\mathsf{I}$	ϵ	π_n	$\to_{\beta_{\mathsf{w}}}$
I	ϵ	ϵ	

The space consumed (and thus also the low-level time) is at least exponential in n because the size of e_n is exponential in n (Lemma 10.5). □

Since the Naive KAM is less efficient than the Space KAM, an analogous reasoning also shows that the Naive KAM is unreasonable with respect to abstract time.

Proposition 10.8 (Naive KAM abstract time overhead explosion)**.** *There is a family $\{t_n\}_{n\in\mathbb{N}}$ of closed λ-terms such that there is a complete evaluation $\rho_n : t_n \to^n_{wh} u_n$ simulated by Naive KAM runs σ_n taking both space and time at least exponential in n, that is, $|\sigma_n|_{\mathsf{sp}} = |\sigma_n|_{\mathsf{tm}} = \Omega(2^n)$.*

11. Time vs Space

Here we discuss how to obtain, or approximate, reasonability for both space and time.

Reasonable Low-Level Time. One way of recovering time reasonability is changing the time cost model from the number of β steps to the time taken by the Space KAM itself, which is a low-level notion of time. Such a time cost model is indeed reasonable. The key point is that the explosions of Prop. 10.7 never happen on λ-terms encoding TMs.

Theorem 11.1 (TMs are simulated by the Space KAM in reasonable low-level time)**.**

(1) *Every TM run ρ can be simulated by the Space KAM in time $\mathcal{O}(poly(|\rho|))$.*
(2) *Every Space KAM run $\rho : \mathsf{init}(t) \to_{\mathrm{SpKAM}}{}^{*} q$ can be implemented on TMs in time $\mathcal{O}(|\rho|_{\mathsf{tm}})$.*
(3) *Closed CbN and the time of the Space KAM provide a reasonable time cost model for the λ-calculus.*

Proof. The first point is the only one which is non-trivial. We have already proved that the Space KAM can simulate TMs runs ρ in a number of *transitions* which is polynomial in $|\rho|$. However, this does not necessarily means that the (low-level) time is also polynomial in ρ, see Proposition 10.7. About the execution of terms which are the image of the encoding of TMs into the λ-calculus, we can say however that the overhead stays polynomial. Indeed, the exponential blowup comes from the fact that environments are duplicated in an uncontrolled way. This does not happen in the execution of the encoding of TMs, where duplication is restrained to the fix-point operator and to the input components of the state. In other words, we duplicate only objects of fixed size, thus confirming the polynomial bound.

There is another, indirect, way of proving the same results. If the (low-level) time were exponential in $|\rho|$, then the space should be at least linear in $|\rho|$[4]. But we have proved that this is not the case since space is linearly related with the *space* consumption of ρ, and *not* with its length (which is the time consumption). □

The drawback of this solution is that one gives up the natural cost model for time. Moreover, the low-level time of the Space KAM can be very lax in comparison, as Prop. 10.7 shows.

KAM and Sharing of Environments. The Linked KAM of Fig. 2 (page 23) adopts sharing of environments, which is the most common way of turning the Outlined KAM into a machine reasonable for abstract time. The drawback is that such an implementation schema does not work with the space-oriented optimizations of the Collecting KAM, at least not smoothly. The culprit is eager garbage collection (there are no problems instead for unchaining), because, when an environment e is shared among many closures, knowing that an entry $[x \leftarrow c]$ of e is garbage for a closure (t, e) is not enough to remove $[x \leftarrow c]$ from e, since $[x \leftarrow c]$ might not be garbage for another closure using e.

Without eager garbage collection, the space consumption of the Linked KAM depends linearly on time, which is unreasonable for space. Therefore, sharing environments enables reasonability for abstract time at the expenses of space reasonability.

An interesting point is that the counter-example to time efficiency of the Space KAM in Prop. 10.7 is executed on the Linked KAM in exponentially less time *and* space than on the Space KAM, thanks to the sharing of environments. This fact stresses that space

[4]This is because space cannot be less than logarithmic in time, when space and time are locked and reasonable.

reasonable does not mean space efficient: the Space KAM is efficient on the encoding of TMs, but beyond the image of the encoding it can be terribly inefficient.

The Interleaving Technique. Forster et al. in [FKR20] show that, given one machine that is reasonable for abstract time but not for ink space and one machine that is reasonable for ink space but not for abstract time, it is possible to build a third machine that is reasonable for both ink space and abstract time—even if the two are explosive together—by interleaving the two machines in a smart way. Despite being presented on a specific case, their construction is quite general (in fact it is not even limited to the λ-calculus), and can be adapted to our case replacing ink space with the space of the Space KAM (the two starting machines being the Linked KAM and the Space KAM), under the assumption that the two machines share the same input, which is essential for logarithmic space[5]. The drawback of this solution is that it admits space exponential in time, as Prop. 10.7 shows.

Trading Time for Space. From a practical rather than theoretical point of view, there is a further *semi solution* that we now outline. The idea is to modify the Linked KAM as to share *closures* rather then environments, thus copying environments when crossing applications, but copying only their shallow structure, because for closures one would just copy pointers to them. Let us refer to this schema as to the *Closure KAM*. The Closure KAM is reasonable for abstract time (even if slightly slower than the Linked KAM) and—crucially—is compatible with the Collecting KAM. Compared with the Linked KAM, it thus has the advantage of disentangling space from time. It is not space reasonable, because of data pointers for sharing closures, that add an unreasonable pointer overhead which is logarithmic in the closure space of the Collecting KAM. But such an overhead is only *mildly* unreasonable.

The Closure KAM is probably the best compromise between reasonability and efficiency for the practice of implementing functional programs. We plan to study it in future work.

12. Call-by-Value and Other Strategies

How robust is our space cost model to changes of the evaluation strategy? The short answer is *very robust*.

Closed Call-by-Value. Our results smoothly adapt to weak call-by-value evaluation with closed terms, which we refer to as *Closed CbV* and define as follows. Values and right-to-left CbV evaluation contexts are given by:

$$\begin{array}{rrcl} \textsc{Values} & v & ::= & \lambda x.t \\ \textsc{Right-to-Left CbV Ctxs} & E & ::= & \langle\cdot\rangle \mid Ev \mid tE \end{array}$$

The (deterministic) reduction strategy $\to_{\mathsf{v}}$ is defined as the contextual closure of the β_{v} variant of the β rule

$$(\lambda x.t)v \mapsto_{\beta_{\mathsf{v}}} t\{x \leftarrow v\}$$

by right-to-left CbV evaluation contexts.

[5]The results in Forster et al. [FKR20] hold for decision problems (where the output is either *yes* or *no*), instead of computation problems (where the output can be any value) as in this paper, and are also given using fixed simulations. Their technique however is flexible and fairly independent from the notion of problem and also from the specific simulations, which are used as black boxes. The requirements for the technique are very lax, essentially that 1) the logarithm of time is linear in space (which is a fact true for most choices of cost models), and 2) the simulations should be runnable 'in rounds' (see [FKR20]).

Dumps	Closures	Environments
$d ::= \epsilon \mid d{\cdot}c{\diamond}\pi$	$c ::= (t, e)$	$e ::= \epsilon \mid [x{\leftarrow}c]{\cdot}e$
Stacks	**States**	
$\pi ::= \epsilon \mid c{\cdot}\pi$	$q ::= (d, t, e, \pi)$	

Dump	Term	Env	Stack		Dump	Term	Env	Stack	
d	tu	e	π	$\to_{\mathsf{sea}}$	$d{\cdot}(t, e\|_t){\diamond}\pi$	u	$e\|_u$	ϵ	
$d{\cdot}(u, e'){\diamond}\pi$	$\lambda x.t$	e	ϵ	$\to_{\mathsf{ret}}$	d	u	e'	$(\lambda x.t, e){\cdot}\pi$	
d	$\lambda x.t$	e	$c{\cdot}\pi$	$\to_{\beta_{\mathsf{w}}}$	d	t	e	π	if $x \notin \mathsf{fv}(t)$
d	$\lambda x.t$	e	$c{\cdot}\pi$	$\to_{\beta_{\neg\mathsf{w}}}$	d	t	$[x{\leftarrow}c]{\cdot}e$	π	if $x \in \mathsf{fv}(t)$
d	x	e	π	$\to_{\mathsf{sub}}$	d	u	e'	π	if $e(x) = (u, e')$

where $e|_t$ denotes the restriction of e to the free variables of t.

Figure 6: The Collecting LAM, the abstract layer of the Space LAM.

Our results smoothly adapt to such a setting, as we now explain. First, it is easy to adapt the Space KAM to Closed CbV. The LAM (Leroy Abstract Machine) is a right-to-left[6] CbV analogue of the KAM defined by Accattoli et al. in [ABM14] and modeled after Leroy's ZINC [Ler90] (whence the name). It uses a further data structure, the *dump*, storing the left sub-terms of applications yet to be evaluated. It is upgraded to the Collecting LAM in Fig. 6 by removing data pointers and adding eager GC. Lastly, it gives the Space LAM by adopting an abstract implementation analogous to that of the Space KAM. Unchaining comes for free in CbV, if one considers values to be only abstractions, see Accattoli and Sacerdoti Coen [ASC17].

The next step is realizing that, because of the mentioned *indifference property* of the deterministic λ-calculus $\Lambda_{\mathtt{det}}$ (containing the image of the encoding of TMs), the run of the Space LAM on a term $t \in \Lambda_{\mathtt{det}}$ is almost identical (technically, weakly bisimilar) to the one of the Space KAM on t.

Proposition 12.1. *The Space KAM and the Space LAM are weakly bisimilar when executed on $\Lambda_{\mathtt{det}}$-terms. Moreover, their space consumption is the same.*

Proof. The transitions of the Space KAM not dealing with applications are identical to the corresponding ones of the Space LAM (if one ignores the dump, that remains untouched). For the two transitions of the Space KAM dealing with applications, we show that, when the argument is a variable or an abstraction (as in $\Lambda_{\mathtt{det}}$), the Space LAM behaves as the Space KAM. If the active term is tx, indeed, the $\to_{\mathsf{sea_v}}$ transition of the Space KAM is simulated on the Space LAM by (with $e(x) = (\lambda y.u, e')$):

$$\begin{array}{lll} (\epsilon, tx, e, \pi) & \to_{\text{SpLAM}} & ((t, e|_t){\diamond}\pi, x, e|_x, \epsilon) \\ & \to_{\text{SpLAM}} & ((t, e|_t){\diamond}\pi, \lambda y.u, e', \epsilon) \\ & \to_{\text{SpLAM}} & (\epsilon, t, e|_t, (\lambda y.u, e'){\cdot}\pi) \\ & = & (\epsilon, t, e|_t, e(x){\cdot}\pi) \end{array}$$

[6]The argument presented here smoothly adapts to the left-to-right order.

	Abstract Time Reasonable (polyn. in # of β)	**Low-Level Time Reasonable** (actual implementation cost)	**Low-Level Space Reasonable**
Naive KAM	No, Proposition 10.7	No, Proposition 9.1.2	No, Proposition 9.1.2
Space KAM	No, Proposition 10.7	Yes, Theorem 11.1	Yes, Theorem 10.4
Linked KAM	Yes, see Section 7	Yes, see Section 7	No, see Section 7
Space LAM (CbV)	No, via Proposition 12.1 and Proposition 10.7	Yes, via Proposition 12.1 and Theorem. 11.1	Yes, Theorem 12.2

Figure 7: Summary of the results of the paper.

If the active term instead is $t(\lambda x.u)$, the $\rightarrow_{\mathsf{sea}_{\neg \mathsf{v}}}$ transition of the Space KAM is simulated on the Space LAM by:

$$\begin{array}{lll}(\epsilon, t(\lambda x.u), e, \pi) & \rightarrow_{\text{SpLAM}} & ((t, e|_t)\diamond\pi, \lambda x.u, e|_{\lambda x.u}, \epsilon) \\ & \rightarrow_{\text{SpLAM}} & (\epsilon, t, e|_t, (\lambda x.u, e|_{\lambda x.u}) \cdot \pi)\end{array}$$

In particular, these macro steps show that to evaluate TMs there is no need of the dump. Now, by defining a relation $\mathcal{R}$ between states of the Space KAM and the Space LAM as

$$q_K \,\mathcal{R}\, q_L \qquad \text{iff} \qquad q_K = (t, e, \pi) \text{ and } q_L = (\epsilon, t, e, \pi)$$

the previous reasoning shows that $\mathcal{R}$ is a weak bisimulation preserving time and space complexity (modulo a constant overhead). □

Since the simulation of the Space LAM on TMs is as smooth as for the Space KAM, we have the following result.

Theorem 12.2 (The Space LAM is reasonable for space)**.** *Closed CbV evaluation and the space of the Space LAM provide a reasonable space cost model for the λ-calculus.*

Open and Strong Evaluation. Extending CbN/CbV evaluation to deal with open terms or even under abstractions, which is notoriously very delicate in the study of reasonable time, is instead straightforward for space. This is because these extensions play no role in the simulation of TMs, which is the delicate direction for space. Given the absence of difficulties, we refrain from introducing variants of the Space KAM/LAM for open and strong evaluation.

Call-by-Need. The only major scheme for which our technique breaks is call-by-need (CbNeed) evaluation. To our knowledge, implementations of CbNeed inevitably rely on a heap and on data pointers similar to those of the Linked KAM, to realize the memoization mechanism at the heart of CbNeed. Therefore, they are space unreasonable. This is not really surprising: being a time optimization of CbN, CbNeed trades space for time, sacrificing space reasonability.

13. Conclusions

Via a fine study of abstract machines and of the encoding of Turing machines, we provide the first space cost model for the λ-calculus accounting for logarithmic space. We have reported our main results in Fig. 7.

Our cost model is given by an external device, the 700th abstract machine for the λ-calculus, so how canonical is it? The constraints for reasonable logarithmic space are *very* strict. It seems that there is no room for significant variations in the machine nor in the encoding of TMs. Moreover, our cost model for space has the same relationship to abstract time than ink space (that is, it is explosive, as shown by Prop. 10.7), and it smoothly adapts to other evaluation strategies, such as call-by-value. We then dare to say that our space cost model is fairly canonical.

We have also isolated an abstract notion of *closure space*, given by the maximum number of closures used by the Space KAM (without accounting for the size of sub-term pointers in the closures). In a companion paper [ADLV22a], we have given a machine independent characterization of closure space based on multi types, providing evidence of the naturality of our cost model.

Acknowledgment

This work has been inspired by an old talk by Kazushige Terui on the space efficiency of the KAM [Ter08]. The second author is partially supported by the ERC CoG "DIAPASoN" (GA 818616). The third author is partially supported by the ANR project "PPS" (ANR-19-CE48-0014) and by the European Union's Horizon 2020 research and innovation programme under the Marie Sklodowska-Curie grant agreement No 101034255.

References

[AB17] Beniamino Accattoli and Bruno Barras. Environments and the complexity of abstract machines. In Wim Vanhoof and Brigitte Pientka, editors, *Proceedings of the 19th International Symposium on Principles and Practice of Declarative Programming, Namur, Belgium, October 09 - 11, 2017*, pages 4–16. ACM, 2017. doi:10.1145/3131851.3131855.

[ABM14] Beniamino Accattoli, Pablo Barenbaum, and Damiano Mazza. Distilling abstract machines. In Johan Jeuring and Manuel M. T. Chakravarty, editors, *Proceedings of the 19th ACM SIGPLAN international conference on Functional progranitarmming, Gothenburg, Sweden, September 1-3, 2014*, pages 363–376. ACM, 2014. doi:10.1145/2628136.2628154.

[Acc17] Beniamino Accattoli. (In)Efficiency and Reasonable Cost Models. In *12th Workshop on Logical and Semantic Frameworks, with Applications, LSFA 2017, Brasília, Brazil, September 23-24, 2017*, volume 338 of *Electronic Notes in Theoretical Computer Science*, pages 23–43. Elsevier, 2017. doi:10.1016/j.entcs.2018.10.003.

[ACC21] Beniamino Accattoli, Andrea Condoluci, and Claudio Sacerdoti Coen. Strong call-by-value is reasonable, implosively. In *36th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2021, Rome, Italy, June 29 - July 2, 2021*, pages 1–14. IEEE, 2021. doi:10.1109/LICS52264.2021.9470630.

[Acc23] Beniamino Accattoli. Exponentials as substitutions and the cost of cut elimination in linear logic. *Log. Methods Comput. Sci.*, 19(4), 2023. doi:10.46298/LMCS-19(4:23)2023.

[ADL12] Beniamino Accattoli and Ugo Dal Lago. On the invariance of the unitary cost model for head reduction. In Ashish Tiwari, editor, *23rd International Conference on Rewriting Techniques and Applications (RTA'12) , RTA 2012, May 28 - June 2, 2012, Nagoya, Japan*, volume 15 of *LIPIcs*, pages 22–37. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2012. doi:10.4230/LIPIcs.RTA.2012.22.

[ADL16] Beniamino Accattoli and Ugo Dal Lago. (Leftmost-Outermost) Beta Reduction is Invariant, Indeed. *Logical Methods in Computer Science*, 12(1), 2016. doi:10.2168/LMCS-12(1:4)2016.

[ADLV20] Beniamino Accattoli, Ugo Dal Lago, and Gabriele Vanoni. The machinery of interaction. In *PPDP '20: 22nd International Symposium on Principles and Practice of Declarative Programming, Bologna, Italy, 9-10 September, 2020*, pages 4:1–4:15. ACM, 2020. doi:10.1145/3414080.3414108.

[ADLV21a] Beniamino Accattoli, Ugo Dal Lago, and Gabriele Vanoni. The (in)efficiency of interaction. *Proc. ACM Program. Lang.*, 5(POPL):1–33, 2021. doi:10.1145/3434332.

[ADLV21b] Beniamino Accattoli, Ugo Dal Lago, and Gabriele Vanoni. The space of interaction. In *2021 36th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, pages 1–13, 2021. doi:10.1109/LICS52264.2021.9470726.

[ADLV22a] Beniamino Accattoli, Ugo Dal Lago, and Gabriele Vanoni. Multi types and reasonable space. *Proc. ACM Program. Lang.*, 6(ICFP):799–825, 2022. doi:10.1145/3547650.

[ADLV22b] Beniamino Accattoli, Ugo Dal Lago, and Gabriele Vanoni. Reasonable space for the λ-calculus, logarithmically. In Christel Baier and Dana Fisman, editors, *LICS '22: 37th Annual ACM/IEEE Symposium on Logic in Computer Science, Haifa, Israel, August 2 - 5, 2022*, pages 47:1–47:13. ACM, 2022. doi:10.1145/3531130.3533362.

[ADLV23] Beniamino Accattoli, Ugo Dal Lago, and Gabriele Vanoni. A log-sensitive encoding of turing machines in the λ-calculus. 2023. doi:10.48550/arXiv.2301.12556.

[AJM00] Samson Abramsky, Radha Jagadeesan, and Pasquale Malacaria. Full abstraction for PCF. *Inf. Comput.*, 163(2):409–470, 2000. doi:10.1006/inco.2000.2930.

[ASC17] Beniamino Accattoli and Claudio Sacerdoti Coen. On the value of variables. *Inf. Comput.*, 255:224–242, 2017. doi:10.1016/j.ic.2017.01.003.

[BCD21] Malgorzata Biernacka, Witold Charatonik, and Tomasz Drab. A derived reasonable abstract machine for strong call by value. In Niccolò Veltri, Nick Benton, and Silvia Ghilezan, editors, *PPDP 2021: 23rd International Symposium on Principles and Practice of Declarative Programming, Tallinn, Estonia, September 6-8, 2021*, pages 6:1–6:14. ACM, 2021. doi:10.1145/3479394.3479401.

[BCD22] Malgorzata Biernacka, Witold Charatonik, and Tomasz Drab. A simple and efficient implementation of strong call by need by an abstract machine. *Proc. ACM Program. Lang.*, 6(ICFP):109–136, 2022. doi:10.1145/3549822.

[BG95] Guy E. Blelloch and John Greiner. 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*, pages 226–237. ACM, 1995. doi:10.1145/224164.224210.

[BG96] Guy E. Blelloch and John Greiner. A provable time and space efficient implementation of NESL. In Robert Harper and Richard L. Wexelblat, editors, *Proceedings of the 1996 ACM SIGPLAN International Conference on Functional Programming, ICFP 1996, Philadelphia, Pennsylvania, USA, May 24-26, 1996*, pages 213–225. ACM, 1996. doi:10.1145/232627.232650.

[CAC19] Andrea Condoluci, Beniamino Accattoli, and Claudio Sacerdoti Coen. Sharing equality is linear. In Ekaterina Komendantskaya, editor, *Proceedings of the 21st International Symposium on Principles and Practice of Programming Languages, PPDP 2019, Porto, Portugal, October 7-9, 2019*, pages 9:1–9:14. ACM, 2019. doi:10.1145/3354166.3354174.

[DF07] Rémi Douence and Pascal Fradet. The next 700 krivine machines. *High. Order Symb. Comput.*, 20(3):237–255, 2007. doi:10.1007/s10990-007-9016-y.

[DLA17] Ugo Dal Lago and Beniamino Accattoli. Encoding turing machines into the deterministic lambda-calculus. *CoRR*, abs/1711.10078, 2017. URL: http://arxiv.org/abs/1711.10078, arXiv:1711.10078.

[DLS10] Ugo Dal Lago and Ulrich Schöpp. Functional programming in sublinear space. In Andrew D. Gordon, editor, *19th European Symposium on Programming, ESOP 2010, Paphos, Cyprus, March 20-28, 2010, Proceedings.*, volume 6012 of *Lecture Notes in Computer Science*, pages 205–225. Springer, 2010. doi:10.1007/978-3-642-11957-6_12.

[DLS16] Ugo Dal Lago and Ulrich Schöpp. Computation by interaction for space-bounded functional programming. *Information and Computation*, 248:150–194, 2016. doi:10.1016/j.ic.2015.04.006.

[DR95] Vincent Danos and Laurent Regnier. Proof-nets and the hilbert space. In *Proceedings of the Workshop on Advances in Linear Logic*, pages 307–328, USA, 1995. Cambridge University Press. doi:10.1017/CBO9780511629150.016.

[FGSW07] Daniel P. Friedman, Abdulaziz Ghuloum, Jeremy G. Siek, and Onnie Lynn Winebarger. Improving the lazy krivine machine. *High. Order Symb. Comput.*, 20(3):271–293, 2007. doi:10.1007/s10990-007-9014-0.

[FKR20] Yannick Forster, Fabian Kunze, and Marc Roth. The weak call-by-value λ-calculus is reasonable for both time and space. *Proc. ACM Program. Lang.*, 4(POPL):27:1–27:23, 2020. doi:10.1145/3371095.

[FS08] Maribel Fernández and Nikolaos Siafakas. New developments in environment machines. In Aart Middeldorp, editor, *Proceedings of the 8th International Workshop on Reduction Strategies in Rewriting and Programming, WRS@RTA 2008, Hagenberg, Austria, July 14, 2008*, volume 237 of *Electronic Notes in Theoretical Computer Science*, pages 57–73. Elsevier, 2008. doi:10.1016/J.ENTCS.2009.03.035.

[Ghi07] Dan R. Ghica. Geometry of synthesis: a structured approach to VLSI design. In Martin Hofmann and Matthias Felleisen, editors, *Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2007, Nice, France, January 17-19, 2007*, pages 363–375. ACM, 2007. doi:10.1145/1190216.1190269.

[Gir89] Jean-Yves Girard. Geometry of Interaction 1: Interpretation of System F. In R. Ferro, C. Bonotto, S. Valentini, and A. Zanardo, editors, *Logic Colloquium '88*, volume 127 of *Studies in Logic and the Foundations of Mathematics*, pages 221–260. Elsevier, 1989. doi:10.1016/S0049-237X(08)70271-4.

[GMR12] Marco Gaboardi, Jean-Yves Marion, and Simona Ronchi Della Rocca. An implicit characterization of PSPACE. *ACM Trans. Comput. Log.*, 13(2):18:1–18:36, 2012. doi:10.1145/2159531.2159540.

[HU79] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, 1979.

[JHM11] Richard E. Jones, Antony L. Hosking, and J. Eliot B. Moss. *The Garbage Collection Handbook: The art of automatic memory management*. Chapman and Hall / CRC Applied Algorithms and Data Structures Series. CRC Press, 2011. URL: http://gchandbook.org/.

[Jon99] Neil D. Jones. LOGSPACE and PTIME characterized by programming languages. *Theor. Comput. Sci.*, 228(1-2):151–174, 1999. doi:10.1016/S0304-3975(98)00357-0.

[KBH12] Neelakantan R. Krishnaswami, Nick Benton, and Jan Hoffmann. Higher-order functional reactive programming in bounded space. In John Field and Michael Hicks, editors, *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012, Philadelphia, Pennsylvania, USA, January 22-28, 2012*, pages 45–58. ACM, 2012. doi:10.1145/2103656.2103665.

[Kri07] Jean-Louis Krivine. A Call-by-name Lambda-calculus Machine. *Higher Order Symbol. Comput.*, 20(3):199–207, 2007. doi:10.1007/s10990-007-9018-9.

[Ler90] Xavier Leroy. The ZINC experiment: an economical implementation of the ML language. Technical report 117, INRIA, 1990. URL: http://gallium.inria.fr/~xleroy/publi/ZINC.pdf.

[Mac95] Ian Mackie. The Geometry of Interaction Machine. In Ron K. Cytron and Peter Lee, editors, *Conference Record of POPL'95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Francisco, California, USA, January 23-25, 1995*, pages 198–208. ACM Press, 1995. doi:10.1145/199448.199483.

[Maz15] Damiano Mazza. Simple parsimonious types and logarithmic space. In Stephan Kreutzer, editor, *24th EACSL Annual Conference on Computer Science Logic, CSL 2015, September 7-10, 2015, Berlin, Germany*, volume 41 of *LIPIcs*, pages 24–40. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2015. doi:10.4230/LIPIcs.CSL.2015.24.

[PA19] Zoe Paraskevopoulou and Andrew W. Appel. Closure conversion is safe for space. *Proc. ACM Program. Lang.*, 3(ICFP):83:1–83:29, 2019. doi:10.1145/3341687.

[SBHG10] Daniel Spoonhower, Guy E. Blelloch, Robert Harper, and Phillip B. Gibbons. Space profiling for parallel functional programs. *J. Funct. Program.*, 20(5-6):417–461, 2010. doi:10.1017/S0956796810000146.

[Sch06] Ulrich Schöpp. Space-efficient computation by interaction. In Zoltán Ésik, editor, *Computer Science Logic, 20th International Workshop, CSL 2006, 15th Annual Conference of the EACSL,*

Szeged, Hungary, September 25-29, 2006, Proceedings, volume 4207 of *Lecture Notes in Computer Science*, pages 606–621. Springer, 2006. doi:10.1007/11874683_40.

[Sch07] Ulrich 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*, pages 411–420. IEEE Computer Society, 2007. doi:10.1109/LICS.2007.45.

[Ses97] Peter Sestoft. Deriving a lazy abstract machine. *J. Funct. Program.*, 7(3):231–264, 1997. URL: http://journals.cambridge.org/action/displayAbstract?aid=44087.

[SGM02] David Sands, Jörgen Gustavsson, and Andrew Moran. Lambda calculi and linear speedups. In Torben Æ. Mogensen, David A. Schmidt, and Ivan Hal Sudborough, editors, *The Essence of Computation, Complexity, Analysis, Transformation. Essays Dedicated to Neil D. Jones [on occasion of his 60th birthday]*, volume 2566 of *Lecture Notes in Computer Science*, pages 60–84. Springer, 2002. doi:10.1007/3-540-36377-7_4.

[SJ95] Patrick M. Sansom and Simon L. Peyton Jones. Time and space profiling for non-strict higher-order functional languages. In Ron K. Cytron and Peter Lee, editors, *Conference Record of POPL'95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Francisco, California, USA, January 23-25, 1995*, pages 355–366. ACM Press, 1995. doi:10.1145/199448.199531.

[SvEB88] Cees F. Slot and Peter van Emde Boas. The problem of space invariance for sequential machines. *Inf. Comput.*, 77(2):93–122, 1988. doi:10.1016/0890-5401(88)90052-1.

[Ter08] Kazushige Terui. On space efficiency of krivine's abstract machine and hyland-ong games. https://www.kurims.kyoto-u.ac.jp/~terui/space2.pdf, 2008. Accessed: 2022-05-31.

[vEB90] Peter van Emde Boas. Machine models and simulation. In *Handbook of Theoretical Computer Science, Volume A: Algorithms and Complexity (A)*, pages 1–66. MIT Press, 1990.

[vEB12] Peter van Emde Boas. Turing machines for dummies - why representations do matter. In Mária Bieliková, Gerhard Friedrich, Georg Gottlob, Stefan Katzenbeisser, and György Turán, editors, *SOFSEM 2012: Theory and Practice of Computer Science - 38th Conference on Current Trends in Theory and Practice of Computer Science, Špindlerův Mlýn, Czech Republic, January 21-27, 2012. Proceedings*, volume 7147 of *Lecture Notes in Computer Science*, pages 14–30. Springer, 2012. doi:10.1007/978-3-642-27660-6_2.

[Wan07] Mitchell Wand. On the correctness of the krivine machine. *High. Order Symb. Comput.*, 20(3):231–235, 2007. doi:10.1007/s10990-007-9019-8.

Appendix A. Proofs of Section 9

In the following we will often use the execution of the fixed point combinator $\mathsf{fix} := \theta\theta$, where $\theta := \lambda x.\lambda y.y(xxy)$. For this reason, we encapsulate its execution by the Space KAM in a lemma.

Lemma A.1. *For each term u, $(\theta, \epsilon, (\theta, \epsilon)\cdot(u, e)\cdot\pi) \to_{\mathrm{SpKAM}}^{\mathcal{O}(1)} (u, e, \mathsf{fix}^{\mathrm{K}}\cdot\pi)$ where $\mathsf{fix}^{\mathrm{K}} := (xxy, [y\leftarrow(u, e)]\cdot[x\leftarrow(\theta, \epsilon)])$ consuming space $\mathcal{O}(|e| + |\pi| + \log(|u|))$.*

Proof.

Term	Env	Stack	
$\theta := \lambda x.\lambda y.y(xxy)$	ϵ	$(\theta, \epsilon)\cdot(u, e)\cdot\pi$	$\to_{\beta_{\neg\mathsf{w}}}$
$\lambda y.y(xxy)$	$[x\leftarrow(\theta, \epsilon)]$	$(u, e)\cdot\pi$	$\to_{\beta_{\neg\mathsf{w}}}$
$y(xxy)$	$[y\leftarrow(u, e)]\cdot[x\leftarrow(\theta, \epsilon)]$	π	$\to_{\mathsf{sea}_{\neg\mathsf{v}}}$
y	$[y\leftarrow(u, e)]$	$\overbrace{(xxy, [y\leftarrow(u, e)]\cdot[x\leftarrow(\theta, \epsilon)])}^{\mathsf{fix}^{\mathrm{K}}}\cdot\pi$	$\to_{\mathsf{sub}}$
u	e	$\mathsf{fix}^{\mathrm{K}}\cdot\pi$	

□

A.1. **Proof of Proposition 9.1.** We prove the following proposition in a top-down style, i.e. required lemmata are below. This is done because otherwise lemmata statements would seem quite arbitrary. Nonetheless, we need a preliminary definition of some specific environments.

Definition A.2. Let $s := b_1\cdot\ldots\cdot b_n\cdot\varepsilon$ be a string of length $n \geq 0$. Then, for each $0 \leq i \leq n$ we can define e_i, e'_i, e''_i as follows:

$$e_0 := [x\leftarrow(\theta, \epsilon)]\cdot[y\leftarrow(\mathtt{toyaux}, \epsilon)] \qquad e_{i+1} := [x\leftarrow(x, e_i)]\cdot[y\leftarrow(y, e_i)]$$
$$e''_0 := \epsilon \qquad e''_{i+1} := [x_\varepsilon\leftarrow(\mathsf{I}, e'_i)]\cdot[x_1\leftarrow(f, e'_i)]\cdot[x_0\leftarrow(f, e'_i)]\cdot e''_i$$
$$e'_i := [z\leftarrow(\overline{b_{i+1}..b_n\cdot\varepsilon}, e''_i)]\cdot[f\leftarrow(xxy, e_i)]$$

One can easily notice that the sizes of e_i, e'_i, e''_i are exponential in i.

Proposition A.3. *Let $s \in \mathbb{B}^*$ and $\mathtt{toy} := \mathsf{fix}\,\mathtt{toyaux}$ where $\mathtt{toyaux} := \lambda f.\lambda z.zff\mathsf{I}$.*
(1) $\mathtt{toy}\,\overline{s}^{\mathbb{B}} \to_{wh}^{\Theta(|s|)} \mathsf{I}$.
(2) *The Naive KAM evaluates $\mathtt{toy}\,\overline{s}^{\mathbb{B}}$ in space $\Omega(2^{|s|})$.*
(3) *The Space KAM evaluates $\mathtt{toy}\,\overline{s}^{\mathbb{B}}$ in space $\Theta(\log|s|)$.*

Proof. Since $\mathbb{B}$ is the only alphabet that we are using, we remove all the superscripts.
(1) This point follows from the implementation theorem, applied to the sequence of point 3.
(2) We prove the statement executing $\mathtt{toy}\,\overline{s}$ with the Naive KAM.

Term	Env	Stack	
$\mathtt{toy}\,\overline{s}$	ϵ	ϵ	$\to_{\mathsf{sea}}$
$\mathtt{toy} := \mathsf{fix}\,\mathtt{toyaux}$	ϵ	$(\overline{s}, \epsilon)$	$\to_{\mathsf{sea}}$
$\mathsf{fix} := \theta\theta$	ϵ	$(\mathtt{toyaux}, \epsilon)\cdot(\overline{s}, \epsilon)$	$\to_{\mathsf{sea}}$
$\theta := \lambda x.\lambda y.y(xxy)$	ϵ	$(\theta, \epsilon)\cdot(\mathtt{toyaux}, \epsilon)\cdot(\overline{s}, \epsilon)$	$\to_\beta^2$
$y(xxy)$	$[x\leftarrow(\theta, \epsilon)]\cdot[y\leftarrow(\mathtt{toyaux}, \epsilon)] =: e_0$	$(\overline{s}, \epsilon)$	Lemma A.4
I	$e'_{\lvert s\rvert}$	ϵ	

The space bound is proved since $e'_{|s|}$ is exponential in $|s|$.

(3) We prove the statement executing $\mathtt{toy}\,\overline{s}$ with the Space KAM.

Term	Env	Stack	
$\mathtt{toy}\,\overline{s}$	ϵ	ϵ	$\to_{\mathsf{sea}_{\neg\mathsf{v}}}$
$\mathtt{toy} := \mathsf{fix}\,\mathtt{toyaux}$	ϵ	$(\overline{s},\epsilon)$	$\to_{\mathsf{sea}_{\neg\mathsf{v}}}$
$\mathsf{fix} := \theta\theta$	ϵ	$(\mathtt{toyaux},\epsilon)\cdot(\overline{s},\epsilon)$	$\to_{\mathsf{sea}_{\neg\mathsf{v}}}$
$\theta := \lambda x.\lambda y.y(xxy)$	ϵ	$(\theta,\epsilon)\cdot(\mathtt{toyaux},\epsilon)\cdot(\overline{s},\epsilon)$	$\to^2_{\beta_{\neg\mathsf{w}}}$
$y(xxy)$	$[x\leftarrow(\theta,\epsilon)]\cdot[y\leftarrow(\mathtt{toyaux},\epsilon)] =: e_0$	$(\overline{s},\epsilon)$	Lemma A.5
I	ϵ	ϵ	

The space bound is proved considering the bound in Lemma A.5. □

The second point in the statement of the previous proposition needs the following auxiliary lemma, proved by induction.

Lemma A.4. $(y(xxy), e_i, (\overline{s}, e''_i)) \to_{\mathrm{NaKAM}}{}^{\Omega(|s|)} (\mathsf{I}, e'_{i+|s|}, \epsilon)$.

Proof. By induction on the structure of s.

Term	Env	Stack	
$y(xxy)$	e_i	$(\overline{s}, e''_i)$	$\to_{\mathsf{sea}}$
y	e_i	$(xxy, e_i)\cdot(\overline{s}, e''_i)$	$\to^{i+1}_{\mathsf{sub}}$
$\mathtt{toyaux} := \lambda f.\lambda z.zff\mathsf{I}$	ϵ	$(xxy, e_i)\cdot(\overline{s}, e''_i)$	$\to^2_{\beta}$
$zff\mathsf{I}$	$[z\leftarrow(\overline{s}, e''_i)]\cdot[f\leftarrow(xxy, e_i)] =: e'_i$	ϵ	$\to^3_{\mathsf{sea}}$
z	$[z\leftarrow(\overline{s}, e''_i)]\cdot[f\leftarrow(xxy, e_i)]$	$(f, e'_i)\cdot(f, e'_i)\cdot(\mathsf{I}, e'_i)$	$\to_{\mathsf{sub}}$
$\overline{s}$	e''_i	$(f, e'_i)\cdot(f, e'_i)\cdot(\mathsf{I}, e'_i)$	

Case $s = \varepsilon$.

Term	Env	Stack	
$\overline{s} := \lambda x_0.\lambda x_1.\lambda x_\varepsilon.x_\varepsilon$	e''_i	$(f, e'_i)\cdot(f, e'_i)\cdot(\mathsf{I}, e'_i)$	$\to^3_{\beta}$
x_ε	$[x_\varepsilon\leftarrow(\mathsf{I}, e'_i)]\cdot[x_1\leftarrow(f, e'_i)]\cdot[x_0\leftarrow(f, e'_i)]\cdot e''_i$	ϵ	$\to_{\mathsf{sub}}$
I	e'_i	ϵ	

Case $s = b{\cdot}r$.

Term	Env	Stack	
$\overline{s} := \lambda x_0.\lambda x_1.\lambda x_\varepsilon.x_b\overline{r}$	e''_i	$(f, e'_i)\cdot(f, e'_i)\cdot(\mathsf{I}, e'_i)$	$\to^3_{\beta}$
$x_b\overline{r}$	$[x_\varepsilon\leftarrow(\mathsf{I}, e'_i)]\cdot[x_1\leftarrow(f, e'_i)]\cdot[x_0\leftarrow(f, e'_i)]\cdot e''_i =: e''_{i+1}$	ϵ	$\to_{\mathsf{sea}}$
x_b	$[x_\varepsilon\leftarrow(\mathsf{I}, e'_i)]\cdot[x_1\leftarrow(f, e'_i)]\cdot[x_0\leftarrow(f, e'_i)]\cdot e''_i$	$(\overline{r}, e''_{i+1})$	$\to_{\mathsf{sub}}$
f	$[z\leftarrow(\overline{s}, e''_i)]\cdot[f\leftarrow(xxy, e_i)]$	$(\overline{r}, e''_{i+1})$	$\to_{\mathsf{sub}}$
xxy	$[x\leftarrow(x, e_{i-1})]\cdot[y\leftarrow(y, e_{i-1})]$	$(\overline{r}, e''_{i+1})$	$\to^2_{\mathsf{sea}}$
x	$[x\leftarrow(x, e_{i-1})]\cdot[y\leftarrow(y, e_{i-1})]$	$(x, e_i)\cdot(y, e_i)\cdot(\overline{r}, e''_{i+1})$	$\to^{i+1}_{\mathsf{sub}}$
$\theta := \lambda x.\lambda y.y(xxy)$	ϵ	$(x, e_i)\cdot(y, e_i)\cdot(\overline{r}, e''_{i+1})$	$\to^2_{\beta}$
$y(xxy)$	$[x\leftarrow(x, e_i)]\cdot[y\leftarrow(y, e_i)] =: e_{i+1}$	$(\overline{r}, e''_{i+1})$	*i.h.*
I	$e'_{i+|s|}$	ϵ	

□

The third point in the statement of the proposition A.3 needs the following auxiliary lemma, proved by induction.

Lemma A.5. *The Space KAM executes the reduction* $(y(xxy), e_0, (\overline{s}, \epsilon)) \to_{\mathrm{SpKAM}}{}^{\Theta(|s|)} (\mathsf{I}, \epsilon, \epsilon)$ *consuming* $\mathcal{O}(\log(|s|))$ *space.*

Proof. By induction on the structure of s.

Term	Env	Stack	
$y(xxy)$	e_0	$(\overline{s}, \epsilon)$	$\to_{\mathsf{sea}_{\neg\mathsf{v}}}$
y	$[y{\leftarrow}(\mathtt{toyaux}, \epsilon)]$	$(xxy, e_0){\cdot}(\overline{s}, \epsilon)$	$\to_{\mathsf{sub}}$
$\mathtt{toyaux} := \lambda f.\lambda z.zff\mathsf{I}$	ϵ	$(xxy, e_0){\cdot}(\overline{s}, \epsilon)$	$\to^2_{\beta_{\neg\mathsf{w}}}$
$zff\mathsf{I}$	$[z{\leftarrow}(\overline{s}, \epsilon)]{\cdot}[f{\leftarrow}(xxy, e_0)]$	ϵ	$\to_{\mathsf{sea}_{\neg\mathsf{v}}}\to^2_{\mathsf{sea}_{\mathsf{v}}}$
z	$[z{\leftarrow}(\overline{s}, \epsilon)]$	$(xxy, e_0){\cdot}(xxy, e_0){\cdot}(\mathsf{I}, \epsilon)$	$\to_{\mathsf{sub}}$
$\overline{s}$	ϵ	$(xxy, e_0){\cdot}(xxy, e_0){\cdot}(\mathsf{I}, \epsilon)$	

Case $s = \varepsilon$.

Term	Env	Stack	
$\overline{s} := \lambda x_0.\lambda x_1.\lambda x_\varepsilon.x_\varepsilon$	ϵ	$(xxy, e_0){\cdot}(xxy, e_0){\cdot}(\mathsf{I}, \epsilon)$	$\to^2_{\beta_{\mathsf{w}}}\to_\beta$
x_ε	$[x_\varepsilon{\leftarrow}(\mathsf{I}, \epsilon)]$	ϵ	$\to_{\mathsf{sub}}$
I	ϵ	ϵ	

Case $s = b{\cdot}r$.

Term	Env	Stack	
$\overline{s} := \lambda x_0.\lambda x_1.\lambda x_\varepsilon.x_b\overline{r}$	ϵ	$(xxy, e_0){\cdot}(xxy, e_0){\cdot}(\mathsf{I}, \epsilon)$	$\to^2_\beta\to_{\beta_{\mathsf{w}}}$
$x_b\overline{r}$	$[x_b{\leftarrow}(xxy, e_0)]$	ϵ	$\to_{\mathsf{sea}}$
x_b	$[x_b{\leftarrow}(xxy, e_0)]$	$(\overline{r}, \epsilon)$	$\to_{\mathsf{sub}}$
xxy	e_0	$(\overline{r}, \epsilon)$	$\to_{\mathsf{sea}_{\mathsf{v}}}$
x	$[x{\leftarrow}(\theta, \epsilon)]$	$(\theta, \epsilon){\cdot}(\mathtt{toyaux}, \epsilon){\cdot}(\overline{r}, \epsilon)$	$\to_{\mathsf{sub}}$
$\theta := \lambda x.\lambda y.y(xxy)$	ϵ	$(\theta, \epsilon){\cdot}(\mathtt{toyaux}, \epsilon){\cdot}(\overline{r}, \epsilon)$	$\to_{\beta_{\neg\mathsf{w}}}$
$y(xxy)$	e_0	$(\overline{r}, \epsilon)$	*i.h.*
I	ϵ	ϵ	

The space bound is proved since there is a static bound, namely 8, on the number of closures stored during the execution, and the size of each closure is clearly bounded by $\log(|s|)$. □

A.2. **Proof of Proposition 9.4.** As before an auxiliary lemma is required to prove the proposition. We state and prove it below the main proposition.

Proposition A.6. *Let* $s \in \mathbb{B}^*$ *and* $\mathtt{glCpy} := \lambda z.(\mathsf{fix}\,(\lambda f.\lambda s'.s'ffz)z)$.

(1) $\mathtt{glCpy}\,\overline{s}^{\mathbb{B}} \to^{\Theta(|s|)}_{wh} \overline{s}^{\mathbb{B}}$.
(2) *The space used by the Space KAM to simulate the evaluation of the previous point is* $\Theta(\log|s|)$.

Proof. The first point is a consequence of the second one, since the Space KAM correctly implements Closed Call-by-Name. We prove the second point of the statement by directly executing the Space KAM. Since $\mathbb{B}$ is the only alphabet that we are using, we remove all the superscripts. Let us define $t := \lambda f.\lambda s'.s'ffz$.

Term	Environment	Stack	
$\mathtt{glCpy}\,\overline{s}$	ϵ	ϵ	$\to_{\mathsf{sea}_{\neg\mathsf{v}}}$
$\mathtt{glCpy} := \lambda z.(\mathsf{fix}\,tz)$	ϵ	$(\overline{s},\epsilon) =: s^{\mathrm{K}}$	$\to_{\beta_{\neg\mathsf{w}}}$
$\mathsf{fix}\,tz$	$[z{\leftarrow}s^{\mathrm{K}}]$	ϵ	$\to_{\mathsf{sea}_{\mathsf{v}}}$
$\mathsf{fix}\,t$	$[z{\leftarrow}s^{\mathrm{K}}]$	s^{K}	$\to_{\mathsf{sea}_{\neg\mathsf{v}}}$
$\mathsf{fix} := \theta\theta$	ϵ	$(t,[z{\leftarrow}s^{\mathrm{K}}])\cdot s^{\mathrm{K}}$	$\to_{\mathsf{sea}_{\neg\mathsf{v}}}$
$\theta := \lambda x.\lambda y.y(xxy)$	ϵ	$(\theta,\epsilon)\cdot(t,[z{\leftarrow}s^{\mathrm{K}}])\cdot s^{\mathrm{K}}$	$\to^{\Theta(\lvert s\rvert)}$ (Lemma A.7)
$\overline{s}$	ϵ	ϵ	

The space bound is immediate considering the space bound of lemma A.7, and the fact that during the execution only a fixed number of closures is stored. □

Lemma A.7. *Let $s \in \mathbb{B}^*$ and $t := \lambda f.\lambda s'.s'fffz$. Then $(\theta,\epsilon,(\theta,\epsilon)\cdot(t,[z{\leftarrow}(u,e)])\cdot s^{\mathrm{K}}) \to_{\mathrm{SpKAM}}{}^{\Theta(\lvert s\rvert)} (u,e,\epsilon)$ and the space used is $\mathcal{O}(\lvert e\rvert + \log\lvert s\rvert + \log\lvert u\rvert)$.*

Proof. We proceed by induction on the structure of s. The first steps are common to both the base case and the induction step. We define $\mathsf{fix}^{\mathrm{K}} := (xxy,[y{\leftarrow}(t,[z{\leftarrow}(u,e)])]\cdot[x{\leftarrow}(\theta,\epsilon)])$.

Term	Environment	Stack	
$\theta := \lambda x.\lambda y.y(xxy)$	ϵ	$(\theta,\epsilon)\cdot(t,[z{\leftarrow}(u,e)])\cdot s^{\mathrm{K}}$	$\to$ (Lemma A.1)
$t := \lambda f.\lambda s'.s'fffz$	$[z{\leftarrow}(u,e)]$	$\mathsf{fix}^{\mathrm{K}}\cdot s^{\mathrm{K}}$	$\to^2_{\beta_{\neg\mathsf{w}}}$
$s'fffz$	$[f{\leftarrow}\mathsf{fix}^{\mathrm{K}}]\cdot[s'{\leftarrow}s^{\mathrm{K}}]\cdot[z{\leftarrow}(u,e)]$	ϵ	$\to^3_{\mathsf{sea}_{\mathsf{v}}}$
s'	$[s'{\leftarrow}s^{\mathrm{K}}]$	$\mathsf{fix}^{\mathrm{K}}\cdot\mathsf{fix}^{\mathrm{K}}\cdot(u,e)$	$\to_{\mathsf{sub}}$
$\overline{s}$	ϵ	$\mathsf{fix}^{\mathrm{K}}\cdot\mathsf{fix}^{\mathrm{K}}\cdot(u,e)$	

Base case: $s = \varepsilon$.

Term	Environment	Stack	
$\overline{s} := \lambda x_0.\lambda x_1.\lambda x_\varepsilon.x_\varepsilon$	ϵ	$\mathsf{fix}^{\mathrm{K}}\cdot\mathsf{fix}^{\mathrm{K}}\cdot(u,e)$	$\to^2_{\beta_{\mathsf{w}}}\to_{\beta_{\neg\mathsf{w}}}\to_{\mathsf{sub}}$
u	e	ϵ	

Inductive case: $s : b{\cdot}r$ where $b \in \{0,1\}$.

Term	Environment	Stack	
$\overline{s} := \lambda x_0.\lambda x_1.\lambda x_\varepsilon.x_b\overline{r}$	ϵ	$\mathsf{fix}^{\mathrm{K}}\cdot\mathsf{fix}^{\mathrm{K}}\cdot(u,e)$	$\to^2_{\beta}\to_{\beta_{\mathsf{w}}}$
$x_b\overline{r}$	$[x_b{\leftarrow}\mathsf{fix}^{\mathrm{K}}]$	ϵ	$\to_{\mathsf{sea}}$
x_b	$[x_b{\leftarrow}\mathsf{fix}^{\mathrm{K}}]$	$(\overline{r},\epsilon) =: r^{\mathrm{K}}$	$\to_{\mathsf{sub}}$
xxy	$[y{\leftarrow}(t,[z{\leftarrow}(u,e)])]\cdot[x{\leftarrow}(\theta,\epsilon)]$	r^{K}	$\to^2_{\mathsf{sea}_{\mathsf{v}}}$
x	$[x{\leftarrow}(\theta,\epsilon)]$	$(\theta,\epsilon)\cdot(t,[z{\leftarrow}(u,e)])\cdot r^{\mathrm{K}}$	$\to_{\mathsf{sub}}$
θ	ϵ	$(\theta,\epsilon)\cdot(t,[z{\leftarrow}(u,e)])\cdot r^{\mathrm{K}}$	$\to^{\Theta(\lvert r\rvert)}$ *i.h.*
u	e	ϵ	

About space, it is immediate to see that all computations are constrained in space $\mathcal{O}(\lvert e\rvert + \log\lvert s\rvert + \log\lvert u\rvert)$, since at any point during the computation there is a *bounded* number of closures, independent from both s and u. □

Appendix B. Proofs of Section 10

This appendix is devoted to the proof of the main theorem of the paper, i.e. the space reasonable simulation of TMs into the λ-calculus (better, the Space KAM). It is a boring proof, where we simply execute the image of the encoding of TMs into the λ-calculus with the Space KAM.

First, we need to understand how a TM configuration is represented in the Space KAM, i.e. how it is mapped to environments and closures. This will be a sort of *invariant* of the execution.

Definition B.1. A configuration C of a TM is represented as a KAM closure C^{K} in the following way:

$$(i,n,s,a,r,q)^{\mathrm{K}} := (\langle f,c,m,\lceil a\rceil,d,\lceil q\rceil\rangle,[f{\leftarrow}(\bar{i},\epsilon)],[c{\leftarrow}n^{\mathrm{K}}],[m{\leftarrow}s^{\mathrm{K}}],[d{\leftarrow}r^{\mathrm{K}}])$$

where $s^{\mathrm{K}} = \begin{cases} (\lceil\varepsilon\rceil,\epsilon) & \text{if } s=\varepsilon \\ (\lambda x_1.\ldots.\lambda x_{|\Sigma|}.\lambda y.x_{i_a}z,[z{\leftarrow}r^{\mathrm{K}}]) & \text{if } s=a_i r \end{cases}$

We observe that this representation preserves the space consumption, i.e. it is reasonable.

Lemma B.2. *Let $C := (i,n,s,a,r,q)$ be a configuration of a Turing machine and $|C| := |s|+|r|$ its space consumption. Then $|C^{\mathrm{K}}| = \Theta(|C|+\log(|i|))$.*

In this lemma, we have already considered that the size of pointers inside n^{K}, s^{K}, r^{K} is constant and that $n \le \log|i|$.

Now we are able to prove the main theorem. A series of intermediate lemmata, about the different combinators used in the encoding ($\mathtt{init},\mathtt{final},\mathtt{trans}$), are necessary. They are stated and proved below the main statement . By $\to^*_f$, we mean that the space consumption of that series of transitions is f.

Theorem B.3 (TM are simulated by the Space KAM in reasonable space)**.** *There is an encoding $\bar{\cdot}$ of log-sensitive TM into $\Lambda_{\mathtt{det}}$ such that if the run ρ of the TM M on input $i \in \mathbb{B}^*$:*

1. Termination*: ends in q_b with $b\in\mathbb{B}$, then there is a complete sequence $\sigma : \overline{M}\,\bar{i} \to^n_{det} \bar{b}$ where $n = \Theta((T_{TM}(\rho)+1)\cdot|i|\cdot\log|i|)$.*
2. Divergence*: diverges, then $\overline{M}\,\bar{i}$ is $\to_{det}$-divergent.*
3. Space KAM*: the space used by the Space KAM to simulate the evaluation of point 1 is $\mathcal{O}(S_{TM}(\rho)+\log|i|)$ if $\overline{M}$ and $\bar{i}$ have separate address spaces.*

Proof. The first two points are proved in [ADLV23]. We concentrate on the third point.

We simply evaluate $\overline{M}\,\bar{i}$ with the Space KAM.

Term	Env	Stack	
$\overline{M}\,\bar{i} := \mathtt{init}(\mathtt{trans}^M(\mathtt{final}(\lambda x.x)))\bar{i}$	ϵ	ϵ	$\to^*_{\mathcal{O}(\log(\lvert i\rvert))}$ (Lemma B.4)
$\mathtt{trans}(\mathtt{final}(\lambda x.x))$	ϵ	$C_{\mathtt{in}}(i)^{\mathrm{K}}$	$\to^*_{\mathcal{O}(S_{\mathrm{TM}}(\rho)+\log\lvert i\rvert)}$ (Lemma B.8)
$\mathtt{final}(\lambda x.x)$	ϵ	D^{K}	$\to^*_{\mathcal{O}(S_{\mathrm{TM}}(\rho)+\log\lvert i\rvert)}$ (Lemma B.5)
$\bar{b}$	ϵ	ϵ	

□

Init and Final. Here, we provide the execution traces for the combinators `init` and `final`.

Lemma B.4. $(\texttt{init}\, k\, \bar{i}, \epsilon, \epsilon) \to_{\text{SpKAM}}^{\mathcal{O}(1)} (k, \epsilon, C_{\texttt{in}}(i)^{\text{K}})$ *and consumes space* $\Theta(\log(|i|))$.

Proof. The Space KAM execution is in Figure 8. The space bound is immediate by inspecting the execution. □

Lemma B.5. *Let C be a final configuration, i.e. $C := (i, n, s, a, r, q_{fin})$ where $q_{fin} \in Q_{fin}$. Then*

$$(\texttt{final}(\lambda x.x), \epsilon, C^{\text{K}}) \to_{\text{SpKAM}}^{\mathcal{O}(1)} \begin{cases} (\lambda x.\lambda y.x, \epsilon, \epsilon) & \text{if } q_{fin} = q_T \\ (\lambda x.\lambda y.y, \epsilon, \epsilon) & \text{if } q_{fin} = q_F \end{cases}$$

Moreover, the space consumption is $\Theta(|C^{\text{K}}|)$.

Proof. Let us define $t := \lambda i'.\lambda n'.\lambda w_l'.\lambda a'.\lambda w_r'.\lambda q'.q' N_1 \dots N_{|Q|} k'$. We execute the Space KAM on this term in Figure 9.

Two cases. If $q_{fin} = q_T$, then:

Term	Env	Stack	
$\lceil q_T \rceil := \lambda x_1 \dots \lambda x_{\lvert Q\rvert}.x_i$	ϵ	$(N_i, \epsilon)_{1 \le i \le \lvert Q\rvert} \cdot (\lambda x.x, \epsilon)$	$\to_\beta^{\lvert Q\rvert}$
x_i	$[x_i \leftarrow (N_i, \epsilon)]$	$(\lambda x.x, \epsilon)$	$\to_{\mathsf{sub}}$
$N_i := \lambda k'.k'(\lambda x.\lambda y.x)$	ϵ	$(\lambda x.x, \epsilon)$	$\to_{\beta_{\neg \mathsf{w}}}$
$k'(\lambda x.\lambda y.x)$	$[k' \leftarrow (\lambda x.x, \epsilon)]$	ϵ	$\to_{\mathsf{sea}_{\neg \mathsf{v}}}$
k'	$[k' \leftarrow (\lambda x.x, \epsilon)]$	$(\lambda x.\lambda y.x, \epsilon)$	$\to_{\mathsf{sub}}$
$\lambda x.x$	ϵ	$(\lambda x.\lambda y.x, \epsilon)$	$\to_{\beta_{\neg \mathsf{w}}}$
x	$[x \leftarrow (\lambda x.\lambda y.x, \epsilon)]$	ϵ	$\to_{\mathsf{sub}}$
$\lambda x.\lambda y.x$	ϵ	ϵ	

If $q_{fin} = q_F$, then:

Term	Env	Stack	
$\lceil q_F \rceil := \lambda x_1 \dots \lambda x_{\lvert Q\rvert}.x_i$	ϵ	$(N_i, \epsilon)_{1 \le i \le \lvert Q\rvert} \cdot (\lambda x.x, \epsilon)$	$\to_\beta^{\lvert Q\rvert}$
x_i	$[x_i \leftarrow (N_i, \epsilon)]$	$(\lambda x.x, \epsilon)$	$\to_{\mathsf{sub}}$
$N_i := \lambda k'.k'(\lambda x.\lambda y.y)$	ϵ	$(\lambda x.x, \epsilon)$	$\to_{\beta_{\neg \mathsf{w}}}$
$k'(\lambda x.\lambda y.y)$	$[k' \leftarrow (\lambda x.x, \epsilon)]$	ϵ	$\to_{\mathsf{sea}_{\neg \mathsf{v}}}$
k'	$[k' \leftarrow (\lambda x.x, \epsilon)]$	$(\lambda x.\lambda y.y, \epsilon)$	$\to_{\mathsf{sub}}$
$\lambda x.x$	ϵ	$(\lambda x.\lambda y.y, \epsilon)$	$\to_{\beta_{\neg \mathsf{w}}}$
x	$[x \leftarrow (\lambda x.\lambda y.y, \epsilon)]$	ϵ	$\to_{\mathsf{sub}}$
$\lambda x.\lambda y.y$	ϵ	ϵ	

The space bound is immediate by inspecting the execution. □

Transition Function. Here we execute the combinator $\texttt{trans}^M$ (abbreviated to `trans` for readability reasons), i.e. the main ingredient of the encoding. First, we execute the initialization steps.

Lemma B.6. $(\texttt{trans}\, k, \epsilon, C_{\texttt{in}}(i)^{\text{K}}) \to_{\text{SpKAM}}^{\mathcal{O}(1)} (\theta, \epsilon, (\theta, \epsilon) \cdot (\texttt{transaux}, \epsilon) \cdot (k, \epsilon) \cdot C_{\texttt{in}}(i)^{\text{K}})$ *in space* $\mathcal{O}(\log(|i|))$.

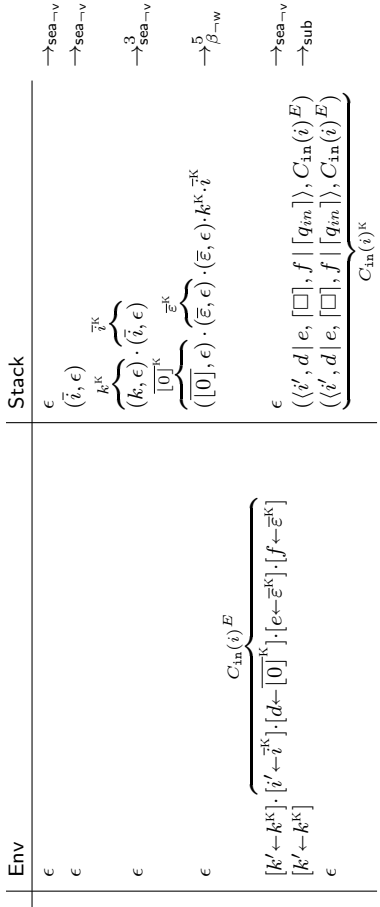
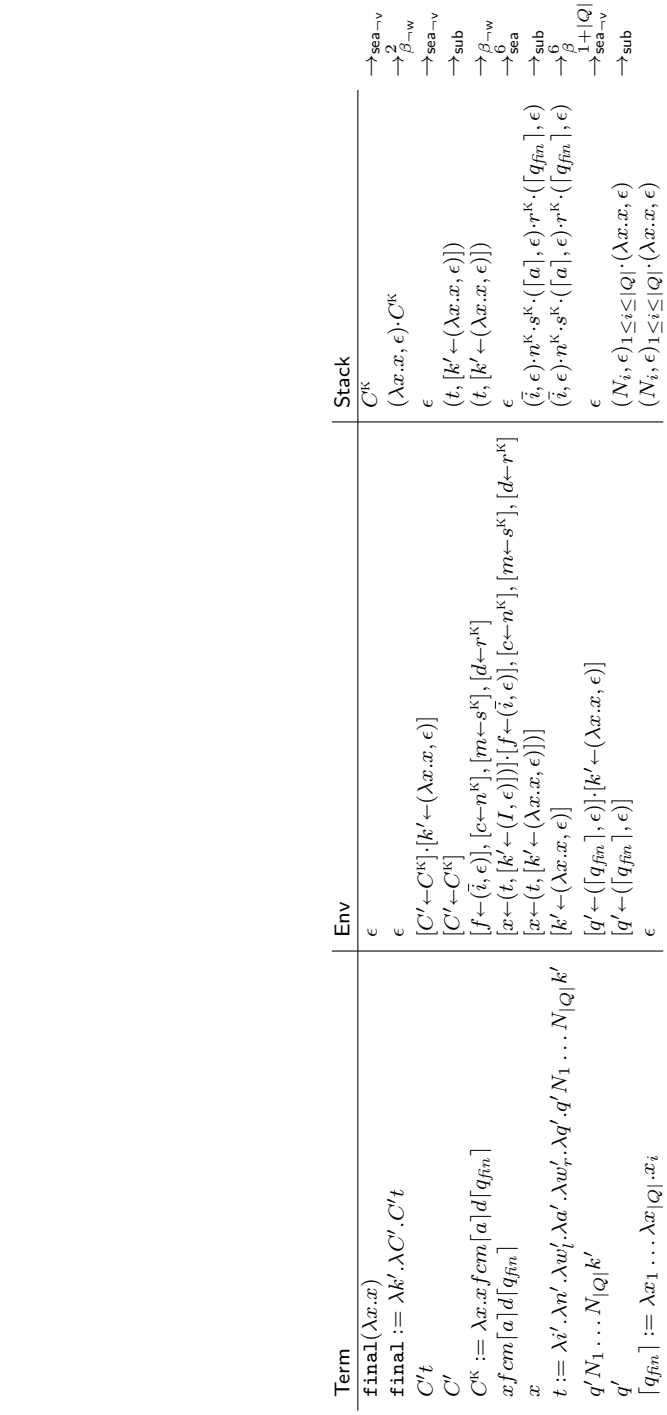


Figure 8: Space KAM execution of the `init` combinator.



Term	Env	Stack	
$\texttt{final}(\lambda x.x)$	ϵ	C^{K}	$\to_{\mathsf{sea}_\neg \mathsf{v}}$
$\texttt{final} := \lambda k'.\lambda C'.C'\,t$	ϵ	$(\lambda x.x, \epsilon)\cdot C^{\mathsf{K}}$	$\to^{2}_{\beta_\neg \mathsf{w}}$
$C'\,t$	$[C' \leftarrow C'^{\mathsf{K}}]\cdot[k' \leftarrow (\lambda x.x, \epsilon)]$	ϵ	$\to_{\mathsf{sea}_\neg \mathsf{v}}$
C'	$[C' \leftarrow C'^{\mathsf{K}}]$	$(t, [k' \leftarrow (\lambda x.x, \epsilon)])$	$\to_{\mathsf{sub}}$
$C^{\mathsf{K}} := \lambda x.x\, fcm\lceil a \rceil d\lceil q_{fin} \rceil$	$[f \leftarrow (\vec{i}, \epsilon)], [c \leftarrow n^{\mathsf{K}}], [m \leftarrow s^{\mathsf{K}}], [d \leftarrow r^{\mathsf{K}}]$	$(t, [k' \leftarrow (\lambda x.x, \epsilon)])$	$\to^{\beta_\neg \mathsf{w}}$
$x\, fcm\lceil a \rceil d\lceil q_{fin} \rceil$	$[x \leftarrow (t, [k' \leftarrow (I, \epsilon)])]\cdot[f \leftarrow (\vec{i}, \epsilon)], [c \leftarrow n^{\mathsf{K}}], [m \leftarrow s^{\mathsf{K}}], [d \leftarrow r^{\mathsf{K}}]$	ϵ	$\to^{6}_{\mathsf{sea}}$
x	$[x \leftarrow (t, [k' \leftarrow (\lambda x.x, \epsilon)])]$	$(\vec{i}, \epsilon)\cdot n^{\mathsf{K}}\cdot s^{\mathsf{K}}\cdot(\lceil a \rceil, \epsilon)\cdot r^{\mathsf{K}}\cdot(\lceil q_{fin} \rceil, \epsilon)$	$\to_{\mathsf{sub}}$
$t := \lambda i'.\lambda n'.\lambda w'_l.\lambda a'.\lambda w'_r.\lambda q'.q'\,N_1 \dots N_{\lvert Q \rvert}\,k'$	$[k' \leftarrow (\lambda x.x, \epsilon)]$	$(\vec{i}, \epsilon)\cdot n^{\mathsf{K}}\cdot s^{\mathsf{K}}\cdot(\lceil a \rceil, \epsilon)\cdot r^{\mathsf{K}}\cdot(\lceil q_{fin} \rceil, \epsilon)$	$\to^{6}_{\beta}$
$q'N_1 \dots N_{\lvert Q \rvert}k'$	$[q' \leftarrow (\lceil q_{fin} \rceil, \epsilon)]\cdot[k' \leftarrow (\lambda x.x, \epsilon)]$	ϵ	$\to^{1+\lvert Q \rvert}_{\mathsf{sea}_\neg \mathsf{v}}$
q'	$[q' \leftarrow (\lceil q_{fin} \rceil, \epsilon)]$	$(N_i, \epsilon)_{1 \le i \le \lvert Q \rvert}\cdot(\lambda x.x, \epsilon)$	$\to_{\mathsf{sub}}$
$\lceil q_{fin} \rceil := \lambda x_1 \dots \lambda x_{\lvert Q \rvert}.x_i$	ϵ	$(N_i, \epsilon)_{1 \le i \le \lvert Q \rvert}\cdot(\lambda x.x, \epsilon)$	

Figure 9: First part of the Space KAM execution of the `final` combinator.

Proof.

Term	Env	Stack	
$\texttt{trans}k$	ϵ	$C_{\texttt{in}}(i)^{\mathrm{K}}$	$\to_{\mathsf{sea}_{\neg\mathsf{v}}}$
$\texttt{trans} := \mathsf{fix}\,\texttt{transaux}$	ϵ	$(k,\epsilon)\cdot C_{\texttt{in}}(i)^{\mathrm{K}}$	$\to^*$ (Lemma A.1)
θ	ϵ	$(\theta,\epsilon)\cdot(\texttt{transaux},\epsilon)\cdot(k,\epsilon)\cdot C_{\texttt{in}}(i)^{\mathrm{K}}$	

□

Then, we prove the main lemma about the `trans` combinator. It simply states that each transition of the Turing machine is simulated by the Space KAM, with the right, i.e. linear, space complexity overhead.

Lemma B.7. *Let C be a Turing machine configuration. Then:*

- *if C is a final configuration, then $(\theta,\epsilon,(\theta,\epsilon)\cdot(\texttt{transaux},\epsilon)\cdot k^{\mathrm{K}}\cdot C^{\mathrm{K}}) \to_{\mathrm{SpKAM}}{}^{\mathcal{O}(1)} (k,\epsilon,C^{\mathrm{K}})$ in space $\mathcal{O}(|C^{\mathrm{K}}|)$;*
- *otherwise if $C \to_{\mathcal{M}} D$, then $(\theta,\epsilon,(\theta,\epsilon)\cdot(\texttt{transaux},\epsilon)\cdot k^{\mathrm{K}}\cdot C^{\mathrm{K}}) \to_{\mathrm{SpKAM}}{}^{\mathcal{O}(1)} (\theta,\epsilon,(\theta,\epsilon)\cdot(\texttt{transaux},\epsilon)\cdot k^{\mathrm{K}}\cdot D^{\mathrm{K}})$ in space $\mathcal{O}(|C^{\mathrm{K}}|)$.*

Proof. The first part of the proof is common to both points.
Let us define $\texttt{tx} := \texttt{transaux}$ and $t := \lambda i'.\lambda n'.\lambda w'_l.\lambda a'.\lambda w'_r.\lambda q'.\texttt{lookup}\,K i' n'$. The execution is in Figure 10. Cases of the transition to apply:

- *No transition*, that is, C is a final configuration, which happens when $q_g \in Q_{fin}$.
  We have $C_{i,j,q_g} := \lambda x.\lambda k'.\lambda i'.\lambda n'.\lambda w'_l.\lambda w'_r.k'\langle i', n' \mid w'_l, \lceil a_j \rceil, w'_r \mid \lceil q_g \rceil\rangle$, and
  $C^{\mathrm{K}} := (\langle i', n' \mid w'_l, \lceil a_j \rceil, w'_r \mid \lceil q_g \rceil\rangle, E_2)$, where $E_2 := [w'_l{\leftarrow}s^{\mathrm{K}}]\cdot[w'_r{\leftarrow}r^{\mathrm{K}}]\cdot[i'{\leftarrow}\bar{i}^{\mathrm{K}}]\cdot[n'{\leftarrow}n^{\mathrm{K}}]$

Term	Env	Stack	
C_{i,j,q_g}	ϵ	$\mathsf{fix}^{\mathrm{K}}\cdot k^{\mathrm{K}}\cdot\bar{i}^{\mathrm{K}}\cdot n^{\mathrm{K}}\cdot s^{\mathrm{K}}\cdot r^{\mathrm{K}}$	$\to^6_\beta$
$k'\langle i', n' \mid w'_l, \lceil a_j \rceil, w'_r \mid \lceil q_g \rceil\rangle$	$[k'{\leftarrow}k^{\mathrm{K}}]\cdot\overbrace{[w'_l{\leftarrow}s^{\mathrm{K}}]\cdot[w'_r{\leftarrow}r^{\mathrm{K}}]\cdot[i'{\leftarrow}\bar{i}^{\mathrm{K}}]\cdot[n'{\leftarrow}n^{\mathrm{K}}]}^{E_2}$	ϵ	$\to_{\mathsf{sea}_{\neg\mathsf{v}}}$
k'	$[k'{\leftarrow}k^{\mathrm{K}} =: (k,\epsilon)]$	C^{K}	$\to_{\mathsf{sub}}$
k	E	C^{K}	

- *The heads do not move*, that is, $\delta(a_i,a_j,q_g) = (0 \mid a_h, \downarrow \mid q_l)$. We set
  $D^{\mathrm{K}} := (\langle i', n'' \mid w'_l, \lceil a_h \rceil, w'_r \mid \lceil q_l \rceil\rangle, E_2)$, where $E_2 := [w'_l{\leftarrow}s^{\mathrm{K}}]\cdot[w'_r{\leftarrow}r^{\mathrm{K}}]\cdot[i'{\leftarrow}\bar{i}^{\mathrm{K}}]\cdot[n''{\leftarrow}n^{\mathrm{K}}]$.

Term	Env	Stack	
$C_{i,j,q_g} := \lambda x.\lambda k'.\lambda i'.\lambda n'.\lambda w'_l.\lambda w'_r.Sn'$	ϵ	$\mathsf{fix}^{\mathrm{K}}\cdot k^{\mathrm{K}}\cdot\bar{i}^{\mathrm{K}}\cdot n^{\mathrm{K}}\cdot s^{\mathrm{K}}\cdot r^{\mathrm{K}}$	$\to^6_{\beta_{\neg\mathsf{w}}}$
Sn'	$[x{\leftarrow}\mathsf{fix}^{\mathrm{K}}]\cdot[k'{\leftarrow}k^{\mathrm{K}}]\cdot[w'_l{\leftarrow}s^{\mathrm{K}}]\cdot[w'_r{\leftarrow}r^{\mathrm{K}}]\cdot[i'{\leftarrow}\bar{i}^{\mathrm{K}}]\cdot[n'{\leftarrow}n^{\mathrm{K}}]$	ϵ	$\to_{\mathsf{sea}_{\mathsf{v}}}$
$S := \lambda n''.xk'\langle i', n'' \mid w'_l, \lceil a_h \rceil, w'_r \mid \lceil q_l \rceil\rangle$	$[x{\leftarrow}\mathsf{fix}^{\mathrm{K}}]\cdot[k'{\leftarrow}k^{\mathrm{K}}]\cdot[w'_l{\leftarrow}s^{\mathrm{K}}]\cdot[w'_r{\leftarrow}r^{\mathrm{K}}]\cdot[i'{\leftarrow}\bar{i}^{\mathrm{K}}]$	n^{K}	$\to_{\beta_{\neg\mathsf{w}}}$
$xk'\langle i', n'' \mid w'_l, \lceil a_h \rceil, w'_r \mid \lceil q_l \rceil\rangle$	$[x{\leftarrow}\mathsf{fix}^{\mathrm{K}}]\cdot[k'{\leftarrow}k^{\mathrm{K}}]\cdot\overbrace{[w'_l{\leftarrow}s^{\mathrm{K}}]\cdot[w'_r{\leftarrow}r^{\mathrm{K}}]\cdot[i'{\leftarrow}\bar{i}^{\mathrm{K}}]\cdot[n''{\leftarrow}n^{\mathrm{K}}]}^{E_2}$	ϵ	$\to_{\mathsf{sea}}$
x	$[x{\leftarrow}\mathsf{fix}^{\mathrm{K}}]$	$k^{\mathrm{K}}\cdot D^{\mathrm{K}}$	$\to_{\mathsf{sub}}$
xxy	$[y{\leftarrow}(\texttt{tx},\epsilon)]\cdot[x{\leftarrow}(\theta,\epsilon)]$	$k^{\mathrm{K}}\cdot D^{\mathrm{K}}$	$\to^2_{\mathsf{sea}_{\mathsf{v}}}$
x	$[x{\leftarrow}(\theta,\epsilon)]$	$(\theta,\epsilon)\cdot(\texttt{tx},\epsilon)\cdot k^{\mathrm{K}}\cdot D^{\mathrm{K}}$	$\to_{\mathsf{sub}}$
θ	ϵ	$(\theta,\epsilon)\cdot(\texttt{tx},\epsilon)\cdot k^{\mathrm{K}}\cdot D^{\mathrm{K}}$	

- *The heads move right*, that is, $\delta(a_i,a_j,q_g) = (1 \mid a_h, \to \mid q_l)$. The execution of the first part is in Figure 11. Two cases.
  – $r = \varepsilon$. Define $t := (\lambda d.\lambda w'_l.xk'\langle i', n' \mid w'_l, \lceil\square\rceil, d \mid \lceil q_l \rceil\rangle)\bar{\varepsilon}$. The execution is in Figure 12.
  – $r = a''\cdot r'$. Define $t := \lambda w'_l.xk'\langle i', n' \mid w'_l, \lceil a''\rceil, w'_r \mid \lceil q_l \rceil\rangle$. The execution is in Figure 13.
- All the other cases are almost identical *mutatis mutandis*.

Term	Env	Stack	
$\theta := \lambda x.\lambda y.y(xxy)$	ϵ	$(\theta,\epsilon)\cdot(\mathtt{tx},\epsilon)\cdot k^{\mathsf{K}}\cdot C^{\mathsf{K}}$	$\to^{2}_{\beta_{\neg\mathsf{w}}}$
$y(xxy)$	$[x\leftarrow(\theta,\epsilon)]\cdot[y\leftarrow(\mathtt{tx},\epsilon)]$	$k^{\mathsf{K}}\cdot C^{\mathsf{K}}$	$\to_{\mathsf{sea}_{\neg\mathsf{v}}}$
y	$[y\leftarrow(\mathtt{tx},\epsilon)]$	$\overbrace{(xxy,[y\leftarrow(\mathtt{tx},\epsilon)]\cdot[x\leftarrow(\theta,\epsilon)])}^{\mathsf{fix}^{\mathsf{K}}}\cdot k^{\mathsf{K}}\cdot C^{\mathsf{K}}$	$\to_{\mathsf{sub}}$
$\mathtt{tx}$	ϵ	$\mathsf{fix}^{\mathsf{K}}\cdot k^{\mathsf{K}}\cdot C^{\mathsf{K}}$	$=$
$\lambda x.\lambda k.\lambda C'.C't$	ϵ	$\mathsf{fix}^{\mathsf{K}}\cdot k^{\mathsf{K}}\cdot C^{\mathsf{K}}$	$\to^{2}_{\beta_{\neg\mathsf{w}}}$
	$\overbrace{[k'\leftarrow k^{\mathsf{K}}]\cdot[x\leftarrow\mathsf{fix}^{\mathsf{K}}]}^{E}$		
$\lambda C'.C't$	$[C'\leftarrow C^{\mathsf{K}}]\cdot E$	C^{K}	$\to_{\beta_{\neg\mathsf{w}}}$
$C't$	$[C'\leftarrow C^{\mathsf{K}}]$	ϵ	$\to_{\mathsf{sea}_{\neg\mathsf{v}}}$
C'	$[f\leftarrow(\bar{i},\epsilon)],[c\leftarrow n^{\mathsf{K}}],[m\leftarrow s^{\mathsf{K}}],[d\leftarrow r^{\mathsf{K}}]$	(t,E)	$\to_{\mathsf{sub}}$
$\lambda x.xfcm\lceil a_j\rceil d\lceil q_g\rceil$		(t,E)	$\to^{7}$
	$[x\leftarrow(t,E)]$	$\overbrace{(\bar{i},\epsilon)}^{\bar{i}^{\mathsf{K}}}\cdot n^{\mathsf{K}}\cdot s^{\mathsf{K}}\cdot\overbrace{(\lceil a_j\rceil,\epsilon)}^{\lceil a\rceil^{\mathsf{K}}}\cdot r^{\mathsf{K}}\cdot\overbrace{(\lceil q_g\rceil,\epsilon)}^{\lceil q\rceil^{\mathsf{K}}}$	$\to_{\mathsf{sub}}$
x	E	$\bar{i}^{\mathsf{K}}\cdot n^{\mathsf{K}}\cdot s^{\mathsf{K}}\cdot\lceil a\rceil^{\mathsf{K}}\cdot r^{\mathsf{K}}\cdot\lceil q\rceil^{\mathsf{K}}$	$\to^{6}_{\beta_{\neg\mathsf{w}}}$
$\lambda i'.\lambda n'.\lambda w'_l.\lambda a'.\lambda w'_r.\lambda q'.\mathtt{lookup}\ Ki'n'$	$\overbrace{[i'\leftarrow\bar{i}^{\mathsf{K}}]\cdot[n'\leftarrow n^{\mathsf{K}}]\cdot[w_l\leftarrow s^{\mathsf{K}}]\cdot[a'\leftarrow\lceil a\rceil^{\mathsf{K}}]\cdot[w_r\leftarrow r^{\mathsf{K}}]\cdot[q'\leftarrow\lceil q\rceil^{\mathsf{K}}]\cdot E}^{E'}$		
$\mathtt{lookup}\ Ki'n'$	ϵ	ϵ	$\to^{3}_{\mathsf{sea}}$
$\mathtt{lookup}$	E'	$(K,E')\cdot\bar{i}^{\mathsf{K}}\cdot n^{\mathsf{K}}$	$\to^{*}$
$K := \lambda b'.b'A_0A_1A_{\mathsf{L}}A_{\mathsf{R}}a'q'xk'i'n'w'_lw'_r$	E'	$(\lceil a_i\rceil,\epsilon)$	$\to^{12}_{\beta_{\neg\mathsf{w}}}$
$b'A_0A_1A_{\mathsf{L}}A_{\mathsf{R}}a'q'xk'i'n'w'_lw'_r$	$[b'\leftarrow(\lceil a_i\rceil,\epsilon)],E'$	ϵ	$\to^{1}_{\mathsf{sea}}$
b'	$[b'\leftarrow(\lceil a_i\rceil,\epsilon)]$	$(A_b,\epsilon)_{b\in\mathbb{B}_1}\cdot\lceil a\rceil^{\mathsf{K}}\cdot\lceil q\rceil^{\mathsf{K}}\cdot\mathsf{fix}^{\mathsf{K}}\cdot k^{\mathsf{K}}\cdot\bar{i}^{\mathsf{K}}\cdot n^{\mathsf{K}}\cdot s^{\mathsf{K}}\cdot r^{\mathsf{K}}$	$\to_{\mathsf{sub}}$
$\lceil a_i\rceil := \lambda x_0.\lambda x_1.\lambda x_{\mathsf{L}}.\lambda x_{\mathsf{R}}.x_i$	ϵ	$(A_b,\epsilon)_{b\in\mathbb{B}_1}\cdot\lceil a\rceil^{\mathsf{K}}\cdot\lceil q\rceil^{\mathsf{K}}\cdot\mathsf{fix}^{\mathsf{K}}\cdot k^{\mathsf{K}}\cdot\bar{i}^{\mathsf{K}}\cdot n^{\mathsf{K}}\cdot s^{\mathsf{K}}\cdot r^{\mathsf{K}}$	$\to^{5}$
$A_i := \lambda a'.a'B_{i,0}B_{i,1}B_{i,\square}$	ϵ	$\lceil a\rceil^{\mathsf{K}}\cdot\lceil q\rceil^{\mathsf{K}}\cdot\mathsf{fix}^{\mathsf{K}}\cdot k^{\mathsf{K}}\cdot\bar{i}^{\mathsf{K}}\cdot n^{\mathsf{K}}\cdot s^{\mathsf{K}}\cdot r^{\mathsf{K}}$	$\to_{\beta_{\neg\mathsf{w}}}$
$a'B_{i,0}B_{i,1}B_{i,\square}$	$[a'\leftarrow\lceil a\rceil^{\mathsf{K}}]$	$\lceil q\rceil^{\mathsf{K}}\cdot\mathsf{fix}^{\mathsf{K}}\cdot k^{\mathsf{K}}\cdot\bar{i}^{\mathsf{K}}\cdot n^{\mathsf{K}}\cdot s^{\mathsf{K}}\cdot r^{\mathsf{K}}$	$\to^{3}_{\mathsf{sea}_{\neg\mathsf{v}}}$
a'	$[a'\leftarrow\lceil a\rceil^{\mathsf{K}}]$	$(B_{i,b},\epsilon)_{b\in\mathbb{B}_{\mathsf{W}}}\cdot\lceil q\rceil^{\mathsf{K}}\cdot\mathsf{fix}^{\mathsf{K}}\cdot k^{\mathsf{K}}\cdot\bar{i}^{\mathsf{K}}\cdot n^{\mathsf{K}}\cdot s^{\mathsf{K}}\cdot r^{\mathsf{K}}$	$\to_{\mathsf{sub}}$
$\lceil a_j\rceil := \lambda x_0.\lambda x_1.\lambda x_{\square}.x_j$	ϵ	$(B_{i,b},\epsilon)_{b\in\mathbb{B}_{\mathsf{W}}}\cdot\lceil q\rceil^{\mathsf{K}}\cdot\mathsf{fix}^{\mathsf{K}}\cdot k^{\mathsf{K}}\cdot\bar{i}^{\mathsf{K}}\cdot n^{\mathsf{K}}\cdot s^{\mathsf{K}}\cdot r^{\mathsf{K}}$	$\to^{4}$
$B_{i,j} := \lambda q'.q'C_{i,j,q_1}\dots C_{i,j,q_{\lvert Q\rvert}}$	ϵ	$\lceil q\rceil^{\mathsf{K}}\cdot\mathsf{fix}^{\mathsf{K}}\cdot k^{\mathsf{K}}\cdot\bar{i}^{\mathsf{K}}\cdot n^{\mathsf{K}}\cdot s^{\mathsf{K}}\cdot r^{\mathsf{K}}$	$\to_{\beta_{\neg\mathsf{w}}}$
$q'C_{i,j,q_1}\dots C_{i,j,q_{\lvert Q\rvert}}$	$[q'\leftarrow\lceil q\rceil^{\mathsf{K}}]$	$\mathsf{fix}^{\mathsf{K}}\cdot k^{\mathsf{K}}\cdot\bar{i}^{\mathsf{K}}\cdot n^{\mathsf{K}}\cdot s^{\mathsf{K}}\cdot r^{\mathsf{K}}$	$\to^{\lvert Q\rvert}_{\mathsf{sea}_{\neg\mathsf{v}}}$
q'	$[q'\leftarrow\lceil q\rceil^{\mathsf{K}}]$	$(C_{i,j,q_g},\epsilon)_{1\le g\le\lvert\Sigma\rvert}\cdot\mathsf{fix}^{\mathsf{K}}\cdot k^{\mathsf{K}}\cdot\bar{i}^{\mathsf{K}}\cdot n^{\mathsf{K}}\cdot s^{\mathsf{K}}\cdot r^{\mathsf{K}}$	$\to_{\mathsf{sub}}$
$\lceil q_g\rceil := \lambda x_1\dots\lambda x_{\lvert Q\rvert}.x_g$	ϵ	$(C_{i,j,q_g},\epsilon)_{1\le g\le\lvert\Sigma\rvert}\cdot\mathsf{fix}^{\mathsf{K}}\cdot k^{\mathsf{K}}\cdot\bar{i}^{\mathsf{K}}\cdot n^{\mathsf{K}}\cdot s^{\mathsf{K}}\cdot r^{\mathsf{K}}$	$\to^{1+\lvert Q\rvert}$
C_{i,j,q_g}	ϵ	$\mathsf{fix}^{\mathsf{K}}\cdot k^{\mathsf{K}}\cdot\bar{i}^{\mathsf{K}}\cdot n^{\mathsf{K}}\cdot s^{\mathsf{K}}\cdot r^{\mathsf{K}}$	

Figure 10: The first part of the Space KAM execution of the combinator `trans`.

Term	Env	Stack	
$C_{i,j,q_g} := \lambda x.\lambda k'.\lambda i'.\lambda n'.\lambda w'_l.\lambda w'_r.\mathtt{succ}Rn'$	ϵ	$\mathsf{fix}^{\kappa}\cdot k^{\kappa}\cdot\overline{i}^{\kappa}\cdot n^{\kappa}\cdot s^{\kappa}\cdot r^{\kappa}$	$\to^{6}_{\beta\neg\mathsf{w}}$
$\mathtt{succ}Rn'$	$\overbrace{[x\leftarrow\mathsf{fix}^{\kappa}]\cdot[k'\leftarrow k^{\kappa}]\cdot[w'_l\leftarrow s^{\kappa}]\cdot[w'_r\leftarrow r^{\kappa}]\cdot[i'\leftarrow\overline{i}^{\kappa}]}^{E_2}\cdot[n'\leftarrow n^{\kappa}]$	ϵ	$\to^{2}_{\mathsf{sea}}$
$\mathtt{succ}$	ϵ	$(R, E_2)\cdot n^{\kappa}$	$\to^{*}$
$R := \lambda n''.w'_r R_0^{q_l,a_h} R_1^{q_l,a_h} R_{\square}^{q_l,a_h} R_{\varepsilon}^{q_l,a_h} xk'i'n''w'_l$	E_2	$m^{\kappa} := (n+1)^{\kappa}$	$\to^{\beta\neg\mathsf{w}}$
$w'_r R_0^{q_l,a_h} R_1^{q_l,a_h} R_{\square}^{q_l,a_h} R_{\varepsilon}^{q_l,a_h} xk'i'n''w'_l$	$[x\leftarrow\mathsf{fix}^{\kappa}]\cdot[k'\leftarrow k^{\kappa}]\cdot[w'_l\leftarrow s^{\kappa}]\cdot[w'_r\leftarrow r^{\kappa}]\cdot[i'\leftarrow\overline{i}^{\kappa}]\cdot[n''\leftarrow m^{\kappa}]$	ϵ	$\to^{9}_{\mathsf{sea}_{\mathsf{v}}}$
w'_r	$[w'_r\leftarrow r^{\kappa}]$	$(R^{q_l,a_h}_{\mathtt{x}}, \epsilon)_{\mathtt{x}\in\{0,1,\square,\varepsilon\}}\cdot\mathsf{fix}^{\kappa}\cdot k^{\kappa}\cdot\overline{i}^{\kappa}\cdot m^{\kappa}\cdot s^{\kappa}$	

Figure 11: The Space KAM execution of the beginning of "the heads move right".

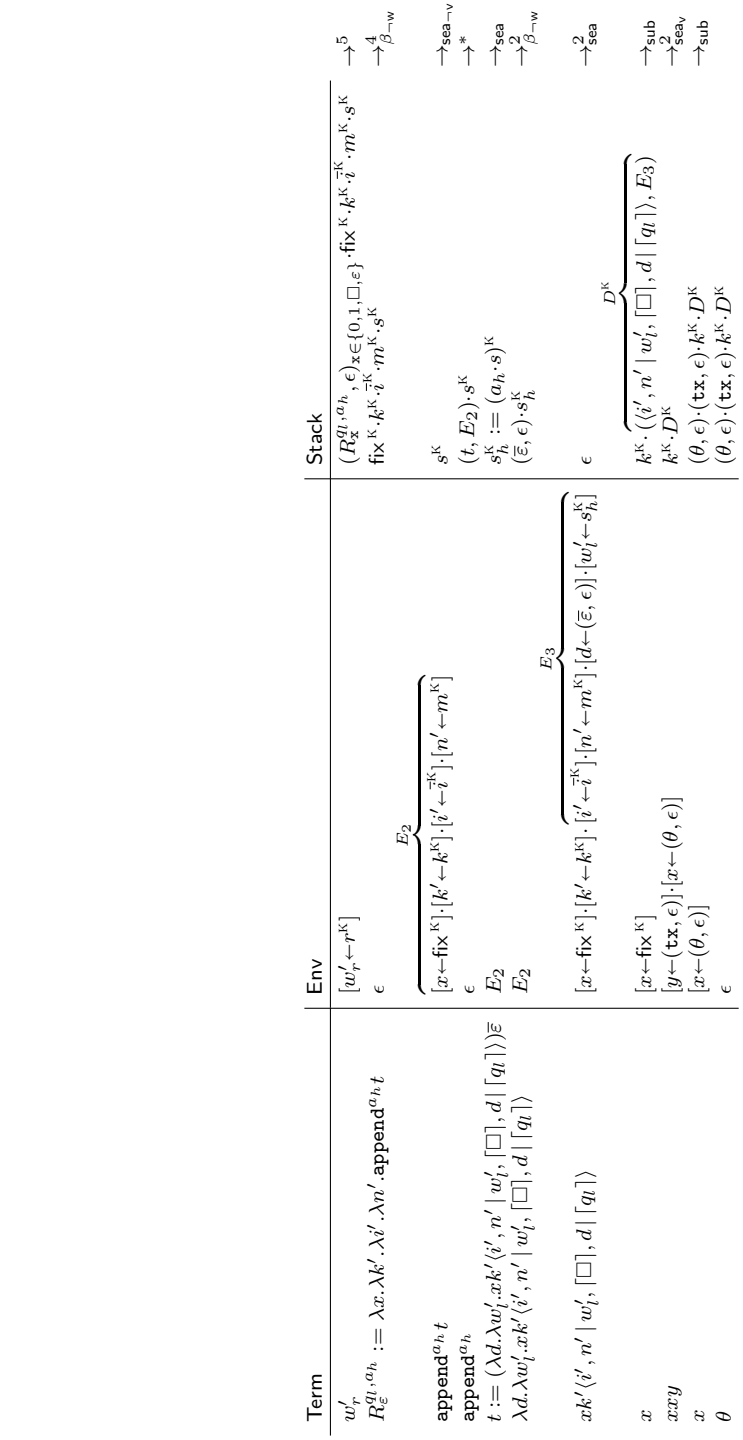


Figure 12: The Space KAM execution of the sequel of "the heads move right", case $r = \varepsilon$.

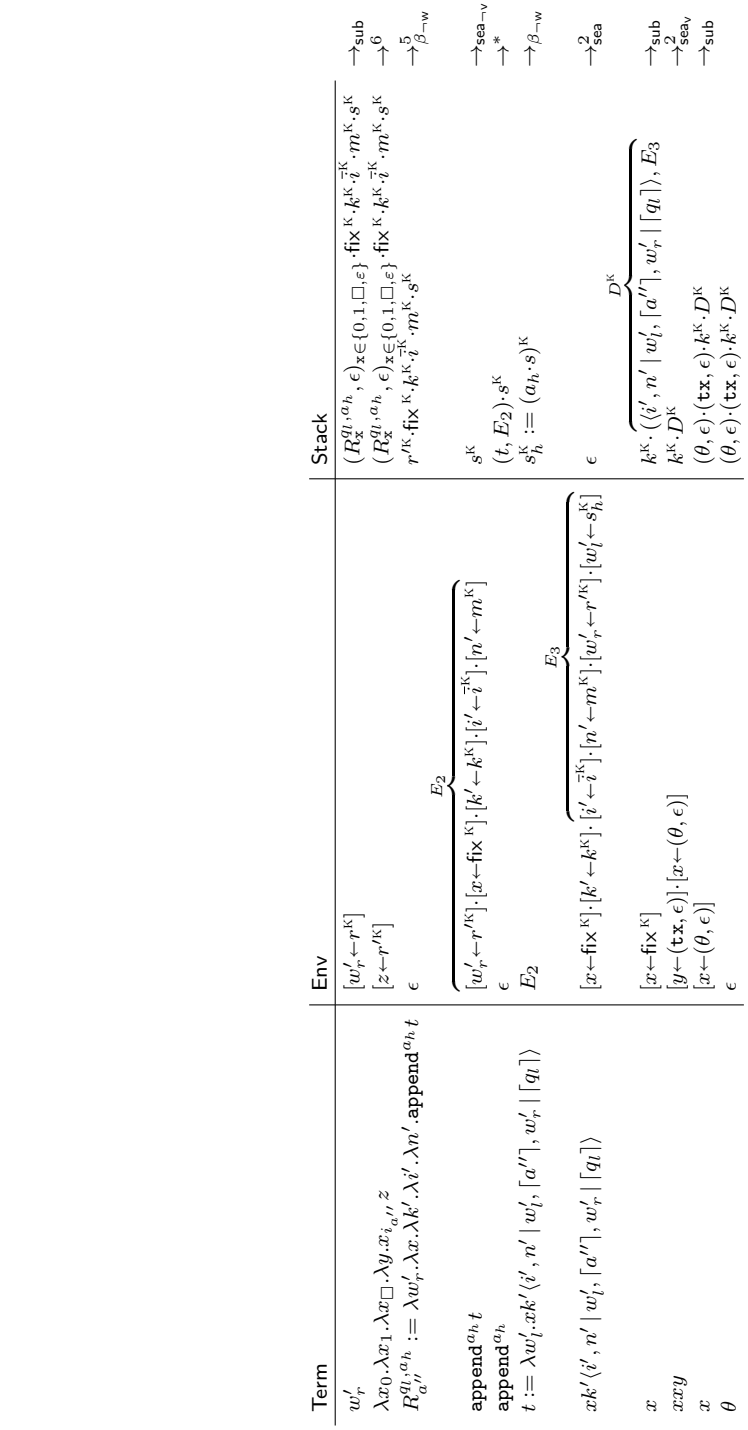


Figure 13: The Space KAM execution of the sequel of "the heads move right", case $r = a'' \cdot r'$.

About the space bound we observe that in the simulations *all* the pointers except for those related to the input part of the state, which are in *fixed* number, are pointers to the machine, and not to the input. Moreover, the space overhead of the simulation of one step of the TM is constant, i.e. non input dependent. □

Lemma B.8. *If* $\rho : C \to^n D$ *and* D *is final, then* $(\mathtt{trans}\, k, \epsilon, C_{\mathtt{in}}(i)^{\mathrm{K}}) \to_{\mathrm{SpKAM}} (k, \epsilon, C^{\mathrm{K}})$ *in space* $\mathcal{O}(S_{TM}(\rho) + \log(|i|))$.

Proof. By a simple induction on n, using the two lemmata above, and knowing that $S_{\mathrm{TM}}(\rho) = \max_{C \in \rho} |C|$ (we have also to consider that $|C| = |C^{\mathrm{K}}|$, by Lemma B.2). □