A Machine-Independent, Log-Sensitive Space-Cost Measure for the Weak Lambda-Calculus

Thibaut Balabonski
Université Paris-Saclay, CNRS, Laboratoire Méthodes Formelles, 91190 Gif-sur-Yvette, France

Abstract

We propose a simple space-cost measure for the λ-calculus, that extends the natural model measuring the size of the terms by also taking into consideration their origin. This new model is able to capture sublinear space complexity and we prove that, in the context of weak reduction, it is reasonable with respect to standard complexity theory. Precisely, the weak λ-calculus and Turing machines can simulate each other with a constant-factor space overhead, for any computation of logarithmic or higher space complexity. This means that the weak λ-calculus equipped with our cost model gives a proper characterization of the classical space complexity classes, including LOGSPACE and PSPACE.

2012 ACM Subject Classification Theory of computation → Lambda calculus; Theory of computation → Complexity classes

Keywords and phrases Lambda-calculus, Complexity, Space, Logspace, Weak reduction

Digital Object Identifier 10.4230/LIPIcs.LICS.2026.9

Related Version *Full Version*: `https://hal.science/hal-05322360` [8]

1 Introduction

The λ-calculus is born as a model of computability in the 1930s [12], together with Turing machines and equivalent to them in terms of expressive power [34, 33]. These two models are deeply different, with complementary strengths: Turing machines focus on elementary operations on *low-level* representation of data, while the λ-calculus emphasizes higher-level and *compositional* computation mechanisms. Their difference is a strength of computability theory: it multiplies the available points of view and proof techniques.

However, when computability theory started being refined into complexity theory in the 1960s/70s, the λ-calculus disappeared from the picture. Foundational papers [13, 19] as well as reference textbooks [20, 28, 6] mostly use Turing or RAM machines, and derivatives thereof. In a sense, increased preciseness came at the cost of a narrower range of available tools[1].

An important contribution of complexity theory is the characterization of classes of algorithmic problems, that can be solved with constrained time or memory resources, like PTIME (problems solvable in a polynomial number of elementary steps with respect to the size of the input) or LOGSPACE (problems solvable with memory space logarithmic in the size of the input). A key concept that makes these classes robust and theoretically meaningful is Slot and van Emde Boas' notion of *reasonable* computation models [30], described as "machines that can simulate each other within a polynomially-bounded overhead in time and a constant-factor overhead in space". This reasonability criterion is what ensures that complexity classes are universal, and not tied to a particular computational device. So if we want to reintroduce the λ-calculus in the picture, make it an appropriate tool for expressing complexity classes, and allow complexity theory to benefit from its specific properties and techniques, an essential step is to connect the λ-calculus with this reasonability criterion.

[1] This divorce between two pillars of theoretical computer science is even reflected in the thematic classification of theoretical computer science into "track A" and "track B" used by EATCS since its creation in the 1970s, or by the influential Handbook of Theoretical Computer Science [35, 36]: each of complexity theory and λ-calculus lives on its own side.

Is the λ-calculus a reasonable model? There are straightforward ways of expressing the complexity of a computation in the λ-calculus. First, β-reduction defines a notion of computation step which makes a natural unit of time. Second, the size of a λ-term gives a convenient basis for expressing space, dubbed "ink space" by reference to the quantity of ink used for writing down a term. The question is whether these natural measures of space and time, or variants thereof, can be related to known reasonable measures like Turing complexity.

This is not straightforward [17], and some facts even make it counter-intuitive. First, β-reduction is a non-atomic computation operation expressed using a subtle capture-avoiding term substitution operation. Can it really be an appropriate "unit"? Or do we need an implementation model before saying anything about complexity? Even more disturbing, there are simple examples of computations in the λ-calculus that build a term of exponential size in a linear number of steps [1], which is utterly impossible in a Turing-like model and suggest a tension between time-reasonability and space-reasonability in the λ-calculus.

Yet the model is now known to be reasonable. First positive results about the adequacy of the natural cost measures for the λ-calculus came in the 1990s, for time [11] as well as ink space [26], and have been recently improved. Concerning time, the number of β-reductions has first been proved to be an adequate time complexity measure in the context of weak reduction strategies: strategies that never reduce inside λ-abstractions, and correspond to usual programming language semantics [11, 29, 24]. This has been extended in the 2010s to strong strategies allowing full normalization [2]. The puzzling problem of having a single implementation model reasonable for both space and time has been solved for the first time only in 2020 for the weak call-by-value λ-calculus [16].

Breaking the linear space barrier. The "ink space" cost model, which defines space cost as the maximal size of the terms in a reduction sequence, is simple and natural but has an intrinsic limitation: since the given input is a part of the initial term being reduced, the measured space cost is always at least the length of the input. In other words, no sublinear space complexity can be measured this way: this model is adequate for characterizing PSPACE, but not LOGSPACE.

The same limitation naturally exists in Turing machines, since the input is initially written on the tape. The solution for Turing machines however is simple and well-known: just take a machine with a separate read-only tape for the input, and measure space complexity on the "main" tape, called work tape [31]. This model cleanly separates the program (the rules of the machine), the input, and the computation data that lives on the work tape.

In the λ-calculus this separation is not immediately apparent: a computation is represented by a sequence of terms, each of which mixes parts of the original program, parts of the input, and auxiliary computation data. A crucial step for characterizing sublinear space within the λ-calculus is thus finding a way to separate intermediate computation data from the program and the input.

This separation is natural in abstract machines for the evaluation of λ-terms [25, 22], which are implementation models in which the code is often not modified but accessed via pointers, and where dedicated components of the machine (typically an environment and a stack, although specifics may vary) contain all the auxiliary data. Indeed, a first solution precise enough to capture LOGSPACE emerged recently [5], which is based on a carefully crafted abstract machine. The drawback of this approach however is that the cost is measured on a specific implementation, and is not easily related to an accessible property of the underlying λ-terms. The authors mitigated this issue by proposing in a companion

paper a system of non-idempotent intersection types [3] precisely capturing the evaluation cost of their machine. However, such systems are quite complex, and even though this type system is expressed in the λ-calculus, it is still fundamentally linked to a specific machine.

Research question: is there a natural space-cost model for the λ-calculus, independent from any implementation architecture and able to capture sublinear space complexity?

Contribution. In this work, we propose a space-cost model for the weak λ-calculus which is (almost) as simple and natural as ink space, and yet precise enough to be log-sensitive: our contribution offers the first characterization of the LOGSPACE complexity class that is directly expressed in the λ-calculus independently of any implementation.

Intuitively, this space cost measure is ink space, except for some subterms that can be traced back to parts of the initial term. For those unaltered fragments, we only count the cost of locating the original subterm in the input.

Outline. We first define our space-cost measure in the pure λ-calculus, revisiting standard theory of residuals and descendance to characterize "unaltered" subterms (Section 2), then introduce a decorated variant of the λ-calculus which provides a simple way of tracking descendants of initial subterms and thus allows a simpler characterization of space complexity (Section 3). In a second part, we show that this decorated weak λ-calculus and Turing machines can simulate each other with a constant-factor overhead in space complexity (Sections 4 and 5). Alternatives are discussed at the end (Section 6). This version has been edited for space constraints. Detailed proofs are to be found in the full version [8].

2 A space-cost measure for the weak λ-calculus

We take the usual definition of the λ-calculus [10], with terms defined by the grammar below.

$$t \quad ::= \quad x \mid t\ t \mid \lambda x.t$$

Terms are considered modulo α-renaming of bound variables. We write $\mathsf{fv}(t)$ for the set of free variables of the term t. A term is *closed* if it has no free variable. The only computation rule is β-reduction.

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

It is based on a capture-avoiding substitution operation $t^{\{x \leftarrow u\}}$ that replaces every free occurrence of x in t by u. In *weak reduction*, this reduction rule can be applied anywhere except inside the body of a λ-abstraction $\lambda x.t$. We write $t \to_w t'$ when the term t reduces to the term t' in one weak step, and $t \to_w^* t'$ (resp. $t \to_w^n t'$) when t reduces to t' by a possibly empty sequence of successive weak steps (resp. a sequence of n steps).

2.1 Identifying and tracking subterms

The key ingredient in our cost measure is identifying the parts of a given reduced term that are fragments of the input and thus should not count as "work space" in our computation. For this we have to track "term parts", which we call *subterms*, along reduction sequences.

This idea of tracking a subterm of interest along reduction has long been part of the theoretical corpus of λ-calculus and rewriting theory. Tracked elements are particularly known under the name of *residuals* when we are concerned with what remains of a given redex after some reduction steps. Residuals are a classic tool for studying confluence properties of the λ-calculus [14, Chapter 4] [21] and are also at the root of optimal reduction theory [27, 18, 7].

▶ **Example 2.1** (Residuals). *Take the step* $t = (\lambda z.z)\ ((\lambda x.t)(\lambda y.y)) \to_w (\lambda z.z)\ t^{\{x \leftarrow \lambda y.y\}} = t'$. *The original term* t *contains two redexes: one defined by the application of the* λ*-abstraction* $\lambda x.t$*, which is reduced, and one defined by the application of the* λ*-abstraction* $\lambda z.z$*. The latter still appears in the reduced term* t' *under a different form: while its argument has been modified, the main structure of the redex remains present. This redex of* t' *is a* residual *of the* λz*-redex of* t*.*

Extending this notion of residual from redexes only to any subterm gives a *descendance* relation between subterms of an input term and subterms of a reduced term. While this extended notion also already exists [32], it is not of common use and is rarely formalized, so we include in this section a full definition.

First, remark that a given term may contain several occurrences of the "same" subterm, like $\lambda z.z$ appearing twice in t in Example 2.1. Although they are identical, these occurrences play different roles in the term and its reductions: we want to consider them as distinct subterms of t. For this, we formally characterize each subterm s in a term t by its *position* in the syntax tree of t, described as a sequence of directions from the root of the term.

▶ **Definition 2.2** (Sequences). *Let* Σ *be a set of symbols. A* sequence *on* Σ *is an ordered list* $S = a_0; a_1; \ldots; a_{n-1}$ *of elements of* Σ*, its length* $|S| = n$ *is the number of elements in* S*. We write ';' for the associative operator concatenating symbols and sequences of symbols. Its neutral element (the empty sequence) is written* ε*. A sequence* S_1 *is a* prefix *of a sequence* S_2 *if* S_2 *can be written as* $S_1; S_3$*, with* S_3 *possibly empty. We write* Σ^* *for the set of all sequences, and* Σ^n *for the set of sequences of length* n*.*

Remark that for any $a \in \Sigma$, the notation a can denote either the element itself, or a sequence of one element. The context usually makes clear which one is meant. In some contexts, Σ is also called alphabet, symbols $a \in \Sigma$ are called characters, and sequences are called strings.

▶ **Definition 2.3** (Positions of a term). *A* position *is a sequence on the set* $\{0, 1, 2\}$*, that describes a path in the syntax tree of a* λ*-term. The set* $\mathcal{P}(t)$ *of the positions of a* λ*-term* t *is defined as follows:*

$$\begin{array}{rcl} \mathcal{P}(x) & = & \{\varepsilon\} \\ \mathcal{P}(\lambda x.t) & = & \{\varepsilon\} \cup \{0; p \mid p \in \mathcal{P}(t)\} \\ \mathcal{P}(t\ u) & = & \{\varepsilon\} \cup \{1; p \mid p \in \mathcal{P}(t)\} \cup \{2; p \mid p \in \mathcal{P}(u)\} \end{array}$$

The position of a β-reduction *is the position of the reduced redex. A* weak position *is a position that does not contain* 0*.*

▶ **Definition 2.4** (Subterm at position p, replacement). *Assuming* $p \in \mathcal{P}(t)$*, the* subterm at position p *in* t*, written* $t|_p$*, and its* replacement *by a term* s*, written* $t[s]_p$*, are defined as follows.*

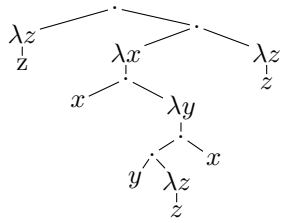
$$\begin{array}{rcl@{\qquad\qquad}rcl} t|_\varepsilon & = & t & t[s]_\varepsilon & = & s \\ (\lambda x.t)|_{0;p} & = & t|_p & (\lambda x.t)[s]_{0;p} & = & \lambda x.(t[s]_p) \\ (t\ u)|_{1;p} & = & t|_p & (t\ u)[s]_{1;p} & = & (t[s]_p)\ u \\ (t\ u)|_{2;p} & = & u|_p & (t\ u)[s]_{2;p} & = & t\ (u[s]_p) \end{array}$$

Contrarily to substitution, the replacement is not capture-avoiding.

As said above we identify the notion of subterm with that of position: even if $t|_{p_1}$ and $t|_{p_2}$ may be identical, they are by definition distinct subterms of t whenever $p_1 \neq p_2$.

▶ **Example 2.5** (Positions/subterms). *Let t be the λ-term given below, as a term on the left and as a syntax tree on the right, with dots '.' denoting the applications in the tree.*

$$(\lambda z.z)\ \Big((\lambda x.x(\lambda y.y(\lambda z.z)x))\ \ (\lambda z.z)\Big)$$



Its subterm $t|_{2102}$ at position 2102 is $\lambda y.y(\lambda z.z)x$. It also has three distinct $\lambda z.z$ subterms, at positions 1, 22, and 2102012.

The notion of *descendance* along a reduction step $t \to_w t'$ relates positions of t' to positions of t, and allows tracking subterms along reduction. Remark that a β-reduction $t \to_w t'$ at position p in t can be written $t \to_w t[u^{\{x \leftarrow s\}}]_p$, with $t|_p = (\lambda x.u)\ s$.

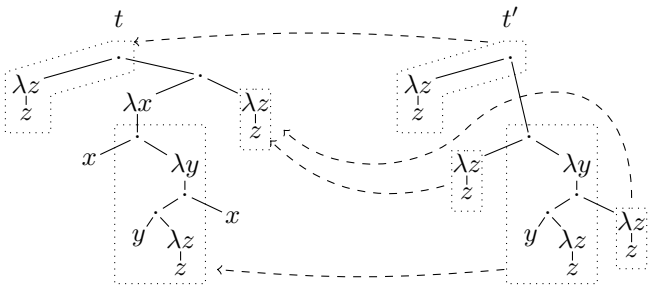
▶ **Definition 2.6** (Descendants). *Let $\rho : t \to_w t[u^{\{x \leftarrow s\}}]_p$, with $t|_p = (\lambda x.u)\ s$, be a weak β-reduction step at position p in t. Let $q \in \mathcal{P}(t)$ be a position in t. The set q/ρ of the* descendants *of q after ρ is the set of positions of $t[u^{\{x \leftarrow s\}}]_p$ defined as follows.*

- *If p is not a prefix of q, then $q/\rho = \{q\}$.*
- *If $q = p; 10; q'$ and $u|_{q'} \neq x$, then $q/\rho = \{p; q'\}$.*
- *If $q = p; 2; q'$, then $q/\rho = \{p; p_x; q' \mid p_x \in \mathcal{P}(u) \text{ such that } u|_{p_x} = x\}$.*
- *Otherwise, $q/\rho = \emptyset$.*

In the two latter conditions, using usual hygienic conditions on variable names [10] we assume x denotes an occurrence of the variable bound by the λ-abstraction of the redex. The position q is called ancestor *of the positions in q/ρ. The notion of descendance extends transitively to sequences of reduction.*

▶ **Definition 2.7** (Full descendants). *Given a reduction step $t \to_w t'$, and two positions p of t and p' of t', we say that the subterm s' at position p' is a* full descendant *of the subterm s at position p if for any position $p_s \in \mathcal{P}(s')$, the position $p'; p_s$ of t' descends from the corresponding position $p; p_s$ of t. The position p' is called* root *position of s', and s' is* maximal *if no prefix of p' is the root position of a full descendant. This notion extends transitively to reduction sequences. Remark that s' being a full descendant of s implies $s = s'$.*

▶ **Example 2.8** (Descendants). *Consider the reduction step ρ at position 2 in the term t of Example 2.5 : $t \to_w (\lambda z.z)\ \big((\lambda z.z)(\lambda y.y(\lambda z.z)(\lambda z.z))\big) = t'$. This reduction and the corresponding descendance relation can be pictured as follows, with arrows going from descendants to their ancestors, and dotted boxes around term parts that stay together.*



Positions ε, 1 and 10 of t, which correspond to the context part of the reduction, each have itself has unique descendant. Among them, we have full descendance for 1 and 10, but not for ε whose right member is modified. Position 210 (resp. 210201) of t, corresponding to the

subterm $x(\lambda y.y(\lambda z.z)x)$ (resp. $y(\lambda z.z)$) of the body of the reduced λ-abstraction, has a unique descendant 2 *(resp.* 201*). Position* 2201 *of t' is a full descendant of its ancestor in t, while position* 2 *of t' is not a full descendant of* 210*, since the corresponding subterm has been effectively modified by the substitution. Position* 22 *(resp.* 220*) of t, corresponding to the argument of the reduced redex, has two full descendants* 21 *and* 2202 *(resp.* 210 *and* 22020*), corresponding to the two copies of the argument. Two sets of positions of t do not have any descendant in t', since they disappear in the reduction. This concerns on the one hand the positions* 2 *and* 21 *that define the redex, and the positions* 2101 *and* 210202 *of the substituted variable occurrences. On the other hand, any position of t' descends from a position of t (the sets of descendants of all the positions of t after ρ are even a partition of the positions of t').*

2.2 Space cost

The space-cost measure we want to define relies on the notion of descendance. This means the space cost of a given intermediate state of computation depends on the reduction sequence that led from the initial state to this intermediate state. In other words, the space cost of a given term t is always relative to a reduction sequence $t_0 \to_w^* t$ from an initial term t_0. The cost measure relies on two constants c_v (cost of a variable) and c_p (cost of a pointer), which estimate a fixed cost for representing a variable and identifying a subterm of the initial λ-term t_0, respectively. These two values are deduced from the initial term t_0 and are fixed along any reduction sequence starting from t_0.

▶ **Definition 2.9** (Size and depth). *We define the* size $|t|$ *of a λ-term t as the number of positions in its syntax tree (equivalently, the number of constructors in the term).*

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

We call variable depth $\mathsf{vd}(t)$ *of a λ-term t the maximal number of binders between a variable occurrence and its own binder. Equivalently, $\mathsf{vd}(t)$ is the maximum index used in the de Bruijn notation for t [15].*

A distinctive property of weak reduction is that $t \to_w t'$ implies $\mathsf{vd}(t') \leqslant \mathsf{vd}(t)$, coming from the fact that the path between a variable occurrence and its binder is never modified.

▶ **Definition 2.10** (Space cost). *Let t_0 be a λ-term and $\rho : t_0 \to_w^* t$ a reduction. Define the* pointer cost c_p *and the* variable cost c_v*:*

$$\begin{cases} c_p \;=\; \mathsf{max}(1, \lceil \mathsf{log}(|t_0|) \rceil) \\ c_v \;=\; \mathsf{max}(1, \lceil \mathsf{log}(\mathsf{vd}(t_0)) \rceil) \quad \textit{or } c_v = 1 \textit{ in case } \mathsf{vd}(t_0) = 0 \end{cases}$$

The space cost *of the term t after the sequence ρ is the sum of the costs of the positions of t, where the cost of a position q is:*

- *if q is in a maximal full descendant of a closed subterm of t_0:*
  - *c_p for the root position of the descendant,*
  - *0 for any other position.*
- *if q is not in a maximal full descendant of a closed subterm of t_0:*
  - *c_v for the position of a variable occurrence,*
  - *1 otherwise.*

The space cost of t_0 after the empty reduction $t_0 \to_w^0 t_0$ is c_p. The space cost of a reduction sequence $t_0 \to_w t_1 \to_w \ldots \to_w t_n$ is the maximal space cost of a term t_i of this sequence after the prefix reduction $t_0 \to_w \ldots \to_w t_i$. Note that log *is the binary logarithm.*

▶ **Example 2.11** (Space cost). *Consider the reduction step $\rho : t \to_w t'$ of Example 2.8. The initial term t has size $|t| = 16$ and variable depth $\mathsf{vd}(t) = 1$ hence we fix a pointer cost $c_p = 4$ and a variable cost $c_v = 1$. The space cost of t' after the reduction ρ is $5 + c_v + 4c_p$, decomposed in the following way:*

- *the four occurrences of $\lambda z.z$ all are full descendants of closed subterms of t, each one contributes c_p to the total cost,*
- *positions ε, 2, 22, 220, not in a full descendant, each contribute 1,*
- *positions 2201 and 22011 are in a full descendant, but not in a* closed *full descendant, they contribute 1 and c_v, respectively.*

The main goal of the rest of the paper is to establish that this space-cost model, that we defined while staying in the traditional pure λ-calculus, is reasonable for any weak reduction sequence. However, we will start by proposing an alternative characterization of this cost model, which reduces the technically complex notion of descendant and its dependency on reduction history to a simple and history-free annotation mechanism.

3 Simple characterization of the cost measure with a marked calculus

In this section we introduce a decorated λ-calculus, called $\lambda_{\{\}}$, which automatically tracks full descendants of closed initial subterms while strictly preserving the reduction behaviour of the weak λ-calculus, and equip this decorated calculus with a space-cost measure provably equivalent to the measure of Section 2.

3.1 The $\lambda_{\{\}}$-calculus and its space-cost measure

Conceptually, our decorated $\lambda_{\{\}}$-calculus is the traditional pure λ-calculus, plus a boolean at each position indicating whether the corresponding subterm is *marked* or not, with the intent that marked subterms correspond to full descendants of closed subterms of an initial $\lambda_{\{\}}$-term. Said initial $\lambda_{\{\}}$-term and the exact reduction path to the current $\lambda_{\{\}}$-term are unknown and irrelevant: marks only identify subterms whose contents need not be taken into account, since they could be represented by an appropriate pointer. To convey this intuition in the syntax, we write braces around marked subterms, and keep traditional λ-calculus syntax otherwise.

▶ **Definition 3.1** ($\lambda_{\{\}}$-Terms). *The grammar of $\lambda_{\{\}}$-terms is*

$$\begin{array}{lcll} t & ::= & \theta \mid \{\theta\} & \lambda_{\{\}}\text{-terms} \\ \theta & ::= & x \mid t\ t \mid \lambda x.t & \lambda_{\{\}}\text{-preterms} \end{array}$$

A $\lambda_{\{\}}$-term of the form $\{\theta\}$ is said to be marked. *Free variables $\mathsf{fv}(t)$ of a $\lambda_{\{\}}$-term t are defined as usual, with the additional equation $\mathsf{fv}(\{\theta\}) = \mathsf{fv}(\theta)$ (meaning that marks do not interfere with the definition). A $\lambda_{\{\}}$-term is* closed *when it has no free variable.*

The two-level syntax makes explicit the fact that each position can have at most one mark. By the definition below, the marks themselves do not count as positions in the $\lambda_{\{\}}$-terms.

▶ **Definition 3.2** (Positions/subterms). *Positions and subterms are as in Definitions 2.3 and 2.4, with these additional equations:*

$$\mathcal{P}(\{\theta\}) = \mathcal{P}(\theta) \qquad \{\lambda x.t\}|_{0.p} = t|_p \qquad \{t\ u\}|_{1.p} = t|_p \qquad \{t\ u\}|_{2.p} = u|_p$$

As before, we identify subterms and positions. By this definition we have $\{\theta\}|_\varepsilon = \{\theta\}$. The $\lambda_{\{\}}$-preterm θ is not considered to be a subterm of $\{\theta\}$. Rather, $\{\theta\}$ is a variant of θ whose root constructor is marked.

▶ **Definition 3.3** (Full marking and unmarking). *The* full marking $\{\!|t|\!\}$ *of a λ-term t is the* $\lambda_{\{\}}$*-term obtained by marking every closed subterm of t. Such* $\lambda_{\{\}}$*-terms serve in particular as initial terms.*

$$\{\!|x|\!\} = x \qquad \{\!|t\ u|\!\} = \begin{cases} \{\{\!|t|\!\}\ \{\!|u|\!\}\} \text{ if } \mathsf{fv}(t\ u) = \emptyset \\ \{\!|t|\!\}\ \{\!|u|\!\} \quad \text{otherwise} \end{cases} \qquad \{\!|\lambda x.t|\!\} = \begin{cases} \{\lambda x.\{\!|t|\!\}\} \text{ if } \mathsf{fv}(\lambda x.t) = \emptyset \\ \lambda x.\{\!|t|\!\} \quad \text{otherwise} \end{cases}$$

A $\lambda_{\{\}}$*-term is* fully marked *if it can be obtained by fully marking a λ-term. The* unmarking t^* *of a* $\lambda_{\{\}}$*-term t is the λ-term obtained by removing all its marks. The operation naturally extends to preterms.*

$$\{\theta\}^* = \theta^* \qquad x^* = x \qquad (t\ u)^* = t^*\ u^* \qquad (\lambda x.t)^* = \lambda x.t^*$$

Remark that a $\lambda_{\{\}}$*-term and its unmarking have the same positions.*

The notion of well-formed $\lambda_{\{\}}$-term captures invariants of the terms t that are obtained after a sequence of reduction steps starting from a fully marked $\lambda_{\{\}}$-term $\{\!|t_0|\!\}$ (see Lemma 3.12 for the invariance statement).

▶ **Definition 3.4** (Well-formed $\lambda_{\{\}}$-terms). *A* $\lambda_{\{\}}$*-term t is* well-formed *when:*

- *every marked subterm of t is closed and fully marked,*
- *every closed subterm of t contains at least one mark.*

The first condition is an essential property of our marking. It reflects the fact that any subterm of a full descendant is also a full descendant. The second property is ancillary, and roughly means that no closed term can be built by weak reduction without reusing some parts of already existing closed terms (which helps in the proof of Lemma 5.17).

▶ **Definition 3.5** (Term substitution). *The substitution* $t^{\{x \leftarrow s\}}$ *of x by s in t is defined as usual, extended by the equation* $\{\theta\}^{\{x \leftarrow s\}} = \{\theta^{\{x \leftarrow s\}}\}$*. Remark that, if t is well-formed, any marked subterm* $\{\theta\}$ *is closed and thus satisfies* $\{\theta\}^{\{x \leftarrow s\}} = \{\theta\}$*.*

Reduction in the $\lambda_{\{\}}$-calculus behaves exactly as weak reduction on the underlying λ-terms. Note that whenever we trigger a redex that is under a mark, this mark is permanently removed as a side effect. Indeed, the term is modified by the reduction.

▶ **Definition 3.6** (Marked reduction). *One-step reduction* $t \multimap t'$ *in the* $\lambda_{\{\}}$*-calculus is defined by the following inference rules.*

$$\frac{}{(\lambda x.t)\ s \multimap t^{\{x \leftarrow s\}}} \qquad \frac{}{\{\lambda x.t\}\ s \multimap t^{\{x \leftarrow s\}}}$$

$$\frac{t \multimap t'}{t\ u \multimap t'\ u} \qquad \frac{u \multimap u'}{t\ u \multimap t\ u'} \qquad \frac{\theta \multimap t'}{\{\theta\} \multimap t'}$$

Descendants are defined as in the λ-calculus (Definition 2.6). A computation *is a reduction sequence* $\{\!|t_0|\!\} \multimap^* t$ *starting from a fully marked closed* $\lambda_{\{\}}$*-term.*

▶ **Example 3.7** (Marked reduction). *The full marking of the λ-term t of Example 2.5 is* $\{\!|t|\!\} = \Big\{\{\lambda z.z\}\ \Big\{\big\{\lambda x.x\,\big(\lambda y.y\{\lambda z.z\}x\big)\big\}\ \{\lambda z.z\}\Big\}\Big\}$*. From it we can derive a marked reduction step* $\{\!|t|\!\} \multimap \{\lambda z.z\}\ \big(\{\lambda z.z\}\,\big(\lambda y.y\{\lambda z.z\}\{\lambda z.z\}\big)\big) = t''$ *where the result* t'' *is a well-formed (but not full) marking of the λ-term* t' *such that* $t \to_w t'$ *seen in Example 2.8. Remark that two marks have been removed at prefix positions of the reduced redex.*

The space cost measure relies on the same two parameters c_v (cost of a variable) and c_p (pointer cost) as Definition 2.10, which are now relative to some unspecified initial $\lambda_{\{\}}$-term and fixed along any reduction sequence. Marked subterms identify the full descendants of the initial $\lambda_{\{\}}$-term, whose cost is c_p. As a consequence, the space cost can now be defined for an isolated $\lambda_{\{\}}$-term (whose origin is implicit) rather than only on an explicit full reduction sequence. Remark that, contrarily to Definition 2.10, this definition of space cost also seamlessly allows composing reduction sequences and their costs, which will be useful in the forthcoming proofs.

▶ **Definition 3.8** (Space cost). *Given two constants c_v and c_p, the* space cost $\|t\|_{c_v,c_p}$ *of a $\lambda_{\{\}}$-term t is defined by the equations:*

$$\begin{array}{rclcrcl} \|\{\theta\}\|_{c_v,c_p} & = & c_p & & \|t\ u\|_{c_v,c_p} & = & 1 + \|t\|_{c_v,c_p} + \|u\|_{c_v,c_p} \\ \|x\|_{c_v,c_p} & = & c_v & & \|\lambda x.t\|_{c_v,c_p} & = & 1 + \|t\|_{c_v,c_p} \end{array}$$

The space cost of a reduction sequence $\rho : t_0 \twoheadrightarrow t_1 \twoheadrightarrow \ldots \twoheadrightarrow t_n$ is $\|\rho\|_{c_v,c_p} = \mathsf{max}(\|t_i\|_{c_v,c_p})$. The space cost of a computation $\rho : \{\!|t_0|\!\} \twoheadrightarrow \ldots$, with t_0 a closed λ-term, is the space cost $\|\rho\|_{c_v,c_p}$ of the sequence with pointer cost and variable cost defined as

$$\begin{cases} c_p & = & \mathsf{max}(1, \lceil \mathsf{log}(|t_0|) \rceil) \\ c_v & = & \mathsf{max}(1, \lceil \mathsf{log}(\mathsf{vd}(t_0)) \rceil) \qquad \text{or } c_v = 1 \text{ if } \mathsf{vd}(t_0) = 0 \end{cases}$$

▶ **Example 3.9** (Space cost). *Consider the $\lambda_{\{\}}$-term $t'' = \{\lambda z.z\}\ \big(\{\lambda z.z\}(\lambda y.y\{\lambda z.z\}\{\lambda z.z\})\big)$ obtained as result in Example 3.7. Given two constants c_v and c_p, the space cost of t'' is $\|t''\|_{c_v,c_p} = 5 + c_v + 4c_p$.*

3.2 Adequacy with the weak λ-calculus

We first show that the $\lambda_{\{\}}$-calculus mirrors the weak λ-calculus, with marks properly tracking full descendance.

▶ **Lemma 3.10** (Soundness). *If $t \twoheadrightarrow t'$ in the $\lambda_{\{\}}$-calculus, then $t^* \to_w t'^*$ in the weak λ-calculus. Conversely, if $t^* \to_w u$ in the weak λ-calculus, then there is t' such that $t \twoheadrightarrow t'$ in the $\lambda_{\{\}}$-calculus and $t'^* = u$. Moreover, both reduction steps induce the same descendance relation on the positions of their source and target.*

▶ **Lemma 3.11** (Descendants). *In any reduction $\rho : t \twoheadrightarrow t'$ of the $\lambda_{\{\}}$-calculus, a subterm s' in t' is marked if and only if it is a full descendant of a marked subterm of t along the step ρ.*

▶ **Lemma 3.12** (Well-formedness preservation). *The full marking $\{\!|t_0|\!\}$ of any λ-term t_0 is well-formed. If a $\lambda_{\{\}}$-term t is well-formed and $t \twoheadrightarrow t'$, then t' is well-formed.*

We deduce that the space-cost measure defined for $\lambda_{\{\}}$ in this section is equivalent to the space-cost measure defined for the weak λ-calculus in Section 2.

▶ **Theorem 3.13** (Space cost soundness). *Let t be a closed λ-term. Let $\{\!|t|\!\} \twoheadrightarrow^* u$ be a computation in the $\lambda_{\{\}}$-calculus, and $\rho_w : t \to_w^* u^*$ be the corresponding computation in the weak λ-calculus. Let c_p and c_v be the cost constants given by the initial λ-term t. Then $\|u\|_{c_v,c_p}$ is equal to the space cost of u^* after reduction ρ_w.*

▶ **Corollary 3.14** (Space cost soundness). *Let t be a closed λ-term. The space cost of any computation $\{\!|t|\!\} \twoheadrightarrow^* u$ in $\lambda_{\{\}}$ is equal to the space cost of the corresponding computation $t \to_w^* u^*$ in the weak λ-calculus.*

As a consequence we may from now on reason on $\lambda_{\{\}}$, and the simulation results we establish in Sections 4 and 5 between $\lambda_{\{\}}$ and Turing machines will also be valid in the weak λ-calculus.

4 Simulation of Turing machines in $\lambda_{\{\}}$

In this section, we detail a space-preserving encoding of Turing machines in the $\lambda_{\{\}}$-calculus. Our encoding is an adaptation of the log-sensitive encoding by Accattoli, Dal Lago and Vanoni [23, 4], targeting the "deterministic λ-calculus": a restriction of the λ-calculus in CPS style ensuring that there is at most one weak redex in any term, and thus that *weak* reduction is deterministic.

Beyond the obvious required adaptation of explicitly handling marks and studying the associated space cost, it appeared also necessary to modify the original encoding of arithmetic operations. We replaced simple recursive algorithms by tail-recursive ones, to avoid an accumulation of pointers in the continuations of recursive calls. Beyond these two points, the encoding is similar to [4], as we only introduce some minor twists such as unnesting continuations and reordering variables to ensure that all intermediate terms are closed.

4.1 Turing machines

In order to be able to capture sublinear space complexity, the considered Turing machine model uses two tapes [31]:

- a read-only input tape, whose length and contents are fixed,
- a read-write work tape, whose characters can be overwritten and which may extend indefinitely in both directions.

▶ **Definition 4.1** (Index in a sequence). *A valid index in a sequence S over a set Σ is an integer i such that $0 \leqslant i < |S|$. When i is a valid index, we write $S[i]$ for the symbol at index i in S, the first index being 0 (if $S = a_0; a_1; \ldots; a_{n-1}$ with all $a_i \in \Sigma$, then $S[i] = a_i$). If i is not a valid index, then $S[i]$ is undefined, which we write $S[i] = \bot$.*

We write $S[a..b)$ for the subsequence of S ranging from index a (included) to index b (excluded), and $S[a..)$ as a shortcut for $S[a..|S|)$. The mirror of S, written $S^{\bowtie}$, is the sequence S in reverse order.

▶ **Definition 4.2** (Turing machines). *Let Σ be a finite alphabet, not containing the three reserved symbols $\diamond$ (blank), $\diamond_l$ (left end) and $\diamond_r$ (right end). Define $\Sigma_{\mathbb{I}} = \Sigma \cup \{\diamond_l, \diamond_r\}$ and $\Sigma_{\mathbb{W}} = \Sigma \cup \{\diamond\}$, the* input alphabet *and the* work alphabet *over Σ. Define $\Delta = \{\leftarrow, \downarrow, \rightarrow\}$. A Turing machine over Σ is a tuple $\mathcal{M} = (\mathbb{Q}, q_0, q_a, q_r, \delta)$ where*

- *$\mathbb{Q}$ is a set of states,*
- *$q_0 \in \mathbb{Q}$ is the initial state and $q_a, q_r \in \mathbb{Q}$ the final states,*
- *$\delta : ((\mathbb{Q} \setminus \{q_a, q_r\}) \times \Sigma_{\mathbb{I}} \times \Sigma_{\mathbb{W}}) \to (\Delta \times \Sigma_{\mathbb{W}} \times \Delta \times \mathbb{Q})$ is a total transition function.*

A binary machine is a machine over the binary alphabet $\mathbb{B} = \{0, 1\}$.

In a Turing machine configuration, each tape has its own head, determining the current read position (which is also written on the work tape). Following [4], we specify the input tape by a pair (I, n), where $I \in \Sigma_{\mathbb{I}}^*$ is the full contents of the tape, delimited by the special characters $\diamond_l$ and $\diamond_r$, and n is an index giving the position of the input head. The work tape is specified by a triple (W_l, b, W_r), where $b \in \Sigma_{\mathbb{W}}$ is the character pointed by the work head, and $W_l, W_r \in \Sigma_{\mathbb{W}}^*$ are two sequences describing the already visited parts of the work tape to the left and to the right of the head, read from the head to the appropriate end. This means the full tape contents is given by the sequence $W_l^{\bowtie}; b; W_r$. The work tape has initially only one cell, pointed by the work head and containing the blank symbol $\diamond$. Any position of the work tape is initialized with the blank symbol $\diamond$ when it is visited for the first time. No cell is ever deleted (but the contents of a cell can be overwritten with the blank symbol $\diamond$).

▶ **Definition 4.3** (Configurations). *A configuration is a tuple* $\langle I, n \mid W_l, b, W_r \mid q\rangle$ *giving the two tapes as well as the state* $q \in \mathbb{Q}$*. The initial configuration for an input* $I \in \Sigma^*$ *is* $\langle \diamond_l; I; \diamond_r, 0 \mid \varepsilon, \diamond, \varepsilon \mid q_0\rangle$*. A final configuration is any configuration* $\langle I, n \mid W_l, b, W_r \mid q\rangle$ *with* $q \in \{q_a, q_r\}$*. It is accepting if* $q = q_a$ *and rejecting if* $q = q_r$*.*

The transition function of a machine determines computation steps based on the current state and currently read characters. The function is written $\delta(q, a, b) = \langle d \mid b', m \mid q'\rangle$. In any non-final configuration it produces a move d on the input tape ($\downarrow$ for stay, $\leftarrow$ or $\rightarrow$ for moving left or right), a character b' written at the current position of the work tape, a move m on the work tape, and a target state q'. Following [31] we assume the transition function is such that the input head never exits the boundaries of the input tape (meaning that n is always a valid index in I).

▶ **Definition 4.4** (Transitions). *Let* $C = \langle I, n \mid W_l, b, W_r \mid q\rangle$ *be a configuration such that* $\delta(q, I[n], b)$ *is defined. We write* $C \longmapsto C'$ *if* C' *is the configuration obtained after applying this transition.*

▶ **Definition 4.5** (Space cost). *The space cost of a configuration* $C = \langle I, n \mid W_l, b, W_r \mid q\rangle$ *is the size* $|C| = 1 + |W_l| + |W_r|$ *of its work tape (including blank cells). The space cost of a computation* $C_0 \longmapsto C_1 \longmapsto \ldots \longmapsto C_n$ *is the maximal space cost of a configuration* C_i*.*

4.2 Configurations

The representation of Turing machine configurations uses the usual Scott encodings of characters, strings and tuples, detailed below. A configuration $\langle I, n \mid W_l, b, W_r \mid q\rangle$ is a tuple of the six elements. Our encoding makes marks explicit, with the following invariants:

- the representation of single characters and of the empty string are fully marked, as they are atoms in weak reduction,
- the representation of the read-only input string is fully marked, as it is present from the beginning and is never changed,
- any other string will be built from scratch and will evolve during computation, and as such will not contain any mark, except in the final empty string.

▶ **Definition 4.6** (Characters and strings). *Let* $\Sigma = \{a_1, \ldots, a_\sigma\}$ *be an alphabet of cardinal* σ*. The encoding* $[a]_\Sigma$ *of a character* $a \in \Sigma$ *is a* σ*-way choice function. The encoding* $[\![S]\!]_\Sigma$ *of a string* S *of length* n *is a* $(\sigma + 1)$*-way choice, which is either the empty string, or one character appended in front of a string of length* $(n - 1)$*.*

$$\begin{array}{rcll} [a_i]_\Sigma & = & \{\!|\lambda x_1 \ldots x_\sigma . x_i|\!\} & = \{\lambda x_1 . \ldots \{\lambda x_i \ldots x_\sigma . x_i\} \ldots\} \\ [\![\varepsilon]\!]_\Sigma & = & \{\!|\lambda x_1 \ldots x_\sigma x_\varepsilon . x_\varepsilon|\!\} & = \{\lambda x_1 . \ldots \{\lambda x_\sigma . \{\lambda x_\varepsilon . x_\varepsilon\}\} \ldots\} \\ [\![a_i; S]\!]_\Sigma & = & \lambda x_1 \ldots x_\sigma x_\varepsilon . x_i \ [\![S]\!]_\Sigma & \end{array}$$

The marked encoding $\{\!|S|\!\}_\Sigma$ *of a string* S *is the full marking of* $[\![S]\!]_\Sigma$*.*

With this encoding, the space cost of a string is bounded by a constant times its length plus the cost c_p of a pointer. We introduce the notation below for expressing such space complexity bounds.

▶ **Definition 4.7** ($\leqslant_\alpha$). *Let* α *be a constant, and* c*,* c' *two numbers. We write* $c \leqslant_\alpha c'$ *when* $c \leqslant \alpha \times c'$*.*

▶ **Lemma 4.8** (Cost of string representation). *Let* Σ *be an alphabet of size* σ*. There is a constant* α *such that for any character* $a \in \Sigma$*, any string* $S \in \Sigma^*$*, and any* $c_p, c_v \geqslant 1$ *we have* $\|[a]_\Sigma\|_{c_v, c_p} \leqslant_{\alpha c_v} c_p$ *and* $\|[\![S]\!]_\Sigma\|_{c_v, c_p} \leqslant_{\alpha c_v} c_p + |S|$*.*

This encoding however, does not allow us to build a universal App term satisfying $\mathsf{App}\ [a]_\Sigma\ [\![S]\!]_\Sigma \twoheadrightarrow^* [\![a;S]\!]_\Sigma$ in weak reduction. Instead, we need one $\mathsf{App}^\Sigma_{a_i}$ for each character a_i of each alphabet $\Sigma = \{a_1, \ldots, a_\sigma\}$. The App^Σ_a terms also take a continuation K as final parameter, as will all other main terms in this encoding, for smooth integration in the marked equivalent of the deterministic λ-calculus. There is a small divergence here from the encoding of [4], in which the continuation is always the first argument. While this difference is minor, we believe the present version slightly simplifies the encoding and improves its clarity.

▶ **Definition 4.9** (App^Σ_a). *For any alphabet $\Sigma = \{a_1, \ldots, a_\sigma\}$ and character $a_i \in \Sigma$, define the $\lambda_{\{\}}$-preterm*

$$\mathsf{App}^\Sigma_{a_i} \quad = \quad \lambda s.\lambda k.k\ (\lambda x_1 \ldots x_\sigma x_\varepsilon.x_i\ s)$$

Note that we consider App^Σ_a as a $\lambda_{\{\}}$-*preterm*, which will usually appear marked as in the lemma below.

▶ **Lemma 4.10** (App^Σ_a). *Let Σ be an alphabet of size σ. There is a constant α such that for any character $a \in \Sigma$, string $S \in \Sigma^*$, $\lambda_{\{\}}$-term K, and constants $c_p, c_v \geqslant 1$ we have $\{\mathsf{App}^\Sigma_a\}\ [\![S]\!]_\Sigma\ K\ \twoheadrightarrow^2_c\ K\ [\![a;S]\!]_\Sigma$ with space cost $c \leqslant_{\alpha c_v} c_p + |S| + \|K\|_{c_v,c_p}$.*

A number n is encoded as the sequence $\underline{n}_b$ of its binary digits, thus a string over the alphabet $\mathbb{B}$, starting with the lowest-order digit, with no trailing 0s, and 0 encoded as the empty sequence. Thus the length of $\underline{n}_b$ is proportional to $\mathsf{log}(n)$, and the cost of $[\![\underline{n}_b]\!]_\mathbb{B}$ is proportional to $\mathsf{log}(n) + c_p$ (there is one pointer at the end).

▶ **Definition 4.11** (Numbers). *Define the encoding $[\![\underline{n}_b]\!]_\mathbb{B}$ of n as*

$$\begin{array}{rcccl} [\![\underline{0}_b]\!]_\mathbb{B} & = & [\![\varepsilon]\!]_\mathbb{B} & = & \{\lambda x_0.\{\lambda x_1.\{\lambda x_\varepsilon.x_\varepsilon\}\}\} \\ [\![\underline{2\times n}_b]\!]_\mathbb{B} & = & [\![0;\underline{n}_b]\!]_\mathbb{B} & = & \lambda x_0 x_1 x_\varepsilon.x_0\ [\![\underline{n}_b]\!]_\mathbb{B} \\ [\![\underline{2\times n+1}_b]\!]_\mathbb{B} & = & [\![1;\underline{n}_b]\!]_\mathbb{B} & = & \lambda x_0 x_1 x_\varepsilon.x_1\ [\![\underline{n}_b]\!]_\mathbb{B} \end{array}$$

▶ **Example 4.12.** *An encoding of the number 6 goes as follows:*

$$\begin{array}{rcl} [\![\underline{6}_b]\!]_\mathbb{B} & = & [\![0;1;1]\!]_\mathbb{B} \\ & = & \lambda x_0 x_1 x_\varepsilon.x_0\ (\lambda x_0 x_1 x_\varepsilon.x_1\ (\lambda x_0 x_1 x_\varepsilon.x_1\ \{\lambda x_0.\{\lambda x_1.\{\lambda x_\varepsilon.x_\varepsilon\}\}\})) \end{array}$$

Each digit is coded by three λ-abstractions, one variable and one application, and the end of the sequence of digits is represented by the marked $\lambda_{\{\}}$-term $\{\lambda x_0.\{\lambda x_1.\{\lambda x_\varepsilon.x_\varepsilon\}\}\}$. The total space cost is thus $\|[\![\underline{6}_b]\!]_\mathbb{B}\|_{c_v,c_p} = 3(4 + c_v) + c_p$.

▶ **Lemma 4.13** (Number representation cost). *There is a constant α such that for any natural number n and constants $c_p, c_v \geqslant 1$ we have $\|[\![\underline{n}_b]\!]_\mathbb{B}\|_{c_v,c_p} \leqslant_{\alpha c_v} c_p + \mathsf{log}(n)$.*

▶ **Definition 4.14** (Turing machine configurations). *A Turing machine configuration $C = \langle I, n \mid W_l, b, W_r \mid q\rangle$ is represented by a Scott tuple of the encodings of its six components, where the input string I is fully marked.*

$$\langle\!\langle C \rangle\!\rangle \quad = \quad \lambda x.x\ \{\!|I|\!\}_\mathbb{I}\ [\![\underline{n}_b]\!]_\mathbb{B}\ [\![W_l]\!]_\mathbb{W}\ [b]_\mathbb{W}\ [\![W_r]\!]_\mathbb{W}\ [q]_\mathbb{Q}$$

We also write $\langle\!\langle C \rangle\!\rangle = \langle \{\!|I|\!\}_\mathbb{I}, [\![\underline{n}_b]\!]_\mathbb{B} \mid [\![W_l]\!]_\mathbb{W}, [b]_\mathbb{W}, [\![W_r]\!]_\mathbb{W} \mid [q]_\mathbb{Q}\rangle$.

▶ **Lemma 4.15** (Configuration representation cost). *There is a constant α such that for any constants $c_p, c_v \geqslant 1$ and configuration $C = \langle I, n \mid W_l, b, W_r \mid q\rangle$ we have the bound $\|\langle\!\langle C \rangle\!\rangle\|_{c_v,c_p} \leqslant_{\alpha c_v} c_p + \mathsf{log}(n) + |C|$.*

4.3 Arithmetic

This section provides adequate representations of the *succ* and *pred* functions on numbers. Remark that the most natural implementation of a function *succ* working on binary representations of numbers follows equations such as $succ(1;\underline{m_b}) = 0;succ(\underline{m_b})$ (this is indeed what is proposed in [4]). A direct implementation of this equation however gives a non-tail recursive algorithm accumulating 0's that will have to be prepended to the result once the base case $succ(0;\underline{m_b})$ or $succ(\varepsilon)$ is reached. This possibly logarithmic number of pending concatenations is not acceptable in our cost model, since each one would itself cost at least c_p, leading to a logarithmic instead of a constant-factor space-cost overhead.

We thus define successor and predecessor functions with tail-recursive algorithms given by the equations below. In both, the accumulator S is the sequence of elements that will have to be prepended to the result (in reverse order). The predecessor function assumes its argument is non-zero.

$$\begin{array}{rcl@{\qquad}rcl} succ(\underline{n_b}) & = & succ(\varepsilon \mid \underline{n_b}) & pred(\underline{n_b}) & = & pred(\varepsilon \mid \underline{n_b}) \\ succ(S \mid \varepsilon) & = & S^{\bowtie};1 & pred(S \mid 1;\varepsilon) & = & S^{\bowtie} \\ succ(S \mid 0;\underline{n_b}) & = & S^{\bowtie};1;\underline{n_b} & pred(S \mid 1;b;\underline{n_b}) & = & S^{\bowtie};0;b;\underline{n_b} \\ succ(S \mid 1;\underline{n_b}) & = & succ(0;S \mid \underline{n_b}) & pred(S \mid 0;\underline{n_b}) & = & pred(1;S \mid \underline{n_b}) \end{array}$$

Both functions will use a common reverse-append auxiliary function. These functions also classically rely on a fixpoint combinator.

The encoding uses a call-by-value fixpoint combinator, not as a choice of strategy but only to ensure deterministic reduction. We omit the marks that would disappear after the first iteration.

▶ **Definition 4.16** (Fix). *Define the fixpoint combinator* Fix *by the following $\lambda_{\{\}}$-preterms.*

$$\begin{array}{rcl} \vartheta & = & \lambda xy.y(\lambda z.xxyz) \\ \mathsf{Fix} & = & \{\vartheta\}\{\vartheta\} \end{array}$$

▶ **Lemma 4.17** (Fix). *There is a constant α such that for any $\lambda_{\{\}}$-term T and any $c_p, c_v \geqslant 1$ we have* $\mathsf{Fix}\ T \ \to_c^2 \ T\ (\lambda z.\mathsf{Fix}\ T\ z)$ *with space cost $c \leqslant_{\alpha c_v} c_p + \|T\|_{c_v,c_p}$.*

The reverse-append auxiliary function takes two bit strings S_1 and S_2, and computes $S_1^{\bowtie};S_2$ using the following equations.

$$\begin{array}{rcl} f(\varepsilon \mid S_2) & = & S_2 \\ f(b;S_1 \mid S_2) & = & f(S_1 \mid b;S_2) \end{array}$$

Its implementation is naturally tail-recursive. The actual term expects a continuation as third parameter, seen in the base case RA_ε.

▶ **Definition 4.18** (RevApp). *Define the reverse-append function* RevApp *by the following $\lambda_{\{\}}$-preterms, where RA_b describes both RA_0 and RA_1.*

$$\begin{array}{rcll} \mathsf{RevApp} & = & \mathsf{Fix}\ \{\mathsf{RA}\} & \\ \mathsf{RA} & = & \lambda f.\lambda s_1 s_2.s_1\ \{\mathsf{RA}_0\}\ \{\mathsf{RA}_1\}\ \{\mathsf{RA}_\varepsilon\}\ s_2\ f & \text{case on } s_1 \\ \mathsf{RA}_\varepsilon & = & \lambda s_2.\lambda f k.k\ s_2 & \text{if empty, return } s_2 \\ \mathsf{RA}_b & = & \lambda s_1' s_2.\{\mathsf{App}_b^\Sigma\}\ s_2\ (\lambda s_2'.\lambda f.f\ s_1'\ s_2') & \text{else append head to } s_2 \end{array}$$

▶ **Lemma 4.19** (RevApp). *There is a constant α such that for any two sequences $S_1, S_2 \in \mathbb{B}^*$, any $\lambda_{\{\}}$-term K, and any $c_p, c_v \geqslant 1$ we have* $\mathsf{RevApp}\ [\![S_1]\!]_{\mathbb{B}}\ [\![S_2]\!]_{\mathbb{B}}\ K \ \to_c^m \ K\ [\![S_1^{\bowtie};S_2]\!]_{\mathbb{B}}$, *with number of steps $m = \mathcal{O}(|S_1|)$ and space cost $c \leqslant_{\alpha c_v} c_p + |S_1| + |S_2| + \|K\|_{c_v,c_p}$.*

The successor and predecessor functions can now be defined in a tail-recursive way. Remark that, while the continuation is not directly visible in the code below, its presence is implied by the final calls to RevApp.

▶ **Definition 4.20** (Succ and Pred). *Define the successor function* Succ *and the predecessor function* Pred *by the following $\lambda_{\{\}}$-preterms. The subprogram* $\{\mathsf{W}\}$ *of* Pred *is arbitrary, and cannot be reached on a non-zero entry. Notation* P_{1b} *describes both* P_{10} *and* P_{11}.

$$
\begin{array}{rcll}
\mathsf{Succ} & = & \mathsf{Fix}\ \{\mathsf{S}\}\ [\![\varepsilon]\!]_{\mathbb{B}} & \\
\mathsf{S} & = & \lambda f.\lambda sn.n\ \{\mathsf{S}_0\}\ \{\mathsf{S}_1\}\ \{\mathsf{S}_\varepsilon\}\ s\ f & \textit{case on } n \\
\mathsf{S}_\varepsilon & = & \lambda s.\lambda f.\{\mathsf{App}_1^\Sigma\}\ [\![\varepsilon]\!]_{\mathbb{B}}\ (\lambda m'.\{\mathsf{RevApp}\}\ s\ m') & \textit{if } 0\textit{, return } 1 \\
\mathsf{S}_0 & = & \lambda m.\{\mathsf{App}_1^\Sigma\}\ m\ \{\lambda m's.\lambda f.\{\mathsf{RevApp}\}\ s\ m'\} & \textit{if even, ret. } n+1 \\
\mathsf{S}_1 & = & \lambda ms.\{\mathsf{App}_0^\Sigma\}\ s\ (\lambda s'.\lambda f.f\ s'\ m) & \textit{if odd, propagate carry} \\
\mathsf{Pred} & = & \mathsf{Fix}\ \{\mathsf{P}\}\ [\![\varepsilon]\!]_{\mathbb{B}} & \\
\mathsf{P} & = & \lambda f.\lambda sn.n\ \{\mathsf{P}_0\}\ \{\mathsf{P}_1\}\ \{\mathsf{W}\}\ s\ f & \textit{case on } n \\
\mathsf{W} & = & \{\lambda x.\{\lambda z.zz\}\ \{\lambda z.zz\}\} & \textit{if } 0\textit{, give up} \\
\mathsf{P}_0 & = & \lambda ms.\lambda f.\{\mathsf{App}_1^\Sigma\}\ s\ (\lambda s'.f\ s'\ m) & \textit{if even, propagate carry} \\
\mathsf{P}_1 & = & \lambda m.m\ \{\mathsf{P}_{10}\}\ \{\mathsf{P}_{11}\}\ \{\mathsf{P}_{1\varepsilon}\} & \textit{if odd, test further} \\
\mathsf{P}_{1\varepsilon} & = & \lambda s.\lambda f.\{\mathsf{RevApp}\}\ s\ [\![\varepsilon]\!]_{\mathbb{B}} & \textit{if } 1\textit{, return } 0 \\
\mathsf{P}_{1b} & = & \lambda r.\{\mathsf{App}_b^\Sigma\}\ r\ \{\mathsf{App}_0^\Sigma\}\ \{\lambda r's.\lambda f.\{\mathsf{RevApp}\}sr'\} & \textit{if } \geqslant 3\textit{, ret. } n-1
\end{array}
$$

▶ **Lemma 4.21** (Succ). *There is a constant α such that for any natural number n, any $\lambda_{\{\}}$-term K, and any $c_p, c_v \geqslant 1$ we have* $\mathsf{Succ}\ [\![\underline{n}_b]\!]_{\mathbb{B}}\ K \twoheadrightarrow_c^m K\ [\![\underline{n+1}_b]\!]_{\mathbb{B}}$, *with number of steps $m = \mathcal{O}(\mathsf{log}(n))$ and space cost $c \leqslant_{\alpha c_v} c_p + \mathsf{log}(n) + \|K\|_{c_v,c_p}$.*

▶ **Lemma 4.22** (Pred). *There is a constant α such that for any number $n > 0$, any $\lambda_{\{\}}$-term K, and any $c_p, c_v \geqslant 1$ we have* $\mathsf{Pred}\ [\![\underline{n}_b]\!]_{\mathbb{B}}\ K \twoheadrightarrow_c^m K\ [\![\underline{n-1}_b]\!]_{\mathbb{B}}$, *with number of steps $m = \mathcal{O}(\mathsf{log}(n))$ and space cost $c \leqslant_{\alpha c_v} c_p + \mathsf{log}(n) + \|K\|_{c_v,c_p}$.*

4.4 Transitions

The transition function $\delta(q, a, b)$ of a Turing machine is encoded as a sequence of three choices in a table, in order on the state q, then on the input character a, then on the work character b (order of [4] changed to halt early on final states). While b and q are readily available in the configuration, the input character a has to be retrieved by getting the character of index n in the input I. A string access function is defined using the following equations.

$$
\begin{array}{rcl}
f(\underline{0}_b \mid a; S) & = & a \\
f(\underline{n}_b \mid a; S) & = & f(\underline{n-1}_b \mid S) \quad \text{if } n > 0
\end{array}
$$

The function takes a continuation as third parameter. As in Definition 4.20, subprogram $\{\mathsf{W}\}$ cannot be reached on a valid input.

▶ **Definition 4.23** (Get). *Define the* Get *function by the following $\lambda_{\{\}}$-preterms.*

$$
\begin{array}{rcll}
\mathsf{Get} & = & \mathsf{Fix}\ \{\mathsf{G}\} & \\
\mathsf{G} & = & \lambda f.\lambda ni.n\ \{\mathsf{G}_0\}\ \{\mathsf{G}_1\}\ \{\mathsf{G}_\varepsilon\}\ i\ f & \textit{case on } n \\
\mathsf{G}_\varepsilon & = & \lambda i.i\ \{\mathsf{G}_{\varepsilon 0}\}\ \{\mathsf{G}_{\varepsilon 1}\}\ \{\mathsf{G}_{\varepsilon\diamond_l}\}\ \{\mathsf{G}_{\varepsilon\diamond_r}\}\ \{\mathsf{W}\} & \textit{if } 0\textit{, get head of } i \\
\mathsf{G}_{\varepsilon a} & = & \lambda j.\{\lambda f.\{\lambda k.k\ [a]_{\mathbb{I}}\}\} & \textit{and return char} \\
\mathsf{G}_b & = & \lambda mi.i\ \{\mathsf{G}_{b*}\}\ \{\mathsf{G}_{b*}\}\ \{\mathsf{G}_{b*}\}\ \{\mathsf{G}_{b*}\}\ \{\mathsf{W}\}\ m & \textit{if } > 0\textit{, get tail of } i \\
\mathsf{G}_{b*} & = & \lambda jm.\{\mathsf{App}_b^\Sigma\}\ m\ \{\lambda n.\{\mathsf{Pred}\}\ n\}\ (\lambda n'.\lambda f.f\ n'\ j) & \textit{and go on}
\end{array}
$$

▶ **Lemma 4.24** (Get). *There is a constant α such that for any input sequence I and natural number n such that $0 \leqslant n < |I|$, any $\lambda_{\{\}}$-term K, and any constants $c_p, c_v \geqslant 1$ we have* $\mathsf{Get}\ [\![\underline{n_b}]\!]_{\mathbb{B}}\ \{\!|I|\!\}_{\mathbb{I}}\ K \twoheadrightarrow_c^m K\ [I[n]]_{\mathbb{I}}$ *with number of steps $m = \mathcal{O}(n \times \log(n))$ and space cost $c \leqslant_{\alpha c_v} c_p + \log(n) + \|K\|_{c_v,c_p}$.*

The main function Compute simulates a step of the encoded Turing machine on the configuration given as parameter, and then iterates.

▶ **Definition 4.25** (Init, Accept, Reject). *Let $\mathcal{M} = (\mathbb{Q}, q_0, q_a, q_r, \delta)$ be a binary Turing machine. The initialization function is the following $\lambda_{\{\}}$-preterm, which builds an initial configuration when given a valid (encoding of an) input string $I = \diamond_l; I'; \diamond_r$.*

$$\mathsf{Init} \quad = \quad \lambda i.\lambda k.k\ \langle i, [\![\underline{0_b}]\!]_{\mathbb{B}} \mid [\![\varepsilon]\!]_{\mathbb{W}}, [\diamond]_{\mathbb{W}}, [\![\varepsilon]\!]_{\mathbb{W}} \mid [q_0]_{\mathbb{Q}} \rangle$$

Let Accept *and* Reject *be two arbitrary but distinguishable fully marked closed terms, respectively returned when computation stops on q_a or q_r.*

▶ **Definition 4.26** (Compute). *Let $\mathcal{M} = (\mathbb{Q}, q_0, q_a, q_r, \delta)$ be a binary Turing machine. Its transition function is encoded by the following $\lambda_{\{\}}$-preterms.*

$$\begin{array}{rcll}
\mathsf{Compute} & = & \mathsf{Fix}\ \{\mathsf{Step}\} & \\
\mathsf{Step} & = & \lambda f.\lambda c.c\ (\lambda in.\lambda w_l b w_r.\lambda q.q\ \{\mathsf{Q}_{q_1}\} \ldots \{\mathsf{Q}_{q_m}\}\ n\ i\ b\ n\ w_l\ w_r\ i)\ f & \\
\mathsf{Q}_{q_a} & = & \{\!|\lambda ni.\lambda bn.\lambda w_l w_r.\lambda if.\mathsf{Accept}|\!\} & \textit{halt on final state or} \\
\mathsf{Q}_{q_r} & = & \{\!|\lambda ni.\lambda bn.\lambda w_l w_r.\lambda if.\mathsf{Reject}|\!\} & \\
\mathsf{Q}_q & = & \lambda ni.\{\mathsf{Get}\}\ n\ i\ \{\mathsf{T}_q\} & \textit{select transition if } q \notin \{q_a, q_r\} \\
\mathsf{T}_q & = & \lambda a.a\ \{\mathsf{A}_{q,0}\}\ \{\mathsf{A}_{q,1}\}\ \{\mathsf{A}_{q,\diamond_l}\}\ \{\mathsf{A}_{q,\diamond_r}\} & \textit{dispatch on input char} \\
\mathsf{A}_{q,a} & = & \lambda b.b\ \{\mathsf{B}_{q,a,0}\}\ \{\mathsf{B}_{q,a,1}\}\ \{\mathsf{B}_{q,a,\diamond}\} & \textit{dispatch on work char} \\
\mathsf{B}_{q,a,b} & = & \lambda n.\{\mathsf{D}_d\}\ n\ \{\mathsf{M}_{b',m}\}\ [q']_{\mathbb{Q}} & \textit{for } \delta(q,a,b) = \langle d \mid b', m \mid q' \rangle \\
\mathsf{D}_{\downarrow} & = & \lambda nk.k\ n & \textit{update input head} \\
\mathsf{D}_{\leftarrow} & = & \mathsf{Pred} & \\
\mathsf{D}_{\rightarrow} & = & \mathsf{Succ} & \\
\mathsf{M}_{b',\downarrow} & = & \lambda n'q'.\lambda w_l w_r.\lambda if.f\ \langle i, n' \mid w_l, [b']_{\mathbb{W}}, w_r \mid q' \rangle & \textit{upd. work tape} \\
\mathsf{M}_{b',\leftarrow} & = & \lambda n'q'.\lambda w_l w_r.\{\mathsf{App}_{b'}^{\Sigma}\}\ w_r\ (\lambda w_r'.w_l\{\mathsf{L}_0\}\{\mathsf{L}_1\}\{\mathsf{L}_\diamond\}\{\mathsf{L}_\varepsilon\}w_r')\ n'\ q' & \\
\mathsf{M}_{b',\rightarrow} & = & \lambda n'q'.\lambda w_l w_r.\{\mathsf{App}_{b'}^{\Sigma}\}\ w_l\ (\lambda w_l'.w_r\{\mathsf{R}_0\}\{\mathsf{R}_1\}\{\mathsf{R}_\diamond\}\{\mathsf{R}_\varepsilon\}w_l')\ n'\ q' & \\
\mathsf{L}_b & = & \lambda w_l' w_r'.\{\mathsf{Z}\}\ w_l'\ [b]_{\mathbb{W}}\ w_r' & \textit{move left} \\
\mathsf{L}_\varepsilon & = & \lambda w_r'.\{\{\{\mathsf{Z}\}\ [\![\varepsilon]\!]_{\mathbb{W}}\}\ [\diamond]_{\mathbb{W}}\}\ w_r' & \textit{move left (empty)} \\
\mathsf{R}_b & = & \lambda w_r' w_l'.\{\mathsf{Z}\}\ w_l'\ [b]_{\mathbb{W}}\ w_r' & \textit{move right} \\
\mathsf{R}_\varepsilon & = & \lambda w_l'.\{\mathsf{Z}\}\ w_l'\ [\diamond]_{\mathbb{W}}\ [\![\varepsilon]\!]_{\mathbb{W}} & \textit{move right (empty)} \\
\mathsf{Z} & = & \lambda w_l' b' w_r'.\lambda n'q'.\lambda if.f\ \langle i, n' \mid w_l', b', w_r' \mid q' \rangle & \textit{build config.}
\end{array}$$

▶ **Lemma 4.27** (Compute (one step)). *Let $\mathcal{M} = (\mathbb{Q}, q_0, q_a, q_r, \delta)$ be a binary Turing machine, and* Compute *be its encoding in the $\lambda_{\{\}}$-calculus. There is a constant α such that for any two configurations C, C' on input I such that there is a transition step $C \longmapsto C'$, and any $c_p, c_v \geqslant 1$ we have* $\mathsf{Compute}\ \langle\!\langle C \rangle\!\rangle \twoheadrightarrow_c^m \mathsf{Compute}\ \langle\!\langle C' \rangle\!\rangle$ *with number of steps $m = \mathcal{O}(|I| \times \log(|I|))$ and space cost $c \leqslant_{\alpha c_v} c_p + \log(|I|) + |C|$.*

▶ **Theorem 4.28** (Soundness (Full simulation)). *Let $\mathcal{M} = (\mathbb{Q}, q_0, q_a, q_r, \delta)$ be a binary Turing machine. There are constants n_0 and α such that for any input string $I \in \mathbb{B}^*$ of length $|I| \geqslant n_0$, if $\langle \diamond_l; I; \diamond_r, 0 \mid \varepsilon, \diamond, \varepsilon \mid q_0 \rangle \longmapsto_c^* \langle \diamond_l; I; \diamond_r, n \mid w_l, b, w_r \mid q_f \rangle$ with $q_f \in \{q_a, q_r\}$, with space cost c, then there is a computation*

$$\{\{\{\mathsf{Init}\}\ \{\!|\diamond_l; I; \diamond_r|\!\}_{\mathbb{I}}\}\ \{\lambda c.\{\mathsf{Compute}\}\ c\}\} \twoheadrightarrow_{c'}^* R$$

with $R = \mathsf{Accept}$ or $R = \mathsf{Reject}$, with space cost $c' \leqslant_\alpha c + \log(|I|)$.

Remark that the global complexity of the simulation does not mention c_v nor c_p, contrarily to all previous lemmas in this section. These two constants are eliminated in the proof for distinct reasons:

- the variable cost c_v depends only on the machine, and is in particular constant with respect to the input string I, and thus asymptotically bounded by $\mathsf{log}(|I|)$;
- the pointer cost c_p depends on the encodings of the machine and of the input, in which the input dominates asymptotically, thus c_p is bounded by a constant factor times $\mathsf{log}(|I|)$.

Therefore, any Turing machine with a logarithmic or higher space complexity can be simulated in the $\lambda_{\{\}}$-calculus (and thus in the weak λ-calculus) with a constant-factor space overhead.

5 Space-sound implementation of $\lambda_{\{\}}$

We will now show that our $\lambda_{\{\}}$-calculus can be implemented on a Turing machine, with a constant-factor overhead in space complexity. For this, we present a weak reduction machine, which keeps a read-only copy of the input λ-term and uses explicit pointers to closed subterms of the input.

While the machine is kept at a semi-abstract level, the $\lambda_{\{\}}$-terms are encoded by sequences of symbols that will be easily represented on a Turing machine tape, and auxiliary operations are decomposed in such a way that their Turing implementation is transparent. A compilation scheme also translates from named λ-terms to nameless programs using de Bruijn indices. The machine and the compilation scheme are inspired by Forster, Kunze and Roth [16], with two major differences: first we include a read-only input and manage explicit pointers to the input, and second we relax the reduction strategy to avoid an early restriction to call-by-value.

5.1 Programs and configurations

In our implementation, we represent $\lambda_{\{\}}$-terms as unambiguous and easy-to-parse sequences of symbols, based on a prefix encoding.

▶ **Definition 5.1** (Programs). *A* raw program P *is a sequence over the (infinite) set of symbols* $\Sigma_P = \{\mathtt{app}, \mathtt{lam}, \mathtt{mal}\} \cup \{\mathtt{var}\ k \mid k \in \mathbb{N}\} \cup \{\mathtt{ptr}\ a \mid a \in \mathbb{N}\}$. *A* well-formed program T *is generated by the following grammar.*

$$T \quad ::= \quad \mathtt{var}\ k \quad | \quad \mathtt{app}\ ;\ T\ ;\ T \quad | \quad \mathtt{lam}\ ;\ T\ ;\ \mathtt{mal} \quad | \quad \mathtt{ptr}\ a$$

An input program *is a well-formed program without* `ptr` a *symbols.*

▶ **Definition 5.2** (Compilation). *The compilation function* $(\!|t|\!)$ *translates a closed* λ*-term* t *into an input program. The auxiliary function* $(\!|t|\!)_\rho$ *takes a possibly open term* t *and a sequence* ρ *ordering the free variable names of* t *for computing de Bruijn indices.*

$$\begin{array}{rclcrcl} (\!|t|\!) & = & (\!|t|\!)_\varepsilon & & (\!|x|\!)_\rho & = & \mathtt{var}\ k \quad \text{if } \rho[k] = x \text{ and } \forall j < k, \rho[j] \neq x \\ (\!|t\ u|\!)_\rho & = & \mathtt{app}; (\!|t|\!)_\rho; (\!|u|\!)_\rho & & (\!|\lambda x.t|\!)_\rho & = & \mathtt{lam}; (\!|t|\!)_{x;\rho}; \mathtt{mal} \end{array}$$

▶ **Example 5.3** (Compilation). *The* λ*-term* $t = (\lambda z.z)\ \Big((\lambda x.x(\lambda y.y(\lambda z.z)x))\ \ (\lambda z.z)\Big)$ *from Example 2.5 is compiled as a the following input program* I*, in which the indentation is not part of the syntax but emphasizes the structure.*

```
app; lam; var 0; mal;
    app; lam; app; var 0;
                lam; app; app; var 0; lam; var 0; mal; var 1; mal; mal;
        lam; var 0; mal
```

A state of our machine combines an initial, read-only input program with a work program representing the ongoing computation.

▶ **Definition 5.4** (Configurations). *A* configuration *is a pair* $\langle I \mid T \rangle$ *with* I *an* input program*, and* T *a well-formed program, called* working program*, which may contain pointers to the input program* I*. The initial configuration for evaluating a* λ*-term* t *is* $\langle (\!|t|\!) \mid \texttt{ptr}\ 0 \rangle$*.*

A working program T of appropriate shape in a configuration $\langle I \mid T \rangle$ represents a $\lambda_{\{\}}$-term t. In this readback operation, the input program I is used to interpret the pointers of the program T.

▶ **Definition 5.5** (Readback). *The* readback *operation* $\gg_I^\rho$ *interprets a well-formed program* T *as a* $\lambda_{\{\}}$*-term* t*, depending on an input program* I *and a sequence* ρ *of names for the free variables of* t*.*

$$
\begin{array}{rll}
\texttt{var}\ k \gg_I^\rho \rho[k] & & \\
\texttt{app}; T_1; T_2 \gg_I^\rho t_1\ t_2 & & \text{if } T_1 \gg_I^\rho t_1 \text{ and } T_2 \gg_I^\rho t_2 \\
\texttt{lam}; T_0; \texttt{mal} \gg_I^\rho \lambda x.t_0 & & \text{if } T_0 \gg_I^{x;\rho} t_0\text{, with } x \text{ fresh} \\
\texttt{ptr}\ a \gg_I^\rho \{\!|t|\!\} & & \text{if there is } b \text{ such that } I[a..b) \gg_I^\rho t
\end{array}
$$

The index b *in the last equation is unique, and exists if and only if* a *is the first index of a well-formed subprogram of* I*. We write* $\gg_I$ *for* $\gg_I^\varepsilon$*. A* closed program *is a* T *that reads back to a closed* $\lambda_{\{\}}$*-term.*

Remark that a translation from $\lambda_{\{\}}$-terms to working programs would be ambiguous, even with a fixed input program I: if I contains two inadvertently equal subprograms representing a λ-term t, then the marked term $\{\!|t|\!\}$ could be represented by either pointer. Readback, on the other hand, is deterministic modulo α-renaming.

▶ **Example 5.6** (Readback). *In this example we write* $[T \gg_I^\rho]$ *for the unique term* t *such that* $T \gg_I^\rho t$*. Given the input program* I *from Example 5.3, and the name* x*, we have:*

$$
\begin{array}{ll}
& [\texttt{lam}; \texttt{app}; \texttt{app}; \texttt{var}\ 0; \texttt{ptr}\ 12; \texttt{var}\ 1; \texttt{mal} \gg_I^x] \\
= & \lambda y.[\texttt{app}; \texttt{app}; \texttt{var}\ 0; \texttt{ptr}\ 12; \texttt{var}\ 1 \gg_I^{y;x}] \\
= & \lambda y.[\texttt{app}; \texttt{var}\ 0; \texttt{ptr}\ 12 \gg_I^{y;x}]\ [\texttt{var}\ 1 \gg_I^{y;x}] \\
= & \lambda y.[\texttt{var}\ 0 \gg_I^{y;x}]\ [\texttt{ptr}\ 12 \gg_I^{y;x}]\ [\texttt{var}\ 1 \gg_I^{y;x}] \\
= & \lambda y.y\ [\texttt{ptr}\ 12 \gg_I^{y;x}]\ x \\
= & \lambda y.y\ \{\!|\lambda z.z|\!\}\ x \\
= & \lambda y.y\ \{\lambda z.z\}\ x
\end{array}
$$

where $\texttt{ptr}\ 12 \gg_I^{y,x} \{\!|\lambda z.z|\!\}$ *since* $I[12..15) = \texttt{lam}; \texttt{var}\ 0; \texttt{mal}$ *and* $\texttt{lam}; \texttt{var}\ 0; \texttt{mal} \gg_I^{y,x} = \lambda z.z$*.*

The space cost of a configuration only counts the space cost of its working program, and represents the cost of actually writing down this program: each atomic symbol costs 1, and the natural numbers in $\texttt{var}\ k$ and $\texttt{ptr}\ a$ cost the actual size of their (binary) representation. The latter is more low-level than the cost measure on $\lambda_{\{\}}$-terms, since the cost of each index or pointer is computed from its actual value rather than given by a program-wide constant.

▶ **Definition 5.7** (Cost of programs and configurations). *We write* $\|\langle I \mid T \rangle\|$ *for the space cost of a configuration* $\langle I \mid T \rangle$*, and* $\|P\|$ *for the space cost of a raw program* P*.*

$$
\begin{array}{rcl@{\qquad}rcl}
\|\langle I \mid T \rangle\| & = & \|T\| & & & \\
\|\texttt{var}\ k; P\| & = & \mathsf{max}(1, \lceil \mathsf{log}(k) \rceil) + \|P\| & \|c; P\| & = & 1 + \|P\| \quad \text{if } c \in \{\texttt{app}, \texttt{lam}, \texttt{mal}\} \\
\|\texttt{ptr}\ a; P\| & = & \mathsf{max}(1, \lceil \mathsf{log}(a) \rceil) + \|P\| & \|\varepsilon\| & = & 0
\end{array}
$$

▶ **Lemma 5.8** (Cost of programs). *Let I be an input program and t_0 a closed λ-term such that $(\!|t_0|\!) = I$. Define $c_v = \mathsf{max}(1, \lceil \mathsf{log}(\mathsf{vd}(t_0)) \rceil)$ and $c_p = \mathsf{max}(1, \lceil \mathsf{log}(|t_0|) \rceil)$. For any program T, if $T \gg_I^\rho t$ for some sequence of names ρ and $\lambda_{\{\}}$-term t such that $\mathsf{vd}(t) \leqslant \mathsf{vd}(t_0)$, then $\|T\| \leqslant 2 \times \|t\|_{c_v,c_p}$.*

5.2 Reduction

Computation steps of the machine are defined by two rules only.

- A subprogram $\mathtt{app}; \mathtt{lam}; U; \mathtt{mal}; S$, where U and S are well-formed subprograms, may trigger a β-step that substitutes S into U. This rule relies on parsing operations identifying the subprograms S and U (Definition 5.9), and on program substitution (Definition 5.13).
- A pointer $\mathtt{ptr}\ a$ can be dereferenced, which means reading the target subprogram U in I and copying it in place of the pointer. This operation (Definition 5.10) also detects closed subprograms of U: these should not be copied but instead referenced by new pointers.

The main parsing function $\mathcal{T}(Q, a)$ applies to a raw program Q and an index a, assuming that a is the first index of a well-formed subprogram $Q[a..b)$ of Q, and returns its end index b if $Q[a..b)$ is closed, or a special symbol $\perp$ otherwise. The parsed program is assumed to be well-formed, thus this function does not check program validity. It locates the end of the subprogram by counting `app` symbols and subprograms, and delegates parsing of λ-abstractions to an auxiliary function $\mathcal{A}(Q, a)$ counting `lam` and `mal` symbols.

▶ **Definition 5.9** (Parsing functions). *$\mathcal{A}(Q, a)$ parses the body of a λ-abstraction starting at index a in Q (not counting the opening* `lam`*, but including the closing* `mal`*), and returns the next index. The auxiliary function $\mathcal{A}_Q(a, k)$ takes the current number k of open binders as an additional parameter.*

$$\begin{array}{rcl} \mathcal{A}(Q,a) & = & \mathcal{A}_Q(a,1) \\ \mathcal{A}_Q(a,0) & = & a \end{array} \qquad \begin{array}{rcl} \mathcal{A}_Q(a,k) \\ (k>0) \end{array} = \begin{cases} \mathcal{A}_Q(a+1,k+1) & \text{if } Q[a] = \mathtt{lam} \\ \mathcal{A}_Q(a+1,k-1) & \text{if } Q[a] = \mathtt{mal} \\ \perp & \text{if } Q[a] = \mathtt{var}\ n \text{ with } n \geqslant k \\ \mathcal{A}_Q(a+1,k) & \text{otherwise} \end{cases}$$

The main function $\mathcal{T}(Q, a)$ parses a $\lambda_{\{\}}$-term starting at index a in Q and returns the next index, or $\perp$ if the term is not closed. The auxiliary function $\mathcal{T}_Q(a, s)$ takes as an additional parameter the number s of subterms that are required to complete all pending applications.

$$\begin{array}{l} \mathcal{T}(Q,a) = \mathcal{T}_Q(a,1) \\ \mathcal{T}_Q(a,0) = a \end{array} \qquad \begin{array}{rcl} \mathcal{T}_Q(a,s) \\ (s>0) \end{array} = \begin{cases} \mathcal{T}_Q(a+1,s+1) & \text{if } Q[a] = \mathtt{app} \\ \mathcal{T}_Q(a+1,s-1) & \text{if } Q[a] = \mathtt{ptr}\ b \\ \mathcal{T}_Q(a',s-1) & \text{if } Q[a] = \mathtt{lam} \text{ and } \mathcal{A}(Q,a+1) = a' \\ \perp & \text{if } Q[a] = \mathtt{var}\ n \end{cases}$$

The copy function extracts a subprogram starting at a given index in the input program, introducing pointers for closed subprograms.

▶ **Definition 5.10** (Copy function). *$\mathcal{R}(I, a)$ returns a shallow copy of the subprogram starting at index a of I. The auxiliary function $\mathcal{R}_I(a, w)$ takes an end index w as an additional parameter, and introduces a pointer whenever a is the start index of a closed subterm.*

$$\begin{array}{rcl} \mathcal{R}(I,a) & = & I[a]; \mathcal{R}_I(a+1,w) \quad \text{with } w = \mathcal{T}(I,a) \\ \mathcal{R}_I(w,w) & = & \varepsilon \\ \mathcal{R}_I(a,w) \\ (a<w) \end{array} = \begin{cases} I[a]; \mathcal{R}_I(a+1,w) & \text{if } I[a] = \mathtt{mal} \text{ or } \mathcal{T}(I,a) = \perp \\ \mathtt{ptr}\ a; \mathcal{R}_I(a',w) & \text{if } \mathcal{T}(I,a) = a' \end{cases}$$

The last equation assumes $a' \leqslant w$, which is guaranteed on a well-formed input.

Both parsing functions and the copy function assume the given index a is indeed the start index of a well-formed subprogram in Q (or I), as seen in the soundness lemma below.

▶ **Lemma 5.11** (Soundness of parsing and copy functions).
- *If* $Q[a..b) \gg_I^x t$ *and* $Q[b] = \texttt{mal}$*, then* $\mathcal{A}(Q,a) = b+1$.
- *If* $Q[a..b) \gg_I t$*, then* $\mathcal{T}(Q,a) = b$.
- *If* $I[a..b) \gg_I t$*, then* $\mathcal{R}(I,a) \gg_I \{\!|t|\!\}$.

Reminder: if $T \gg_I^\rho t$ *then any free variable of* t *is in* ρ*. This implies that* $\lambda x.t$ *is closed in the first point, and* t *is closed in the last two.*

▶ **Example 5.12** (Parsing and copy). *Consider again the program* I *of Example 5.3. Then we have* $\mathcal{A}(I,2) = 4$ *and* $\mathcal{A}(I,6) = 18$ *and* $\mathcal{A}(I,19) = 21$*, from which we deduce* $\mathcal{T}(I,0) = 21$ *and* $\mathcal{T}(I,1) = 4$ *and* $\mathcal{T}(I,5) = 21$*, from which we finally build* $\mathcal{R}(I,0) = \texttt{app};\texttt{ptr}\ 1;\texttt{ptr}\ 4$.

The machine uses a naive implementation of β-reduction based on substitution, similarly to the "substitution machine" in [16]. We define a substitution on programs, closely reflecting usual substitution on the represented $\lambda_{\{\}}$-terms. The operation follows the sequence of symbols rather than its structure, the target de Bruijn index k being updated upon crossing the symbols `lam` and `mal`.

▶ **Definition 5.13** (Program substitution). *We write* $P^{\{k \leftarrow S\}}$ *for the program* P *in which the variable of index* k *is replaced by the closed program* S.

$$\begin{aligned}
(\texttt{lam};P)^{\{k\leftarrow S\}} &= \texttt{lam};P^{\{k+1\leftarrow S\}} \qquad (\texttt{var}\ n;P)^{\{k\leftarrow S\}} = \begin{cases} S;P^{\{k\leftarrow S\}} & k=n \\ \texttt{var}\ n;P^{\{k\leftarrow S\}} & k\neq n \end{cases} \\
(\texttt{mal};P)^{\{k\leftarrow S\}} &= \texttt{mal};P^{\{k-1\leftarrow S\}} \qquad \varepsilon^{\{k\leftarrow S\}} = \varepsilon \\
(\texttt{app};P)^{\{k\leftarrow S\}} &= \texttt{app};P^{\{k\leftarrow S\}}
\end{aligned}$$

In the rule for `mal`*, it is assumed the target index* k *is non-zero, which is guaranteed if substitution is applied to a well-formed program.*

▶ **Lemma 5.14** (Soundness of program substitution). *Let* t*,* s *be* $\lambda_{\{\}}$*-terms with* s *closed, and* T*,* S *programs such that* $T \gg_I^\rho t$ *and* $S \gg_I s$*. Then* $T^{\{k\leftarrow S\}} \gg_I^\rho t^{\{\rho[k]\leftarrow s\}}$.

We can now define the two computation rules of the machine. The rule for β-reduction is triggered by an `app;lam` pattern, and then uses the parsing functions $\mathcal{A}(Q,a)$ and $\mathcal{T}(Q,a)$ to identify the appropriate subterms.

▶ **Definition 5.15** (Reduction). *The step relation of the machine, written* $\langle I \mid T\rangle \rightsquigarrow \langle I \mid T'\rangle$*, is defined by:*

$$\begin{aligned}
\langle I \mid P;\texttt{app};\texttt{lam};Q\rangle &\rightsquigarrow \langle I \mid P;Q[0..a-1)^{\{0\leftarrow Q[a..b)\}};Q[b..)\rangle \quad \text{if } a=\mathcal{A}(Q,0),\ b=\mathcal{T}(Q,a) \\
\langle I \mid P;\texttt{ptr}\ a;Q\rangle &\rightsquigarrow \langle I \mid P;\mathcal{R}(I,a);Q\rangle
\end{aligned}$$

In the first rule it is implied that we have $Q[a-1] = \texttt{mal}$.

Contrarily to what is ordinary implied in an abstract machine, these rules are not deterministic: we do not define here any reduction strategy specifying which computation step should be triggered in a given configuration. The goal is to be able to implement any reduction sequence of the $\lambda_{\{\}}$-calculus.

▶ **Example 5.16.** *Consider the input program* I *from Example 5.3. The first steps in call-by-name evaluation would be as follows.*

$$\langle I \mid \texttt{ptr}\ 0\rangle \rightsquigarrow \langle I \mid \texttt{app};\texttt{ptr}\ 1;\texttt{ptr}\ 4\rangle \rightsquigarrow \langle I \mid \texttt{app};\texttt{lam};\texttt{var}\ 0;\texttt{mal};\texttt{ptr}\ 4\rangle \rightsquigarrow \langle I \mid \texttt{ptr}\ 4\rangle$$

Remark that the definition of $\mathcal{T}(Q, a)$ assumes the target program is closed, which implicitly restricts the machine to *closed* reduction (which is little more than weak reduction, since it allows reduction below abstractions only on closed subterms). This is consistent with the focus of the paper on closed weak reduction, but could be easily extended.

▶ **Lemma 5.17** (Soundness (one step)). *Let I be an input program and t_0 a closed λ-term such that $(\!|t_0|\!) = I$. Define the constants $c_v = \mathsf{max}(1, \lceil \mathsf{log}(\mathsf{vd}(t_0)) \rceil)$ and $c_p = \mathsf{max}(1, \lceil \mathsf{log}(|t_0|) \rceil)$. Let t, t' be closed and well-formed $\lambda_{\{\}}$-terms such that $t \twoheadrightarrow t'$. For all T such that $T \gg_I t$ there is T' such that $T' \gg_I t'$ and for all P, Q we have $\langle I \mid P; T; Q\rangle \rightsquigarrow^* \langle I \mid P; T'; Q\rangle$ with space-cost $\|\langle I \mid P; Q\rangle\| + c$ with $c \leqslant 2 \times (\mathsf{max}(\|t\|_{c_v,c_p}, 2c_p + 1) + \|t'\|_{c_v,c_p})$.*

▶ **Theorem 5.18** (Soundness (computations)). *For any closed λ-term t, if $\{\!|t|\!\} \twoheadrightarrow_c^* t'$ with space cost $c \geqslant \mathsf{log}(|t^*|)$, then*

$$\langle (\!|t|\!) \mid \texttt{ptr}\ 0\rangle \rightsquigarrow_{c'}^* \langle (\!|t|\!) \mid T'\rangle$$

with $T' \gg_I t'$, with space cost $c' \leqslant 6c + 2$.

5.3 Simulation of $\lambda_{\{\}}$ by Turing machines

An abstract machine configuration $\langle I \mid T\rangle$ is naturally connected to a configuration of a Turing machine with two tapes, whose read-only input tape contains the input program I, and whose work tape contains the working program T as well as the working data of the auxiliary operations. For convenience we can assume the alphabet is large enough to offer a distinct symbol for each program element, plus some extra auxiliary symbols ; it is folklore that any such implementation can then be encoded into a binary machine while respecting the complexity. All the operations of the abstract machine presented in this section can be implemented with polynomial time and constant-factor space overhead. In particular:

- the parsing and copy functions are implemented as simple loops, with the parameter s, k or w as an auxiliary integer data and the index a either represented directly as an integer data when indexing the input, or indirectly as a special cursor character in the work tape,
- program substitution scans a segment of the working program between two special cursor characters (current position and end position of the segment that receives the substitution), and maintains as auxiliary data the target index k and the substituted program S.

In all these operations, the working head moves back and forth between the working program and the auxiliary data, possibly shifting the contents of a part of the tape when the size of an element changes, all in polynomial time. The space overhead mostly corresponds to the auxiliary data on the working tape, whose size is bounded by the size of the working program and the logarithm of the size of the input.

The final unspecified part is redex search in an abstract machine configuration, corresponding to a reduction strategy in the weak λ-calculus. Call-by-value and call-by-name strategies are easily implemented in our abstract machine:

- for (right-to-left) call-by-value, select the rightmost `app`;`lam` pattern that is not inside a matching `lam`/`mal` pair,
- for call-by-name, select the leftmost such pattern,

and in both, dereference only the pointers needed to make these patterns appear.

Finally, any λ-calculus program with a logarithmic or higher space complexity under call-by-value, call-by-name or any other LOGSPACE-definable weak evaluation strategy can be executed on a Turing machine with a constant-factor overhead in space.

6 Discussion

6.1 Variants of the cost measure

The space-cost measure presented in this paper is actually just one instance, chosen as the most abstract and (subjectively) clear in a family of related reasonable cost measures. In particular, all the definitions and proofs we provide here can be adapted to accommodate the variants discussed below.

Using more pointers. In our main model, we assign a pointer cost to full descendants of closed subterms only. We could instead remove the closedness restriction and assign a pointer cost to full descendants of any initial subterm. For this we adapt the notion of full marking, which now puts a mark at every position of a term, and relax the notion of well-formed term: every marked subterm is still fully marked, but not required to be closed anymore (on the other hand, we still have that any closed subterm contains at least one mark). The encoding of Turing machines is easily adapted by adding appropriate marks. The abstract machine has to be modified in a slightly subtler way, since the copy function (Definition 5.10) now has to introduce pointers for any subprogram, and not only for closed subprograms. The obtained measure seems very close to the one detailed in the paper.

Using less pointers. In the paper we assign a pointer cost to any full descendant of a closed initial subterm, even when said subterm would cost less than a pointer. We could instead reduce the measured cost by assigning a pointer cost only for subterms that are "big enough", by taking a min between a pointer cost and the actual cost of the considered subterm. The notion of full marking and the copy function of the abstract machine are adapted the obvious way, with a check of the actual cost of a subterm or subprogram against the pointer cost c_p. A more interesting difference appears in the encoding of Turing machines, where the constant costs of the λ-terms defining the machine itself will be (asymptotically, if not from the beginning) outweighed by the pointer cost, since the latter grows with the input. This remark applies in particular to App_a^Σ, which will now have a constant cost (that means, depending on Σ but not on the input of the machine), instead of a pointer cost. Then, it is not necessary anymore to use tail-recursive versions of Pred and Succ to get a reasonable encoding, and we can switch back to the simpler presentation of arithmetic used in [4].

Using non-uniform pointers. Our choice of a uniform cost for pointers is an abstraction: the constant c_p can be thought of as an upper bound of the cost of any actual pointer used. This allows us to avoid making any assumption on the way the term is represented (except the standard assumption that the representation stays compact). However, this also induces a disturbing scaling phenomenon when considering the encoding of a Turing machine: code pointers are not distinguished from data pointers, and the cost of code pointers thus grows logarithmically with the size of the input, even though code pointers would be naturally considered to be independent from the input. This scaling phenomenon is visible in the paper: it is the reason why we use tail-recursive encodings of arithmetic operations in Section 4.3.

As an alternative, we could borrow the low-level notion of pointer cost used in [5] under the concept of *left addresses*. For this we replace our uniform pointer cost by the actual size of a pointer to the target subterm in the initial term, seen as an integer representing the index of the subterm in a prefix notation of the term. Remark that this is de facto already the case in our abstract machine, and thus that Section 5 does not need any modification.

This low-level version allows us to make sure that the λ-terms encoding a Turing machine have a constant cost (a pointer to a prefix of the term that does not depend on the input string), and thus allow a naive encoding of arithmetic.

6.2 Comparison with other measures

The log-sensitive space-cost measure proposed by Accattoli, Dal Lago and Vanoni [5, 3] solves a similar problem than ours: equipping the λ-calculus with a space-cost measure in such a way that the λ-calculus can simulate -and be simulated by- Turing machines with a constant-factor space overhead, thus providing a λ-calculus-based way of characterizing standard space complexity classes.

Interestingly however, the two measures are not equivalent. Although they agree on which problems can be solved or not under a given asymptotic space constraint, they differ on the complexity they assign to specific algorithms. In other words, they are extensionally equivalent, but not intentionally equivalent. This was already apparent in Section 4, in the discussion about implementing binary arithmetic: both models agree on the fact that the increment of a binary number can be computed with a logarithmic space cost. However, the naive recursive algorithm for doing so has distinct asymptotic complexity under the two measures: it runs in $\mathcal{O}(\mathsf{log}(n))$ space according to Accattoli et al's measure, and in $\mathcal{O}(\mathsf{log}(n)^2)$ space according to ours. Equivalent extensional expressivity is retrieved here since we can provide a slightly different algorithm (the tail-recursive version given in Definition 4.20) that solves the same problem with the required complexity $\mathcal{O}(\mathsf{log}(n))$ in our model. Remark however that even in the family of complexity models presented in this paper, all measures are not intentionally equivalent: two of the variants discussed in Section 6.1 also give a logarithmic space complexity to the naive binary increment algorithm.

Beyond this observation, let us look more closely at what is measured by the two models.

- Accattoli et al's measure counts the closures used for the evaluation of the term. An important property of the model is that evaluation never builds a new term, but only handles pointers to the input term. In particular, the cost of a closure is the cost of its term pointer.
- On the other hand, our model mixes references to the input term with new term parts. Accordingly our measure counts a unit cost for any newly allocated term constructor, and a pointer cost for subterms that are shared with the input.

As a result, Accattoli et al's model may use more pointers than ours in some situations. Consider the term $(\lambda xy.xy)(\lambda z_1.z_1)(\lambda z_2.z_2)$. After the first two β-steps, Accattoli et al's model has three closures, and thus three pointers: one on xy, and one on each argument. In our model, we have a new, unit cost application of the two arguments (which both have a pointer cost).

It is not clear whether the above phenomenon alone can create a significant space-cost gap between the two measures. There is however another important difference, which is not related to how the space cost of a given configuration is defined, but to the evaluation model itself: Accattoli et al's machine is based on a small-step substitution operation (*linear substitution*), while ours uses the standard full substitution of the λ-calculus. As a result, even if we consider the same evaluation strategy (for instance: call-by-value), the two evaluation models go through significantly different intermediate steps, and finally define their global space measure as a max on incomparable sets. This remark can be used to forge terms on which the two measures give significantly different results. Consider for instance the term $t_n = (\lambda x.x(x(x(\dots(xx)\dots))))(\lambda z.z)$ with n occurrences of x in the left member. In the λ-calculus, the first β-reduction step performs n copies of the argument $\lambda z.z$, with a

total space cost proportional to $n \times \mathsf{log}(n)$ in our model (each copy of $\lambda z.z$ costs c_p, itself proportional to $\mathsf{log}(n)$). In the call-by-name version of Accattoli et al's machine however, thanks to on-demand linear substitution, we never need more than 4 pointers at the same time for evaluating this term.

In the end, it seems that the biggest difference between the two cost measures does not come from what is measured, but rather from the different underlying evaluation models: the pure λ-calculus with full substitution, or a refined model with on-demand linear substitution. This emphasizes a contrast between space and time complexity: a linear substitution-based model is an appropriate intermediate step for showing the time-reasonability of β-reduction (it is the path taken for instance in [2]), while it does not seem to be so for space-reasonability.

6.3 Extension to open weak reduction

The technical development in this paper assumes closed terms, and is thus restricted to closed weak reduction. Note however that it relies only superficially on this assumption, and could be extended to open weak reduction with moderate work.

In particular, most definitions and results still work as expected when considering the globally free variables as constants, and interpreting as "closed" any subterm whose only free variables are constants. The main required adaptations are the following.

- Adjusting the variable cost c_v to take the constants into account. For this, consider that each constant $x \in \mathsf{fv}(t_0)$ increments the variable depth $\mathsf{vd}(t_0)$ of an initial term t_0.
- Adapting the treatment of variables in the parsing functions in Section 5.2, to avoid rejecting terms with constants. This requires an additional parameter, similar to the already present k parameter, to identify the indices that represent constants.
- Extending the call-by-value strategy in Section 5.3 to account for "rigid values": applications $x\ v_1\ \dots\ v_n$ of a constant x to a sequence of values.

6.4 LOGSPACE-definable strategies

The Turing simulation sketched in Section 5.3 is generic relative to the evaluation strategy. For the space reasonability result to hold in the most general case, we are interested in "LOGSPACE-definable strategies", that we informally define as strategies for which the next redex can be identified using a space that is bounded by the space cost of the current term, up to a constant factor (in particular, the cost of the strategy will be bounded by a logarithm in the size of the input whenever we consider a computation that is LOGSPACE with respect to our cost measure).

Beyond call-by-name and call-by-value, examples of LOGSPACE-definable strategies are variations around the latter using restricted additional analyses. For instance: left-to-right call-by-value, with the additional rule that the evaluation of the argument is skipped if the formal parameter of the function does not appear in the function body, can indeed be implemented under the required space constraints.

There are also many ways of describing reduction strategies that are not LOGSPACE-definable. In particular, variations such as the above relying on more complex criteria are likely to provide a large hierarchy of counter-examples, by building finer and finer approximations of the optimal weak pure strategy, which is known to be undecidable [7]. Other counter-examples are strategies relying in an insufficiently restricted way on the history of the reduction (for instance, alternating between subterms in a recursively nested way), or strategies requiring additional machinery such as call-by-need.

6.5 Combining time-reasonability with space-reasonability

There is a well-known tension between time-reasonability and space-reasonability, which appears in the second simulation studied in this paper (implementing the λ-calculus on a Turing machine). This tension has been solved a first time with respect to ink space, using a space-aware interleaving machine which alternates between two simple machines respectively reasonable for time or space [16], and again recently with a simpler unified machine [9]. Both approaches seem compatible with our log-sensitive space cost measure, and the combination of either one with the machine built in Section 5 is expected to yield an implementation of the weak λ-calculus that is reasonable for both space and time, down to LOGSPACE.

7 Conclusion

This work shows that we can alter the "ink space" cost model of the weak λ-calculus to make it log-sensitive, by not counting the parts of terms that need not be written down since they can simply be described by a pointer to the input. This gives the first reasonable space cost model for the weak λ-calculus built directly in the core of the formalism and able to capture LOGSPACE complexity.

We borrow techniques from both previous lines [16] and [4, 5], and bring the new insight of the utility of the descendance relation in studying space for the λ-calculus. The obtained cost measure is more precise than the one in [16], enabling a characterization of LOGSPACE instead of only PSPACE and above. It is also more abstract than the one defined in [5], being expressed directly at the level of the λ-calculus with pure β-reduction and avoiding the indirection through a complex abstract machine relying on linear substitution. Finally, our cost measure is shown to be adequate for any LOGSPACE-definable weak reduction strategy, which is an improvement over both previous papers since [16] deals only with call-by-value, and [5] requires a specific machine for each strategy (the authors provide two such machines for call-by-value and call-by-name).

The main idea of this paper is quite flexible, and we mentioned several reasonable variants. On the other hand, some other extensions are less clear, and we leave two open questions.

- Could our cost model be captured by an intersection type system, similarly to what is achieved by [3] with respect to [5]? Given the recent successes of tightly capturing quantitative properties of reduction by non-idempotent intersection types, this is definitely worth investigating. It is not clear however whether the system from [3] would be a good starting point, since the two evaluation models differ significantly.
- Can our model be extended to strong reduction strategies? The definition of our cost measure contains nothing specific to weak reduction and is naturally extended. The current proof of reasonability however relies on weak reduction in several places. While some parts are easily generalizable, there are a few deeper reasons to this restriction, for instance the invariant in the proof of Lemma 5.17, the fact that the "deterministic" λ-calculus is deterministic only under weak reduction, or the (un)stability of the variable cost c_v along reduction, which is itself related to the presence or absence of α-renaming.

References

1 Beniamino Accattoli. The complexity of abstract machines. In Horatiu Cirstea and Santiago Escobar, editors, *Proceedings Third International Workshop on Rewriting Techniques for Program Transformations and Evaluation, WPTE@FSCD 2016, Porto, Portugal, 23rd June 2016*, volume 235 of *EPTCS*, pages 1–15, 2016. doi:10.4204/EPTCS.235.1.

2 Beniamino Accattoli and Ugo Dal Lago. (leftmost-outermost) beta reduction is invariant, indeed. *Log. Methods Comput. Sci.*, 12(1), 2016. doi:10.2168/LMCS-12(1:4)2016.

3 Beniamino Accattoli, Ugo Dal Lago, and Gabriele Vanoni. Multi types and reasonable space. *Proc. ACM Program. Lang.*, 6(ICFP), August 2022. doi:10.1145/3547650.

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

5 Beniamino Accattoli, Ugo Dal Lago, and Gabriele Vanoni. Reasonable space for the λ-calculus, logarithmically. *Log. Methods Comput. Sci.*, 20(4), 2024. doi:10.46298/LMCS-20(4:15)2024.

6 Sanjeev Arora and Boaz Barak. *Computational Complexity: A Modern Approach*. Cambridge University Press, 2009.

7 Thibaut Balabonski. Weak optimality, and the meaning of sharing. In Greg Morrisett and Tarmo Uustalu, editors, *ACM SIGPLAN International Conference on Functional Programming, ICFP'13, Boston, MA, USA - September 25 - 27, 2013*, pages 263–274. ACM, 2013. doi:10.1145/2500365.2500606.

8 Thibaut Balabonski. A Machine-Independent, Log-Sensitive Space-Cost Measure for the Weak Lambda-Calculus. working paper or preprint, October 2025. URL: https://hal.science/hal-05322360.

9 Thibaut Balabonski. Inlining as a Space Optimization: A Simple Time- and Space-Invariant Implementation of the Weak Lambda-Calculus. *Proc. ACM Program. Lang.*, Volume 10, Issue ICFP(307), August 2026. doi:10.1145/3828705.

10 Hendrik P. Barendregt. *The Lambda Calculus – Its Syntax and Semantics*. North-Holland Publishing Company, Amsterdam, revised edition, 1984.

11 Guy E. Blelloch and John Greiner. Parallelism in sequential functional languages. In John Williams, editor, *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.

12 Alonzo Church. An unsolvable problem of elementary number theory. *American Journal of Mathematics*, 58(2):345–363, 1936. URL: http://www.jstor.org/stable/2371045.

13 Alan Cobham. The intrinsic computational difficulty of functions. In Yehoshua Bar-Hillel, editor, *Logic, Methodology and Philosophy of Science: Proceedings of the 1964 International Congress (Studies in Logic and the Foundations of Mathematics)*, pages 24–30. North-Holland Publishing, 1965.

14 Haskell B. Curry, Robert Feys, and William Craig. *Combinatory logic, Volume I*. Studies in logic and the foundations of mathematics. North-Holland, Amsterdam, 1958.

15 Nicolaas G. de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the church-rosser theorem. *Indagationes Mathematicae (Proceedings)*, 75(5):381–392, 1972. doi:10.1016/1385-7258(72)90034-0.

16 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.

17 Gudmund S. Frandsen and Carl Sturtivant. What is an efficient implementation of the λ-calculus? In John Hughes, editor, *Functional Programming Languages and Computer Architecture*, pages 289–312, Berlin, Heidelberg, 1991. Springer Berlin Heidelberg.

18 John Glauert and Zurab Khasidashvili. Relative normalization in deterministic residual structures. In Hélène Kirchner, editor, *Trees in Algebra and Programming — CAAP '96*, pages 180–195, Berlin, Heidelberg, 1996. Springer Berlin Heidelberg. doi:10.1007/3-540-61064-2_37.

19 Juris Hartmanis and Richard E. Stearns. On the computational complexity of algorithms. *Transactions of The American Mathematical Society - TRANS AMER MATH SOC*, 117:285–285, May 1965. doi:10.2307/1994208.

20 John E. Hopcroft and Jeffrey D. Ullman. *Formal Languages and Their Relation to Automata*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1969.

21 Gérard Huet. Residual theory in λ-calculus: a formal development. *Journal of Functional Programming*, 4(3):371–394, 1994. doi:10.1017/S0956796800001106.

22 Jean-Louis Krivine. A call-by-name lambda-calculus machine. *Higher-Order and Symbolic Computation*, 20:199–207, 2007. doi:10.1007/S10990-007-9018-9.

23 Ugo Dal Lago and Beniamino Accattoli. Encoding turing machines into the deterministic lambda-calculus. *CoRR*, abs/1711.10078, 2017. arXiv:1711.10078.

24 Ugo Dal Lago and Simone Martini. The weak lambda calculus as a reasonable machine. *Theor. Comput. Sci.*, 398(1-3):32–50, 2008. doi:10.1016/J.TCS.2008.01.044.

25 Peter J. Landin. The mechanical evaluation of expressions. *The Computer Journal*, 6(4):308–320, January 1964. doi:10.1093/comjnl/6.4.308.

26 Julia L. Lawall and Harry G. Mairson. Optimality and inefficiency: What isn't a cost model of the lambda calculus? 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 92–101. ACM, 1996. doi:10.1145/232627.232639.

27 Jean-Jacques Lévy. Optimal reductions in the lambda calculus. *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, January 1980.

28 Christos H. Papadimitriou. *Computational Complexity*. Addison-Wesley, Reading, MA, 1994.

29 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.

30 Cees F. Slot and Peter van Emde Boas. On tape versus core; an application of space efficient perfect hash functions to the invariance of space. *J. Inf. Process. Cybern.*, 21(4/5):246–253, 1985.

31 Richard E. Stearns, Juris Hartmanis, and Philip M. Lewis. Hierarchies of memory limited computations. In *Proceedings of the 6th Annual Symposium on Switching Circuit Theory and Logical Design (SWCT 1965)*, FOCS '65, pages 179–190, USA, 1965. IEEE Computer Society. doi:10.1109/FOCS.1965.11.

32 Terese. *Term rewriting systems*, volume 55 of *Cambridge tracts in theoretical computer science*. Cambridge University Press, 2003.

33 Alan M. Turing. Computability and λ-definability. *Journal of Symbolic Logic*, 2(4):153–163, 1937. doi:10.2307/2268280.

34 Alan M. Turing. On computable numbers, with an application to the entscheidungsproblem. *Proceedings of the London Mathematical Society*, s2-42(1):230–265, January 1937. doi:10.1112/plms/s2-42.1.230.

35 Jan van Leeuwen, editor. *Handbook of Theoretical Computer Science, Volume A: Algorithms and Complexity*. Elsevier and MIT Press, 1990.

36 Jan van Leeuwen, editor. *Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics*. Elsevier and MIT Press, 1990. URL: https://www.sciencedirect.com/book/9780444880741/formal-models-and-semantics.