Reasonable Space for the λ-Calculus, Logarithmically

Beniamino Accattoli
Inria & LIX, École Polytechnique
France
beniamino.accattoli@inria.fr

Ugo Dal Lago
Università di Bologna & Inria
Italy
ugo.dallago@unibo.it

Gabriele Vanoni
Università di Bologna & Inria
Italy
gabriele.vanoni2@unibo.it

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 and show how to transport our results to the call-by-value λ-calculus.

CCS CONCEPTS

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

KEYWORDS

lambda-calculus, abstract machines, complexity

ACM Reference Format:
Beniamino Accattoli, Ugo Dal Lago, and Gabriele Vanoni. 2022. Reasonable Space for the λ-Calculus, Logarithmically. In *37th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS) (LICS '22), August 2–5, 2022, Haifa, Israel.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3531130.3533362

1 INTRODUCTION

Overview. Bounding the amount of resources needed by algorithms or programs is a fundamental problem in computer science. Here we are concerned with sub-linear 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. 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.

Dealing with sub-linear space bounds in the λ-calculus, or in functional programming languages, has always been considered a challenge. The first reason is that the natural notions of time and space in the λ-calculus have some puzzling properties, as we shall see. But sub-linear space is special, in that, since the λ-calculus does not distinguish between programs and data, there is also no distinction between input space and work space, and thus no natural notion of sub-linear space.

LICS '22, August 2–5, 2022, Haifa, Israel

ACM ISBN 978-1-4503-9351-5/22/08.
https://doi.org/10.1145/3531130.3533362

The literature about the λ-calculus does offer results about space complexity, but they are all *partial*, as they either concern variants of the λ-calculus (Dal Lago and Schöpp [19, 38], Mazza [33] and Ghica [25]), or they are not valid when the bounds in spaces are sub-linear (Forster et al. [22]).

The main contribution of this paper is the first fully-fledged space reasonability result for the pure, untyped λ-calculus. Precisely, we represent the *input space* as λ-terms, and the *work space* as the space used by a new variant of the well-known Krivine's abstract machine (KAM). Two important aspects of our *Space KAM* are *eager garbage collection* and the fact that we distinguish between two forms of sharing usually considered as one: the *sharing of subterms* provided by environments, and the *sharing of environments* themselves. The Space KAM adopts the former but forbids the latter, which is crucial for proving that its space cost model is reasonable. Designing the Space KAM, however, is only half of the story. The other half is the refinement of the encoding of Turing machines into the λ-calculus: our reference is that of Dal Lago and Accattoli [17], which uses a linear amount of extra space to simulate the TM tapes, thus forbidding to preserve logarithmic space.

Reasonable Cost Models. According to the seminal work by Slot and van Emde Boas [40], the adequacy of space and time cost models is judged in relationship to whether they reflect the corresponding cost models of Turing machines (shortened to TM), the computational theory 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

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

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.

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 natural cost models.

Natural Cost Models for the λ-Calculus. The natural time cost model is the number of β-steps (according to some fixed evaluation strategy), its only notion of computational step. The natural space cost model is the maximum size of λ-terms during reduction. The puzzling point is that, in the λ-calculus, natural space can be exponential in natural 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? This was unclear for a while, because of the intuition that reasonable cost models have to be locked. Note that there is, in principle, an alternative and non-explosive approach to time in the λ-calculus: the time to write down the whole evaluation rather than only the number of evaluation steps. Such *low-level* time is reasonable, and locked with natural space, but requires that the various β-steps in an evaluation sequence are given different costs, which is theoretically not ideal, and also distant from the practice of programming, that rather adopts the natural (and uniform) cost model which corresponds, in practice, to the number of function calls.

(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: the search of the β-redex and capture-avoiding substitution, itself based on 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 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.

Natural Time. In the study of natural 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 natural 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 [6, 16]). 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. The first such result is for weak evaluation by Blelloch and Greiner [14], then extended to strong evaluation by Accattoli and Dal Lago [6], and very recently transferred to Strong CbV by Accattoli et al. [4] and to a variant of Strong CbV by Biernacka et al. [13]. As we shall see, the very sharing mechanism at work in the simulations proving that natural time is reasonable, unfortunately, also makes those simulations space unreasonable.

Natural 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, *juxtaposes* cells rather then linking them via pointers—we shall see that 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 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 observation by Forster et al. [22] that evaluating without *any* data structure (via plain rewriting, without sharing) is reasonable for natural space even if unreasonable for natural time (because of explosiveness).

Pairing up Natural Time and Natural Space. Forster et al. [22] also show a surprising fact. Given two simulations, one that is reasonable for space but not time, and one that is reasonable for time but not space, there is a smart way of interleaving them as to obtain reasonability for time and space *simultaneously*. Their result therefore shows that, surprisingly, a computational theory can be reasonable for time *and* space without being locked. Whenever their interleaving technique applies, however, it also induces a second (less natural) reasonable cost model for space that is locked with the time cost model.

Work vs Natural Space. Another puzzling fact is that sub-linear space *cannot* be measured using the natural 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 we accommodate for *logarithmic* reasonable space? One needs *log-sensitivity*, that is, a distinction between an immutable *input space*, which is not counted for space complexity (because otherwise the complexity would be at least linear), and a (smaller) mutable *work space*, that is counted. Moreover, logarithmic space usually requires manipulating *pointers* to the input (which are of logarithmic size) rather than pieces of the input (which can be linear).

Log-sensitivity thus seems to clash with the natural approach based on the rewriting of λ-terms, which does not distinguish between input and work space and that manipulates actual sub-terms rather than pointers. We shall then model evaluation using λ-terms for the input space, and the space used by an abstract machine for the work space. The abstract machine shall compute over the input without ever modifying it, and shall also manipulate sub-term pointers.

An interesting aspect of the quest for a logarithmic reasonable work space is that it requires dismissing widespread intuitions about the problem, as we now explain.

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 [26]. Mackie's and Danos & Regnier's *interaction abstract machine* (shortened to IAM) [20, 32], and recently reformulated in [11], is a machine rooted in the geometry of interaction and in Abramsky et al.'s game semantics [1]. It is based on a log-sensitive approach, and—apparently—it is parsimonious with respect to space. Since the work of Schöpp [37, 38] (with later developments with Dal Lago [18]), who showed how IAM-like mechanisms can be used for dealing with logarithmic space, it was conjectured that the space of the IAM were a reasonable cost model. The belief in the conjecture was re-inforced by further uses of IAM-like mechanisms for space parsimony related to circuits, by Ghica [25], and for characterizing L, by Mazza [33]. However, in 2021, Accattoli et al. essentially disproved the conjecture: the space used by the IAM to evaluate the reference encoding of TMs is unreasonable [8] (as well as time inefficient [7]). While one might look for different encodings, the unreasonable behavior of the IAM concerns the modeling of recursion via fix-point combinators, which is hardly avoidable by any encoding.

A Wrong Negative Belief: Environments. Another misleading belief was that environment-based abstract machines could not be space reasonable. Environments are data structures used to achieve time reasonability. According to Accattoli and Barras [3], there are two main styles of environments, local and global. Global environments (as in the Milner Abstract Machine [3]) are log-insensitive because they work over the input space. Local environments (as in the KAM) are log-sensitive. There are two reasons why they are usually space unreasonable. Firstly, garbage collection is not usually accounted for, which leads to ever-increasing space usage, while reasonable space should be re-usable. Secondly, and more subtly, 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 share them. Sub-terms pointers, as mentioned above, are a key aspect of logarithmic space computations, and are thus crucial. Environment pointers, which accordingly to Douence and Fradet are the *essence* of the KAM [21], are instead what makes environments space unreasonable: they introduce a logarithmic *pointer overhead* that, at best, gives simulations of TMs with a $O(n \log n)$ overhead in space, instead of the required $O(n)$ for reasonability. It was then generally concluded that environments cannot provide reasonable space.

Work Space Without Environment Pointers. The literature on abstract machines assumes that pointers are used in implementations without however taking them into account in the underlying specification. Here, we are instead very careful with pointers. We design an abstract machine, the Space KAM, using local environments with sub-term pointers, but—crucially—*without* environment pointers. Similarly to the tapes of TMs, the environments of the Space KAM then are *not* linked structures, but simple *unstructured strings*. Consequently, the unreasonable pointer overhead vanishes.

The insight is that the use of pointers is both essential and dangerous for logarithmic space: sub-term pointers, that is, those to the input, are mandatory for log-sensitivity, while the environment ones—those to the working tape, essentially—are space unreasonable.

Garbage Collection and Unchaining. The Space KAM crucially relies also on two optimizations. One is *eager garbage collection*, to maximize space re-usability. It is implemented in the most unsophisticated of ways, because it cannot rely on any pointers or counters, as they would add an unreasonable space overhead. In contrast to common practice, the collection happens *eagerly*, that is, immediately and not when reaching a threshold. The second optimization is *environment unchaining*, a folklore tweak for avoiding space leaks. Essentially, dead closures are removed as soon as possible (eager gc) and alive closures are compacted (unchaining).

Encoding of TMs. Despite the fine tuning of the abstract machine, a reasonable simulation of TMs preserving logarithmic space is not yet obtained, as the reference encoding of TMs has some inherent limitations with respect to logarithmic space. We then analyze its shortcomings, concerning how tapes are represented and scrolled in the λ-calculus, and modify it accordingly. The main result of the paper is that the work space of the Space KAM—to be referred to simply as *the work space*—over the new encoding is a reasonable space cost model accommodating sub-linear space.

Our new encoding is carefully designed so as to retain the *indifference property* of the reference one, i.e. 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.

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

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.

Related Work. The space inefficiency of environment machines is also observed by Krishnaswami et al. [29], who propose techniques to alleviate it in the context of functional-reactive programming and based on linear types. The relevance for space of disabling environment sharing is also stressed by Paraskevopoulou and Appel [34] in their cost-aware study of closure conversion. A characterization of PSPACE in the λ-calculus is given by Gaboardi et al. [24], but it relies on alternating time rather than on a notion of space. The already-mentioned Dal Lago and Schöpp [19, 38] and Mazza [33] characterize L in variants of the λ-calculus, while

Jones characterizes L using a programming language but not based on the λ-calculus [27]. Blelloch and coauthors study in various papers [14, 15, 41] how to profile (that is, measure) space consumption of functional programs, also done by Sansom and Peyton Jones [36]. 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 [28]. We here need a basic eager form, that need not be time efficient, as the Space KAM is time unreasonable anyway.

Proofs. The proofs are in the associated technical report [10].

2 REASONABLE PRELIMINARIES

In the study of reasonable cost models for the λ-calculus, it is customary to show that the λ-calculus simulates Turing machines reasonably, and conversely that the λ-calculus can be simulated reasonably by random access machines (RAMs and TMs being both reasonable models) *up to sharing*. Since space is more delicate than time, we fix the involved theories and their cost measures carefully.

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

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

Free and *bound variables* are defined as usual: $\lambda x.t$ binds x in t. 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*:

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

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 Size 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 bound by $O(|t| \cdot \log |t|)$. 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 of constructors $|t|$ of t. Then a term with n constructors requires space $O(n \log n)$ to be represented. For our study, it is important to stress the difference between $|t|$ and $\|t\|$, because given a binary input string i, at first sight its encoding t_i as a λ-term satisfies $|t_i| = \Theta(|i|)$ and $\|t_i\| = O(|i| \cdot \log |i|)$, and so $\|t_i\|$ has an additional (unreasonable) logarithmic factor. In Sect. 6, 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, variable pointers have constant size, so that $\|t_i\| = |t_i| = \Theta(|i|)$, thus removing the unreasonable pointer overhead due to variables.

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 TM*.

A log-sensitive TM M computes the function $f : \mathbb{B}^* \to \mathbb{B}$ by a sequence of transitions $\rho : C_M(i) \to^n C'_M(f(i))$ where $i \in \mathbb{B}^*$, $C_M(i)$ is an initial configuration of M with input i and $C'_M(f(i))$ is a final configuration of M on the final state $q_{f(i)}$. We define the time of the run ρ as $T_{\text{TM}}(\rho) := n$ and the space $S_{\text{TM}}(\rho)$ of ρ as the maximum number of cells of the work tape used during the run ρ.

Random Access Machines. While we shall study in detail the encoding of Turing machines in the λ-calculus, we are not going to lay out the details of the simulation of the λ-calculus on RAMs. We shall provide an abstract machine for the λ-calculus and study its complexity using standard considerations for algorithmic analysis (which are grounded on RAMs), but without giving the details of the simulation. The RAM model we target has a read-only input register the space of which is not counted for space complexity, similarly to TMs. For the sake of completeness, we clarify the RAM cost models of reference: the logarithmic measure for time and Slot and Van Emde Boas's $\texttt{size}_b$ measure for space [40], counting 0 for unused registers and taking into account the logarithm of both the content and the index of used registers. Given a RAM R, we use $T_{\text{RAM}}(\rho)$ and $S_{\text{RAM}}(\rho)$ for the time and space used by R to reach a final configuration with a sequence of transitions ρ.

Reasonable Cost Models. We give the simulations and the bounds we shall consider for reasonable cost models for the λ-calculus.

Definition 2.1 (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:

- there is an encoding $\overline{\cdot}$ of binary strings and Turing machines into the λ-calculus such that if the run σ of a Turing machine M on input i ends on a state q_b with $b \in \mathbb{B}$, then there is a complete sequence $\rho : \overline{M}\,\overline{i} \to^* \overline{b}$ such that $T_\lambda(\rho) = O(poly(T_{\text{TM}}(\sigma), |i|))$ (resp. space $S_\lambda(\rho) = O(S_{\text{TM}}(\sigma) + \log |i|)$).
- There is an encoding $\underline{\cdot}$ of λ-terms into binary strings, a RAM R, and a decoding $\cdot\!\downarrow$ of final configurations for R such that if $\rho : t \to^* u$ is a complete sequence then the execution σ of R on input $\underline{t}$ produces a final configuration C such that $C\!\downarrow = u$ in time $T_{\text{RAM}}(\sigma) = O(poly(T_\lambda(\rho), |t|))$ (resp. space $S_{\text{RAM}}(\sigma) = O(S_\lambda(\rho) + \log |t|)$).

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.

3 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:

- *Compilation.* $\mathsf{init} : \Lambda \to Q$, turning λ-terms into states;
- *Decoding.* $\cdot\downarrow\colon Q \to \Lambda$, 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* $\rho : q \to^* q'$ 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. 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 3.1 (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 3.2 (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.

4 THE NAIVE KAM

The Krivine abstract machine [30] is a standard environment-based machine for Closed CbN, often defined as in Fig. 1. We refer to it as to *the Naive KAM*, to distinguish it from forthcoming variants. The machine evaluates closed λ-terms to weak head normal form via three transitions, the union of which is noted $\to_{\mathrm{NaKAM}}$:

- $\to_{\mathsf{sea}}$ 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 data structures used by the Naive KAM are *local environments*, *closures*, and a *stack*. *Local environments*, that 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 Naive 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 Naive KAM are defined as $\mathsf{init}(t_0) := (t_0, \epsilon, \epsilon)$ (where

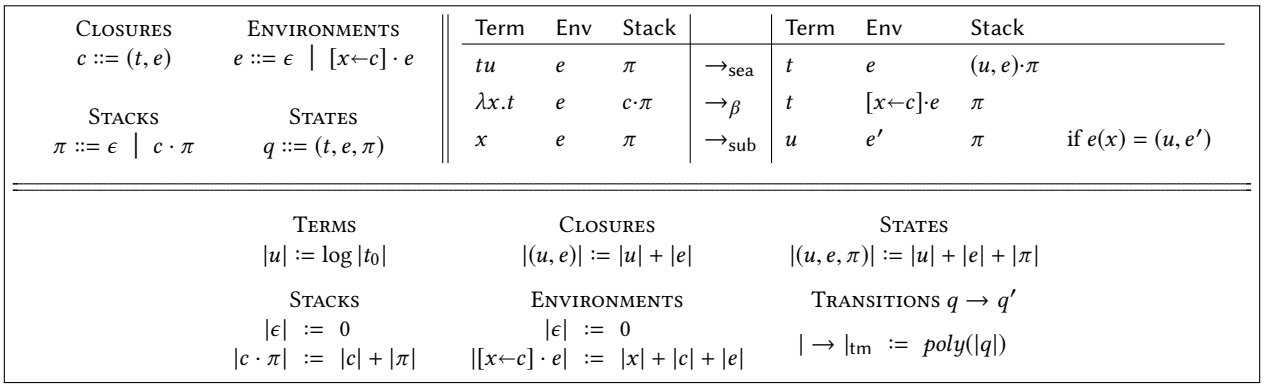

Figure 1: Data structures, transitions and abstract implementation of the Naive KAM.

t_0 is a closed λ-term, and also the code). The decoding of closures and states is as follows:

$$\text{Clos.} \quad (t,\epsilon)\downarrow := t \qquad (t,[x{\leftarrow}c]{\cdot}e)\downarrow := (t\{x{\leftarrow}c\downarrow\},e)\downarrow$$
$$\text{States} \quad (t,e,\epsilon)\downarrow := (t,e)\downarrow \qquad (t,e,\pi{\cdot}c)\downarrow := (t,e,\pi)\downarrow\ c\downarrow$$

Basic Qualitative Properties. Some standard facts about the Naive KAM follow. Let $\rho : \text{init}(t_0) \rightarrow_{\text{NaKAM}}^{*} q$ 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 $\rightarrow_{\text{sub}}$ transition because the environment does not contain an entry for the active variable. Final states then have shape $(\lambda x.u, e, \epsilon)$.

Theorem 4.1 (Implementation). *The Naive KAM implements Closed CbN, that is, there is a complete $\rightarrow_{wh}$-sequence $t \rightarrow_{wh}^{n} u$ if and only if there is a complete run $\rho : \text{init}(t) \rightarrow_{\text{NaKAM}}^{*} q$ 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 $\rightarrow_{wh}$ steps and $\rightarrow_{\beta}$ transitions, so that we can identify the two. Moreover, the number of $\rightarrow_{wh}$ steps is a reasonable cost model for time, as first proved by Sands et al. [35].

Quantitative Properties. We recall also some less known *quantitative* facts, for runs as above, from papers by Accattoli and co-authors [2, 3, 5]. 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 $|\rho|_{\text{sub}}$ of sub transitions in ρ is bounded by $O(|\rho|_{\beta}^{2})$, and there are terms on which the bound is tight. The number $|\rho|_{\text{sea}}$ of sea transitions is bounded by $O(|\rho|_{\beta}^{2} \cdot |t_0|)$, but on complete runs the bound improves to $O(|\rho|_{\beta})$.
- *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 $O(|\rho|_{\beta}^{2} \cdot |t_0|)$.

Sub-Term Pointers and Data Pointers. The Naive KAM is usually implemented using *two* forms of pointers:

1. *Sub-term pointers*: the initial term t_0 provides the initial immutable code. The essential sub-term invariant mentioned above allows us to represent the terms u in every closure (u, e) of every reachable state with a pointer to t_0 instead that with a copy of u.
2. *Data (structure) pointers*: to ensure that the duplication of the environment e in transition $\rightarrow_{\text{sea}}$ 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.

Both kinds of pointer shall draw our attention. Sub-term pointers have size $O(\log |t_0|)$. For the present discussion they are *space-friendly*, because their size does not depend on the length of the run. We shall inspect them in Sect. 6, where we shall ensure that also their number is under control, that is, independent of 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, *time*. Data pointers have thus size $O(\log |\rho|_{\beta})$, entangling space with time, which is unreasonable for space. Since data pointers are hostile, we want them to be explicitly accounted for by the specification of the machine. Therefore, we consider the Naive KAM as being implemented with sub-term pointers but *not* data pointers. This fact is expressed by the abstract implementation of the Naive KAM.

Abstract Implementation of the Naive KAM. Implementing the Naive KAM without data pointers means that environments cannot be implemented as linked lists, and the same is true for the stack, of which length also depends on the length of the run. The idea then is that they are implemented as unstructured *strings*, in a linear

Term	Env	Stack		Term	Env	Stack	
tx	e	π	$\to_{\mathsf{sea_v}}$	t	$e\|_t$	$e(x)\cdot\pi$	
tu	e	π	$\to_{\mathsf{sea_{\neg v}}}$	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 2: Transitions of the Space KAM.

syntax. We abstract from the actual encoding, what we retain is the abstract implementation in Fig. 1, which captures its essence.

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. 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 natural time (Prop. 7.4). 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. 6.1). A space-reasonable refinement of the Naive KAM is the topic of the next section.

5 THE SPACE KAM

Here we define a space optimization of the Naive KAM. The Space KAM is derived from the Naive KAM implementing two modifications aimed at space efficiency: namely *unchaining* and *eager garbage collection.*

Unchaining. It is a folklore optimization for abstract machines bringing speed-ups with respect to both time and space, used e.g. by Sands et al. [35], Wand [44], Friedman et al. [23], and Sestoft [39]. Its first systematic study is by Accattoli and Sacerdoti Coen in [12], 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. Beside the malicious chains connected to unchaining, the Naive 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 Space KAM. The Naive KAM optimized with both eager GC and unchaining (both optimizations are mandatory for space reasonability) is here called Space KAM and it is defined in Fig. 2. 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 mainly by transition $\to_{\beta_{\mathsf{w}}}$, which collects the argument if the variable of the β redex does not occur. Transitions $\to_{\mathsf{sea_v}}$ and $\to_{\mathsf{sea_{\neg v}}}$ also contribute to implement the GC, 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 5.1 (Environment domain invariant). *Let q be a Space 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

The abstract implementation of the Space KAM is the same of the Naive KAM, as the GC 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.

6 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 [17]. The first key step is understanding how to scroll Scott strings.

Encoding alphabets. Let $\Sigma = \{a_1, \ldots, a_n\}$ 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:

$$\lceil a_i \rceil^\Sigma := \lambda x_1.\ldots.\lambda x_n.x_i \ .$$

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 $\overline{s}^{\Sigma^*}$, defined by induction on s as follows:

$$\overline{\varepsilon}^{\Sigma^*} := \lambda x_1.\ldots.\lambda x_n.\lambda y.y \ , \quad \overline{a_i r}^{\Sigma^*} := \lambda x_1.\ldots.\lambda x_n.\lambda y.x_i \overline{r}^{\Sigma^*} .$$

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 announced in Sect. 2, we now explain how to obtain that the code size $\|\overline{s}^{\Sigma^*}\|$ and the constructor size $|\overline{s}^{\Sigma^*}|$ of the encoding of an input string have the same space cost, that is, how to make the size of variable pointers irrelevant for space (for the Scott encoding of strings). 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 $|\Sigma|$, that is, of *constant size*. Note that, similarly, if variables are represented with textual names, again having only $|\Sigma|$ 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 violated. Remarkably, a notable folklore property of the (Space) KAM is that its implementation theorem does not need Barendregt's convention to hold.

In the result about reasonable space, Forster et al. [22] also rely on the Scott encoding of strings and they represent λ-terms using de Bruijn indices. Although they do not stress it, for the reason that we have just explained, their choice of de Bruijn is crucial for their result to hold.

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.

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

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, first character is 0, 1, or the string is empty) 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.

Proposition 6.1. *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 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 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 $\mathtt{loCpy}$ realizing it uses the same fix-point schema as before, but with different, more involved action terms A_0, A_1, and A_ε.

Proposition 6.2. *Let $s \in \mathbb{B}^*$.*

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

The $\Theta(|s|\log|s|)$ 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 Separate Address Spaces. For a space-reasonable managing of the work tape, the local algorithm should rather work in space $O(|s|)$. This improvement can be realized by a finer complexity analysis. In Prop. 6.2.2, the cost comes from the use of $O(|s|)$

sub-term pointers to the code $\texttt{loCpy}\,\overline{s}^{\mathbb{B}}$ used by the Space KAM. These pointers have size $O(\log|s|)$ because $|\texttt{loCpy}\,\overline{s}^{\mathbb{B}}| = O(|s|)$, that is, the size of loCpy is independent of $|s|$ and thus constant. A close inspection of the Space KAM run in Prop. 6.2.2 shows that, of the used $O(|s|)$ pointers, only $O(1)$ of them actually point to $\overline{s}^{\mathbb{B}}$, while all the others (that is, an $O(|s|)$ amount) point to loCpy. Since loCpy is of size independent from $|s|$, if one admits separate address spaces for loCpy and $\overline{s}^{\mathbb{B}}$ then the pointers to loCpy have size $O(1)$. Therefore, one obtains that the space cost is given by

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

From now on then, the first argument of the code of the Space KAM—that in the encoding of TMs shall represent the input—has a dedicated address space.

Proposition 6.3 (Linear Space Local-Copy Scrolling). *Let $s \in \mathbb{B}^*$. The Space KAM evaluates $\texttt{loCpy}\,\overline{s}^{\mathbb{B}}$ in space $O(|s|)$ if loCpy and $\overline{s}^{\mathbb{B}}$ have separate space addresses.*

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 $O(\log|s|)$ bound, we need a more radical solution, which is possible because the tape is read-only (and thus 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 $\texttt{glCpy} := \lambda x.(\texttt{fix}\,(\lambda f.\lambda s'.s' f f x)x)$. Clearly, this approach breaks the almost affinity of the encoding, as copying is no longer encapsulated only in the fix-point.

Proposition 6.4. *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 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 [43]. 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 (and the reference encoding of TM). 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 $(s, |s_l|+1)$.

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 $O(\log|s|)$. The idea is to represent n as a binary string $\lfloor n \rfloor$. Since $n \le |s|$, we have $||\lfloor n \rfloor|| \le \log|s|$. 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 $O(\log|s|)$ space. Then one shows that in the λ-calculus the following crucial operation is doable in space $O(\log|s|)$: given a tape (s, n), returning (s, n) plus the n-th character s_n of s, which is realized by:

- 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.

7 THE SPACE KAM IS REASONABLE FOR SPACE

We are ready for our main result, which is based on a new variant (in the technical report [10]) over Dal Lago and Accattoli encoding of TMs into the λ-calculus [17]. 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 definition is in the technical report [10]).
- *CPS and indifference*: following [17], the encoding is in *continuation-passing style*, and carefully designed (by adding some η-expansions) as to fall into the *deterministic λ-calculus* $\Lambda_{\texttt{det}}$, 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 Sect. 9.
- *Duplication*: duplication is isolated in the unfolding of fix-points and in the managing of the input tape, all other operations are affine.

Theorem 7.1 (TMs are simulated by the Space KAM in reasonable space). *There is an encoding $\overline{\cdot}$ of log-sensitive TMs into $\Lambda_{\texttt{det}}$ 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 $O(S_{TM}(\rho)+\log|i|)$ if $\overline{M}$ and $\overline{i}$ have separate address spaces.*

The previous theorem provides the subtle and important half of the space reasonability result. The first two points establish the qualitative part of the simulation in the λ-calculus, together with the time bound (with respect to the number of β steps). The third point provides the space result for the Space KAM. They are connected by the fact that the Space KAM implements closed call-by-name (that coincides with $\to_{det}$ in $\Lambda_{\texttt{det}}$), via an omitted minor variant of Theorem 4.1.

Closures	Environments	Stacks	Heaps	States
$c ::= (t, e)$	e	π	$h ::= \epsilon \mid \{\pi := c{\cdot}\pi'\} \uplus h \mid \{e := [x{\leftarrow}c]{\cdot}e'\} \uplus h$	$q ::= (t, e, \pi, h)$

Term	Env	Stack	Heap		Term	Env	Stack	Heap	
tx	e	π	h	$\to_{\mathsf{sea_v}}$	t	e	π'	$\pi' := e(x){\cdot}\pi \uplus h$	π' fresh
tu	e	π	h	$\to_{\mathsf{sea}_\neg\mathsf{v}}$	t	e	π'	$\pi' := (u, e){\cdot}\pi \uplus h$	π' fresh, $u \notin \mathcal{V}$
$\lambda x.t$	e	π	$\{\pi := c{\cdot}\pi'\} \uplus h$	$\to_\beta$	t	e'	π'	$\{e' := [x{\leftarrow}c]{\cdot}e\} \uplus h$	e' fresh
x	e	π	h	$\to_{\mathsf{sub}}$	u	e'	π	h	if $e(x) = (u, e')$

Terms	Closures	States
$\lvert u\rvert := \log \lvert t_0\rvert$	$\lvert(u, e)\rvert := \lvert u\rvert + \lvert e\rvert$	$\lvert(u, e, \pi, h)\rvert := \lvert u\rvert + \lvert e\rvert + \lvert\pi\rvert + \lvert h\rvert$
Stacks	**Heaps**	**Transitions** $q \to q'$
$\lvert\pi\rvert := \log \lvert\rho\rvert_\beta$	$\lvert\epsilon\rvert := 0$	
Environments	$\lvert\{\pi{:=}c{\cdot}\pi'\} \uplus h\rvert := \lvert c\rvert + \lvert\pi'\rvert + \lvert h\rvert$	$\lvert \to \rvert_{\mathsf{tm}} := \log(\lvert\rho\rvert_\beta \cdot \lvert t_0\rvert)$
$\lvert e\rvert := \log \lvert\rho\rvert_\beta$	$\lvert\{e{:=}[x{\leftarrow}c]{\cdot}e'\} \uplus h\rvert := \lvert x\rvert + \lvert c\rvert + \lvert e'\rvert + \lvert h\rvert$	

Figure 3: Data structures, transitions and abstract implementation of the Time KAM.

The other half of the result amounts to showing that the Space KAM can be simulated on RAMs within the space costs claimed in Sect. 5. The idea is that it can clearly be simulated reasonably by a multi-tape TM using one work tape for the active term, one for the environment, and one for the stack, which then can be reasonably simulated by a RAM. We use RAM rather TM in the statement for uniformity with the works on time (this is relevant for the discussions in Sect. 8).

Theorem 7.2 (Space KAM is simulated by RAMs in reasonable space). *Let t be a λ-term. Every Space KAM run $\rho : \mathsf{init}(t) \to_{\mathrm{SpKAM}}{}^* q$ is implemented on RAMs in space $O(\lvert\rho\rvert_{\mathsf{sp}})$.*

From Theorems 7.1 and 7.2 follows our main result.

Theorem 7.3 (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.*

The Space KAM is not Reasonable for Natural Time. We complete our study of the Space KAM by analyzing its time behavior. For natural 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, which is the key point in the proof of the next proposition.

Proposition 7.4 (Space KAM natural time overhead explosion). *There is a family $\{t_n\}_{n\in\mathbb{N}}$ of closed λ-terms such that its complete evaluation $\rho_n : t_n \to_{wh}^n u_n$ is simulated by Space KAM runs σ_n taking both space and time exponential in n, that is, $\lvert\sigma_n\rvert_{\mathsf{sp}} = \lvert\sigma_n\rvert_{\mathsf{tm}} = \Omega(2^n)$.*

8 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. 7.4 never happen on λ-terms encoding TMs.

Theorem 8.1 (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 $O(poly(\lvert\rho\rvert))$.*
2. *Every Space KAM run $\rho : \mathsf{init}(t) \to_{\mathrm{SpKAM}}{}^* q$ can be implemented on RAMs in time $O(\lvert\rho\rvert_{\mathsf{tm}})$.*
3. *Closed CbN and the time of the Space KAM provide a reasonable time cost model for the λ-calculus.*

The drawback of this solution is that one gives up the natural cost model for time. Moreover, the low-level time cost model can be very lax in comparison, as Prop. 7.4 shows.

The Time KAM. The Naive KAM can also be optimized for *time*, rather than space, by adopting data pointers together with unchaining (that for time is not mandatory for reasonability but improves the overhead). The definition and the abstract implementation of this machine, named *Time KAM*, is in Fig. 3. Environments and stacks are now implemented as linked lists[1], enabling a sharing mechanisms that allows transitions to be implementable in constant time (actually logarithmic in the size of the code and of $\lvert\rho\rvert_\beta$, if pointers sizes are accounted for). The drawback is that space cannot be collected anymore, since there is *aliasing* i.e. the same memory cell could be referenced more than once. The meta-variables e and π are now meant to represent pointers to the *heap*, which is a memory containing the actual data. Making pointers and the heap explicit,

[1] Actually, we use one of the smarter implementations by Accattoli and Barras [3], namely *balanced trees* or *random access lists*.

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

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 4: The Space LAM.

some subtleties of the KAM become evident. Stack pointers are used linearly, as they are added to the heap in transition sea and removed in transition β. Environment pointers instead are non-linear: they are created by $\to_\beta$, duplicated by $\to_{\mathsf{sea}_{\neg\mathsf{v}}}$, but never removed. In particular, in transition $\to_{\mathsf{sub}}$ no memory gets freed, in contrast to what happens in the Naive/Space KAM.

The specification of the Time KAM allows a more precise account of the cost of implementing the KAM with data pointers, which is obtained starting from the abstract implementation in Fig. 3. The time cost of transitions is given for transitions belonging to complete runs, which satisfy better bounds (see *quantitative properties* above) leading to better estimates of the size of data pointers.

Theorem 8.2. *Let $\rho : \mathsf{init}(t_0) \to_{\mathrm{TiKAM}}{}^* q$ be a complete Time KAM run. It can be implemented on RAMs in time and space $O(|\rho|_\beta \cdot \log(|\rho|_\beta \cdot |t_0|))$.*

The theorem states that, considering as time the number of β steps/transitions, the Time KAM is reasonable for time, whence its name. It is instead unreasonable for space, since its space consumption (quasi)linearly depends from time.

The Interleaving Technique. Forster et al. in [22] show that, given one machine that is reasonable for time but not for space and one machine that is reasonable for space but not for time, it is possible to build a third machine that is reasonable for both space and time, 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 (the two starting machines being the Time KAM and the Space KAM), under the assumption that the two machines share the same input, which is essential for logarithmic space[2]. The drawback of this solution is that it admits space exponential in time, as Prop. 7.4 shows.

Trading Time for Space. From a practical rather than theoretical point of view, there is a further *semi solution* that we now sketch. Adding to the Time KAM (which already has environment unchaining) an eager (reference-counting) GC (thus having both space optimizations at work in the Space KAM in the Time KAM) one obtains a Shared Space KAM which is reasonable for time and *slightly unreasonable* for space. One can observe that the Shared Space KAM and the Space KAM are indeed strongly bisimilar and use the same number of closures. Then, the number of data pointers used by the Shared Space KAM is no longer entangled with the number of β-steps (as for the Time KAM), it is instead related to the Space KAM space cost—this is the effect of GC plus unchaining. Therefore, data pointers add a space overhead that is logarithmic in the *space* of the Space KAM. Also the GC mechanism, based on reference counting, adds the same logarithmic factor, while certainly costs time, though still remaining polynomial. This way, if the Space KAM uses $O(f(n))$ space, then the Shared Space KAM operates in $O(f(n) \log f(n))$ space. Such an overhead is not reasonable but not too unreasonable, and probably the best compromise between reasonability and efficiency for the practice of implementing functional programs.

9 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 as to *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.

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[3] CbV analogue of the KAM defined by Accattoli et al. in [2] and modeled after Leroy's ZINC [31] (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 Space LAM in Fig. 4 by removing data pointers

[2]The results in Forster et al. [22] 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 [22]).

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

	Natural Time Reasonable (cost model = num. of trans. = $\|\beta\|$)	**Low-Level Time Reasonable** (actual implementation cost)	**Space Reasonable** (low-level by def.)
Naive KAM	No, Proposition 7.4	No, Proposition 6.1.2	No, Proposition 6.1.2
Space KAM	No, Proposition 7.4	Yes, Theorem 8.1	Yes, Theorem 7.3
Time KAM	Yes, Theorem 8.2	Yes, Theorem 8.2	No, Theorem 8.2
Space LAM (CbV)	No, via Prop. 9.1 and Prop. 7.4	Yes, via Prop. 9.1 and Thm. 8.1	Yes, Theorem 9.2

Figure 5: Summary of the results of the paper.

and adding eager GC. Unchaining comes for free in CbV, if one considers values to be only abstractions, see Accattoli and Sacerdoti Coen [12].

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

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

Theorem 9.2 (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 Time 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.

10 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. 5.

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 work space has the same relationship to time than natural space, 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. On top of that, in a companion paper [9], we have given a more abstract, machine independent, characterization, based on multi types.

ACKNOWLEDGMENTS

This work has been inspired by an old talk by Kazushige Terui on the space efficiency of the KAM [42]. The second author is partially supported by the ERC CoG "DIAPASoN" (GA 818616).

REFERENCES

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

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

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

[4] Beniamino Accattoli, Andrea Condoluci, and Claudio Sacerdoti Coen. 2021. 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*. IEEE, 1–14. https://doi.org/10.1109/LICS52264.2021.9470630

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

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

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

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

[9] Beniamino Accattoli, Ugo Dal Lago, and Gabriele Vanoni. 2022. Multi Types and Reasonable Space. (2022). Accepted at ICFP 2022.

[10] Beniamino Accattoli, Ugo Dal Lago, and Gabriele Vanoni. 2022. Reasonable Space for the λ-Calculus, Logarithmically. (2022). https://doi.org/10.48550/arXiv.2203.00362

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

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

[13] Malgorzata Biernacka, Witold Charatonik, and Tomasz Drab. 2021. A Derived Reasonable Abstract Machine for Strong Call by Value. In *PPDP 2021: 23rd International Symposium on Principles and Practice of Declarative Programming, Tallinn, Estonia, September 6-8, 2021*, Niccolò Veltri, Nick Benton, and Silvia Ghilezan (Eds.). ACM, 6:1–6:14. https://doi.org/10.1145/3479394.3479401

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

[15] Guy E. Blelloch and John Greiner. 1996. A Provable Time and Space Efficient Implementation of NESL. In *Proceedings of the 1996 ACM SIGPLAN International*

Conference on Functional Programming, ICFP 1996, Philadelphia, Pennsylvania, USA, May 24-26, 1996, Robert Harper and Richard L. Wexelblat (Eds.). ACM, 213–225. https://doi.org/10.1145/232627.232650

[16] Andrea Condoluci, Beniamino Accattoli, and Claudio Sacerdoti Coen. 2019. Sharing Equality is Linear. In *Proceedings of the 21st International Symposium on Principles and Practice of Programming Languages, PPDP 2019, Porto, Portugal, October 7-9, 2019*, Ekaterina Komendantskaya (Ed.). ACM, 9:1–9:14. https://doi.org/10.1145/3354166.3354174

[17] Ugo Dal Lago and Beniamino Accattoli. 2017. Encoding Turing Machines into the Deterministic Lambda-Calculus. *CoRR* abs/1711.10078 (2017). arXiv:1711.10078 http://arxiv.org/abs/1711.10078

[18] Ugo Dal Lago and Ulrich Schöpp. 2010. Functional Programming in Sublinear Space. In *19th European Symposium on Programming, ESOP 2010, Paphos, Cyprus, March 20-28, 2010, Proceedings. (Lecture Notes in Computer Science, Vol. 6012)*, Andrew D. Gordon (Ed.). Springer, 205–225. https://doi.org/10.1007/978-3-642-11957-6_12

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

[20] Vincent Danos and Laurent Regnier. 1995. Proof-Nets and the Hilbert Space. In *Proceedings of the Workshop on Advances in Linear Logic*. Cambridge University Press, USA, 307–328. https://doi.org/10.1017/CBO9780511629150.016

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

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

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

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

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

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

[27] Neil D. Jones. 1999. LOGSPACE and PTIME Characterized by Programming Languages. *Theor. Comput. Sci.* 228, 1-2 (1999), 151–174. https://doi.org/10.1016/S0304-3975(98)00357-0

[28] Richard E. Jones, Antony L. Hosking, and J. Eliot B. Moss. 2011. *The Garbage Collection Handbook: The art of automatic memory management*. CRC Press. http://gchandbook.org/

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

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

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

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

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

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

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

[36] Patrick M. Sansom and Simon L. Peyton Jones. 1995. Time and Space Profiling for Non-Strict Higher-Order Functional Languages. In *Conference Record of POPL'95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Francisco, California, USA, January 23-25, 1995*, Ron K. Cytron and Peter Lee (Eds.). ACM Press, 355–366. https://doi.org/10.1145/199448.199531

[37] Ulrich Schöpp. 2006. Space-Efficient Computation by Interaction. In *Computer Science Logic, 20th International Workshop, CSL 2006, 15th Annual Conference of the EACSL, Szeged, Hungary, September 25-29, 2006, Proceedings (Lecture Notes in Computer Science, Vol. 4207)*, Zoltán Ésik (Ed.). Springer, 606–621. https://doi.org/10.1007/11874683_40

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

[39] Peter Sestoft. 1997. Deriving a Lazy Abstract Machine. *J. Funct. Program.* 7, 3 (1997), 231–264. http://journals.cambridge.org/action/displayAbstract?aid=44087

[40] Cees F. Slot and Peter van Emde Boas. 1988. The Problem of Space Invariance for Sequential Machines. *Inf. Comput.* 77, 2 (1988), 93–122. https://doi.org/10.1016/0890-5401(88)90052-1

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

[42] Kazushige Terui. 2008. On space efficiency of Krivine's abstract machine and Hyland-Ong games. https://www.kurims.kyoto-u.ac.jp/~terui/space2.pdf. Accessed: 2022-05-31.

[43] Peter van Emde Boas. 2012. Turing Machines for Dummies - Why Representations Do Matter. In *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 (Lecture Notes in Computer Science, Vol. 7147)*, Mária Bieliková, Gerhard Friedrich, Georg Gottlob, Stefan Katzenbeisser, and György Turán (Eds.). Springer, 14–30. https://doi.org/10.1007/978-3-642-27660-6_2

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