The Weak Call-by-Value λ-Calculus Is Reasonable for Both Time and Space

YANNICK FORSTER, Saarland University, Germany
FABIAN KUNZE, Saarland University, Germany
MARC ROTH, Saarland University, Germany, M2CI, Germany, and University of Oxford, UK

We study the weak call-by-value λ-calculus as a model for computational complexity theory and establish the natural measures for time and space — the number of beta-reduction steps and the size of the largest term in a computation — as reasonable measures with respect to the invariance thesis of Slot and van Emde Boas from 1984. More precisely, we show that, using those measures, Turing machines and the weak call-by-value λ-calculus can simulate each other within a polynomial overhead in time and a constant factor overhead in space for all computations terminating in (encodings of) *"true"* or *"false"*. The simulation yields that standard complexity classes like *P*, *NP*, *PSPACE*, or *EXP* can be defined solely in terms of the λ-calculus, but does not cover sublinear time or space.

Note that our measures still have the well-known size explosion property, where the space measure of a computation can be exponentially bigger than its time measure. However, our result implies that this exponential gap disappears once complexity classes are considered instead of concrete computations.

We consider this result a first step towards a solution for the long-standing open problem of whether the natural measures for time and space of the λ-calculus are reasonable. Our proof for the weak call-by-value λ-calculus is the first proof of reasonability (including both time and space) for a functional language based on natural measures and enables the formal verification of complexity-theoretic proofs concerning complexity classes, both on paper and in proof assistants.

The proof idea relies on a hybrid of two simulation strategies of reductions in the weak call-by-value λ-calculus by Turing machines, both of which are insufficient if taken alone. The first strategy is the most naive one in the sense that a reduction sequence is simulated precisely as given by the reduction rules; in particular, all substitutions are executed immediately. This simulation runs within a constant overhead in space, but the overhead in time might be exponential. The second strategy is heap-based and relies on structure sharing, similar to existing compilers of eager functional languages. This strategy only has a polynomial overhead in time, but the space consumption might require an additional factor of $\log n$, which is essentially due to the size of the pointers required for this strategy. Our main contribution is the construction and verification of a space-aware interleaving of the two strategies, which is shown to yield both a constant overhead in space *and* a polynomial overhead in time.

CCS Concepts: • **Theory of computation → Computational complexity and cryptography**; • **Mathematics of computing → Lambda calculus**.

Additional Key Words and Phrases: invariance thesis, lambda calculus, weak call-by-value reduction, time and space complexity, abstract machines

Authors' addresses: Yannick Forster, Saarland University, Saarland Informatics Campus, Saarbrücken, Germany, forster@ps.uni-saarland.de; Fabian Kunze, Saarland University, Saarland Informatics Campus, Saarbrücken, Germany, kunze@ps.uni-saarland.de; Marc Roth, Saarland University, Saarland Informatics Campus, Saarbrücken, Germany , M2CI, Saarbrücken, Germany , University of Oxford, Oxford, UK, marc.roth@merton.ox.ac.uk.

2475-1421/2020/1-ART27
https://doi.org/10.1145/3371095

ACM Reference Format:
Yannick Forster, Fabian Kunze, and Marc Roth. 2020. The Weak Call-by-Value λ-Calculus Is Reasonable for Both Time and Space. *Proc. ACM Program. Lang.* 4, POPL, Article 27 (January 2020), 23 pages. https://doi.org/10.1145/3371095

1 INTRODUCTION

Complexity classes like *P*, *NP*, *PSPACE*, or *EXP* are usually defined using Turing machines, which are the de-facto foundation of modern computability and complexity theory, in part due to the conceptual simplicity of their definition. However, this simplicity is also one of the biggest disadvantages: When it comes to detailed or formal reasoning, Turing machines soon become impossible to treat, because they lack compositionality and heavy logical machinery has to be used to reason about them. This is best reflected by the fact that modern day researchers in computability and complexity theory usually have not faced explicit Turing machines since their undergraduate studies. Instead, it is common to rely on pseudo code or mere algorithmic descriptions. For computability theory, other models of computation like RAM machines, recursive functions or variants of the λ-calculus can be used if details are of interest, because the notion of computation is invariant under changing the model. Especially the λ-calculus shines in this aspect, because tree-like inductive datatypes can be directly encoded and equational reasoning is accessible to verify the correctness of programs, which even makes the λ-calculus feasible as a model to formalise computability theory in proof assistants [Forster and Smolka 2017; Norrish 2011]. However, this notion of *invariance* does not suffice for complexity theory. As stated by Slot and van Emde Boas [1984]:

"Reasonable" machines can simulate each other within a polynomially bounded overhead in time and a constant factor overhead in space.

If only reasonable machines are considered, this invariance thesis makes complexity classes robust under changing the model of computation. Until now, only sequential models of computation have been shown to fulfil this strong notion of invariance with natural complexity measures for time and space [Dershowitz and Falkovich-Derzhavetz 2015]. For (idealised) functional languages, no reasonable, natural measures are known. The time and space complexity measures known to be reasonable for the full λ-calculus are "total ink used" and "maximum ink used" in a computation [Lawall and Mairson 1996]. Their notion for space is natural and in fact it is the space measure we will use too. Their notion for time however is very unnatural and not compositional, the combination of measures is thus of no real interest to define complexity classes. Other measures in the literature rely on concrete implementations, giving no satisfying answer to the question whether the λ-calculus can be considered reasonable.

Dal Lago and Martini [2008] gave a preliminary result in 2008 for the weak call-by-value λ-calculus and showed that counting β-steps while taking the size of β-redexes into account is a reasonable measure for time. In 2014 Accattoli and Dal Lago [Accattoli and Dal Lago 2016] showed that counting (leftmost-outermost) β-steps makes the full λ-calculus reasonable for time, starting a long line of research regarding measures for and implementations of the λ-calculus (see e.g. [Accattoli 2016]). Explicitly, they define a linear substitution calculus LSC featuring sharing and show that:

Theorem 11.1 in [Accattoli and Dal Lago 2016]: *There is an algorithm which takes as input a λ-term t and which, in time polynomial in the number of left-most outermost β-steps of t and the size of t outputs an LSC term u such that the unfolding of u is the normal form of t.*

This entails, in particular, that the normal form of a λ-term t whose output has constant size, which is always the case for decision functions, can be found in time polynomial in the number

of β-steps. The result shows that the full λ-calculus can be used to define time complexity classes like *P*, *NP*, or *EXP*. However, their result leaves open whether in conjunction the natural measure for space can also be used to define space complexity classes like *PSPACE*. We solve this problem for the deterministic weak call-by-value λ-calculus (which we call L) and show that the size of the largest intermediate term in a reduction makes L a reasonable machine in the strong sense, proving

Theorem 1.1. *There is an algorithm which takes as input a closed* L*-term t and which, in time polynomial in the number of β-steps of t and the size of t* and space linear in the size of the largest term in the reduction *outputs a heap containing a term u such that the unfolding of u is the normal form of t.*

Our result then suffices to define standard complexity classes like *P*, *NP*[1], or *PSPACE* (but not sublinear space classes like *L* or *NL*) in terms of L. Note that subpolynomial time classes like $O(n^2)$ are machine-dependent, and thus also do not agree for L and Turing machines. In total, our result is answering a question on complexity theory, namely whether the λ-calculus can be used to define time- and space complexity classes. For this goal, our measures work well and weak call-by-value evaluation is sufficient, especially considering that it is the standard model of eager functional programming languages.

Still, our measures leave room for improvement in at least two ways: First, since the size of the input is counted towards the space complexity of a term, our result does not cover sublinear time or space. We hope to resolve this problem in future work. Secondly, our result is only a very first step towards an answer to the question of what the *right* complexity measure for λ-calculi is. The space measure of a term t, i.e. the size of the largest term in the computation of t, can be exponentially larger than the time measure of t. Considering sequential models of computation, this property sounds counterintuitive. However, this is a property of *terms*, and not of problems. For a problem p (think *PRIME* or *SAT*), a decider t with this counterintuitive property can always be optimised to a decider t' which consumes space only polynomial in time. Thus, inclusions like $P \subseteq PSPACE$ still hold in L. In fact, the inclusion is a direct corollary of our result, we elaborate on this in more detail later.

1.1 Result and Discussion

We prove that the weak call-by-value λ-calculus is a reasonable machine with respect to the natural time and space measures, defined as follows:

Definition 1.1. *For a closed term s that reduces to a normal form $\lambda x.u$*

$$s = s_0 \succ s_1 \succ \cdots \succ s_k = \lambda x.u$$

we define the time consumption of the computation to be $\|s\|_T = k$ and the space consumption to be $\|s\|_S = \max_{i=0}^{k} \|s_i\|$ where $\|s\|$ is the size of s.

For the formal statement of reasonability we fix a finite alphabet Σ and say that a function $f : \Sigma^* \to \Sigma^*$ is *computable* by L in time $\mathcal{T}$ and space $\mathcal{S}$ if there exists a closed L-term s_f such that for all $x \in \Sigma^*$ we have that

$$s_f \ulcorner x \urcorner \succ^* \ulcorner f(x) \urcorner \text{ and } \left\| s_f \ulcorner x \urcorner \right\|_{\mathrm{T}} \leq \mathcal{T}(|x|) \text{ and } \left\| s_f \ulcorner x \urcorner \right\|_{\mathrm{S}} \leq \mathcal{S}(|x|)\,.$$

Here $\ulcorner \cdot \urcorner$ is an encoding of strings over Σ.

Theorem 1.2 (L is reasonable). *Let Σ be a finite alphabet such that $\{true, false\} \subseteq \Sigma$ and let $f : \Sigma^* \to \{true, false\}$ be a function. Furthermore, let $\mathcal{T}, \mathcal{S} \in \Omega(n)$.*

[1] using a verifier-based definition

(1) If f is L*-computable in time $\mathcal{T}$ and space $\mathcal{S}$, then f is computable by a Turing machine in time $O(poly(\mathcal{T}(n)))$ and space $O(\mathcal{S}(n))$.*
(2) If f is computable by a Turing machine in time $\mathcal{T}$ and space $\mathcal{S}$, then f is L*-computable in time $O(poly(\mathcal{T}(n)))$ and space $O(\mathcal{S}(n))$.*

The conditions $\mathcal{T} \in \Omega(n)$ and $\mathcal{S} \in \Omega(n)$ state that we do not consider sublinear time and space. Furthermore, the restriction of f to $\{true, false\}$ can be seen as a restriction to characteristic functions, which is sufficient for the complexity theory of decision problems.

To the best of our knowledge this is the first proof of the invariance thesis including time *and space* for a fragment of the λ-calculus using the natural complexity measures.

At this point the reader might have the following objection: A well-known problem in the λ-calculus is the issue of size explosion. There exist terms that reduce to a term of size $\Omega(2^n)$ with only $O(n)$ beta reductions. Let us adapt an example from Dal Lago and Martini [2008]: Given a natural number n, we write $\overline{n}$ for its Church-encoding, defined as

$$\overline{n} := \lambda f x.(\underbrace{f(f(f \cdots (f}_{n \text{ times}} x) \cdots)))$$

and define the Church encoding of the Boolean *true* as $\overline{true} := \lambda xy.x$. Next we define the term $s_E := \lambda x.\overline{true}\ \overline{true}\ (x\overline{2}(\lambda x.x))$. Note that the application $\overline{n}\,\overline{2}$ reduces to a normal form t_n of size $\Omega(2^n)$, extensionally equivalent to the Church-exponentiation $\overline{2^n}$. The term s_E thus encodes a function, computing an exponentially big intermediate result, discarding it and returning $\overline{true}$. Formally, we have

$$s_E\ \overline{n} \succ^4 (\lambda y.\overline{true})\ (\underbrace{\overline{2}(\overline{2} \ldots (\overline{2}}_{n \text{ times}}(\lambda x.x)))) \underbrace{\succ \ldots \succ}_{O(n) \text{ times}} (\lambda y.\overline{true})\ t_n \succ \overline{true}$$

for a term t_n with $\|t_n\| \in \Omega(2^n)$. Now $\|s_E\ \overline{n}\|_{\mathrm{T}} \in \Theta(n)$, i.e., $s_E\ \overline{n}$ reduces to $\overline{true}$ in about n beta reductions. Moreover, $\|s_E\ \overline{n}\|_{\mathrm{S}} \geq \left\|(\lambda y.\overline{true})\ t_n\right\| \in \Omega(2^n)$, i.e., the largest term in the reduction is of exponential size. While it might seem counterintuitive that a reasonable machine allows a computation that requires much more space than time, which is impossible for Turing machines, we consider this as one of the major insights of this work: There are concrete computations exhibiting size explosion, but as long as these computations return values of bounded size, they can always be optimised, i.e. the size explosion can be shown unnecessary to compute the result of the term. Of course, no reduction sequence that *ends* in a term of exponential size can be simulated in less space if the result has to be written down explicitly. However, in complexity theory of decision problems, the functions that matter in the end are characteristic functions which map to *true* and *false*, and thus we still have e.g. $P \subseteq PSPACE$.

The proof for the inclusion works as follows: Let the decision problem p be polynomial, i.e. $p \in P$ and s_E be a (potentially size-exploding) decider for p. We can use Theorem 1.2 twice to obtain a term $\widehat{s_E}$ which still decides p, but now with polynomial space usage: By Theorem 1.2 (1) there is a Turing machine that on an encoding of $\overline{n}$ simulates $s_E\ \overline{n}$ with time and space complexity of $O(n^c)$ (the latter since Turing machines can not use more space than time). By Theorem 1.2 (2), there is a term $\widehat{s_E}$ s.t. $\widehat{s_E}\ \overline{n}$ has the same normal form as $s_E\ \overline{n}$ – but with space complexity $O(n^c)$, since the overhead in space is constant-factor.

In [Accattoli 2018], Accattoli writes *"Essentially one is assuming that space in the λ-Calculus is given by the maximum size of terms during evaluation, and since in sequential models time is greater or equal to space (because one needs a unit of time to use a unit of space), the time cost of the size exploding family must be at least exponential. The wrong hypothesis then is that space is the maximum size of the terms during evaluation. It is not yet clear what accounts for space in the λ-Calculus, however*

the study of time cost models made clear that space is not the size of the term." Our result implies that this conclusion does not apply for weak call-by-value evaluation in L. In this particular case, the wrong hypothesis is that even when simulating on Turing machines, where time is greater or equal to space, the space-measure of a λ-calculus term does not have to coincide with the actual resources used when simulating it and can thus be much larger than the time-measure. Taking maximum size of terms during evaluation as space measure, while possibly counterintuitive, is thus not contradictory with respect to reasonability.

1.2 Simulation Strategies

In the previous paragraph we argued that our proposed cost and time measures for L are not inherently contradictory with respect to the invariance thesis. However, we did not provide explicit information of the simulations yet, which we are going to catch up on now. Note that a simulation of Turing machines in L for the second part of Theorem 1.2 regarding time has already been given by Accattoli and Dal Lago [2012, 2017]. We argue in Section 5 that their construction also only has a constant factor overhead in space and thus works for our purposes as well. The main part of the paper thus focuses on the simulation of L by Turing machines. Essentially, we rely on an interleaving of two different strategies for simulating a reduction in L with a Turing machine. Both strategies are formally introduced in Section 3; in what follows, we provide an intuitive overview.

The first one, which we call the *substitution-based strategy* simulates a reduction sequence naively as given by the reduction rules of L. In particular, all substitutions are executed immediately if a β-reduction is performed. However, we have already seen an example which shows this strategy to be insufficient: Consider again the term $s_E\ \overline{n}$ which reduces to $\overline{true}$ in $\Theta(n)$ beta reductions. If this reduction sequence is simulated naively, exponentially many substitutions have to be performed, and hence the time consumption of that machine would be exponential in n. At the same time, for this term, the strategy is valid if we would only care for space, because the space complexity of the L term is already exponential. In general, we show that any reduction sequence in L can be simulated with only a constant overhead in space by a Turing machine using the substitution-based strategy.

Solving the issue regarding the time consumption requires us to rely on the second simulation strategy which we call the *heap-based strategy*. Intuitively, we do not execute any substitution if a β-reduction is simulated. Instead we use *closures* and keep track of the values assigned to variables in an environment. These environments are stored on an explicit heap containing pointers and terms. This allows for *structure sharing*, similar to real-world execution of functional languages as well as to the strategy used by Accattoli and Dal Lago [2016]. Indeed, applying this strategy to the reduction sequence $s_E\ \overline{n} \succ \ldots \succ \overline{true}$ yields a polynomial number of steps in a simulation with a Turing machine.

At this point, one might be tempted to think that the heap-based strategy is strictly superior to the substitution-based strategy. However, there is one (major) catch: There exist reduction sequences of time and space linear in the input term size n, which yield an overhead of factor $\log n$ in space when simulated using the heap-based strategy. The reason is that the number of heap entries is linear, which requires the pointers, i.e. the heap addresses, to grow in size. The following example illustrates this phenomenon: Let $\mathrm{N} := (\lambda xy.xx)\overline{true}$.

$$s_P := \underbrace{\mathrm{N}(\cdots(\mathrm{N}\,\overline{true})\ldots)}_{n \text{ times}} \succ^n \underbrace{(\lambda y.\overline{true}\,\overline{true})(\cdots((\lambda y.\overline{true}\,\overline{true})\,\overline{true})\ldots)}_{n \text{ times}} \succ^{2n} \overline{true}$$

Since s_P performs $3n$ beta reductions it needs $3n$ entries on the heap. The heap pointers then make the space consumption "explode" again. They are of size $\log n$ if binary numbers are used and n if unary numbers are used, resulting in an overall space consumption of $\Omega(n \log n)$ or $\Omega(n^2)$, both

forming more than constant factor overhead. We call this problem *pointer explosion*, analogous to the discussed size explosion problem, and point out that both phenomena have already been identified and discussed by Slot and van Emde Boas [1984] in their treatment of RAM simulations by Turing machines.

In our case of the weak call-by-value λ-calculus L, we have obtained two simulation strategies, each solving *one* of the problems: The substitution-based strategy works for space, but is insufficient for time on terms exhibiting size explosion (i.e. which have exponentially big intermediate terms). The heap-based strategy works for time, but is insufficient for space on terms exhibiting pointer explosion. The crucial observation is now that on terms exhibiting size explosion, i.e. reaching a term of size $\Omega(2^n)$ in n steps, pointer explosion is a non-issue: n pointers of size $\log n$ can easily be accommodated for in space $O(2^n)$.

Since it is a-priori not decidable whether a term exhibits size explosion or pointer explosion, we interleave the execution of the two simulation strategies. We simulate the execution for every step number k repeatedly, and always try to run the substitution-based strategy first. If the size of intermediate terms becomes big enough to accommodate exploding pointers, we immediately abort and try the heap-based strategy for k steps instead. The heap-based strategy is thus guaranteed to not encounter the pointer explosion problem and the substitution-based strategy can not encounter the size explosion problem, because it is aborted beforehand. The details of this interleaving machine, which we consider as our main technical contribution, are given in Section 4.

1.3 Formalisation in Coq

A technically demanding and error-prone part of our proof is analysing the exact properties of the machines involved.

Because our proofs rely on many simulation notions containing hard-to-check side conditions, we provide a mechanisation of all results for the abstract machines in Section 3 in The Coq Proof Assistant [2019].[2]

A mechanisation of all results, including a formal verification of the Turing machines involved, is an ongoing and challenging project. Turing machines are very hard to mechanise, bordering entire infeasibility. One needs to introduce abstractions to make Turing machines feasible for mechanisation, but existing abstractions do likely not scale to the space-aware interleaving machines we need. In [Forster et al. 2019] the first two authors of this paper together with Maximilian Wuttke report on a framework enabling the mechanisation of a universal Turing machine with polynomial time and constant factor space overhead, based on prior work by Asperti and Ricciotti [2015] in Matita. They however acknowledge that more powerful abstractions and tools are necessary to be able to mechanise our space-aware interleaving simulation of L.

We reported on both the theoretical contributions and the mechanisation of our project previously in [Forster et al. 2017]; the current paper presents the finalised theoretical contributions.

2 PRELIMINARIES

We adopt a notation closely related to type-theory, but the paper can be read with no background in type theory. All defined functions are always total. We use the type $X_\bot$ to denote the type X enriched with a new element $\bot$. This allows us to view $X \to Y_\bot$ as the type of partial functions from X to Y. In the definition of such partial functions, left out cases are meant to default to $\bot$. Concerning lists $A, B : X^*$ over X, we use [] for the empty list, write $x :: A$ to prepend an element

[2]The code is hyperlinked with the PDF version of this document and can be accessed at https://github.com/uds-psl/cbv-lambda-calculus-reasonable or by clicking on the formalised statements and definitions, which are marked with a 🐓-symbol.

to a list and write $[x_1, \ldots, x_k]$ for a list built from the elements x_i. We write $x \in A$ if x is an element of the list A, $A + B : X^*$ for list concatenation, $|A| : \mathbb{N}$ for length and $A[n] : X_\bot$ for list lookup.

2.1 Call-by-Value λ-Calculus L

The call-by-value λ-calculus introduced by Plotkin [1975] in his seminal paper is known to be a reasonable machine for time complexity [Dal Lago and Martini 2008]. In those works, abstractions and variables are treated as values, but β-reduction below binders is not allowed, i.e. reduction is *weak*. We use a deterministic version of the weak call-by-value λ-calculus we call L, originally introduced by Forster and Smolka [2017]. We treat only abstractions as values but keep the weak behaviour of reduction. On closed terms, the number of steps to a normal form agrees with the number of steps needed in the version by Dal Lago and Martini [2008]. We keep the definitions short and use the same notations as in [Kunze et al. 2018], where more details can be found.

We define the syntax of the λ-calculus using a de Bruijn representation of terms [Bruijn 1972]: $s, t, u, v \ : \ \mathsf{Ter} \ ::= \ n \mid st \mid \lambda s$ where $n : \mathbb{N}$.

Definition 2.1. *We define a recursive function s_u^k providing a single-point, capturing* substitution *operation:*

$$k_u^k \ := \ u \qquad\qquad n_u^k \ := \ n \qquad \text{if } n \neq k$$
$$(st)_u^k \ := \ (s_u^k)(t_u^k) \qquad\qquad (\lambda s)_u^k \ := \ \lambda(s_u^{1+k})$$

We say that s is bounded by k if all free de Bruijn indices in s are lower than k. Consequently, s is a closed term iff it is bounded by 0.

Definition 2.2. *We define a deterministic inductive* reduction relation $s \succ t$, *which is weak, call-by-value and agreeing with the reduction relation in [Dal Lago and Martini 2008; Plotkin 1975] on closed terms:*

$$\frac{}{(\lambda s)(\lambda t) \succ s_{\lambda t}^0} \qquad \frac{s \succ s'}{st \succ s't} \qquad \frac{t \succ t'}{(\lambda s)t \succ (\lambda s)t'}$$

Note that the only closed, irreducible terms are abstractions. We write $s \Downarrow_k^{\mathrm{T}} t$ and $s \Downarrow_m^{\mathrm{S}} t$ if $s \succ^k t$ for t being an abstraction and $m = \|s\|_{\mathrm{S}}$ as defined in Definition 1.1.

The size of a term is defined with a unary encoding of indices in mind:

$$\|n\| \ := \ 1 + n \qquad\qquad \|\lambda s\| \ := \ 1 + \|s\| \qquad\qquad \|st\| \ := \ 1 + \|s\| + \|t\|$$

For a binary encoding, i.e. $\|n\| \ := \ 1 + \log_2 n$, we conjecture that the remainder of this paper can be adapted with no essential change.

2.2 Encoding Terms as Programs

Turing machines can not directly operate on tree-like data structures like L-terms. We encode terms as *programs* $P, Q, R \ : \ \mathsf{Pro}$, which are lists of *commands*:

$$c \ : \ \mathsf{Com} \ ::= \ \mathsf{ret} \mid \mathsf{var}\, n \mid \mathsf{lam} \mid \mathsf{app} \qquad \text{with } n : \mathbb{N}$$

Definition 2.3. *The encoding function $\gamma : \mathsf{Ter} \to \mathsf{Pro}$ compiles terms to programs:*

$$\gamma n \ := \ [\mathsf{var}\, n] \qquad \gamma(st) \ := \ \gamma s + \gamma t + [\mathsf{app}] \qquad \gamma(\lambda s) \ := \ [\mathsf{lam}] + \gamma s + [\mathsf{ret}]$$

This encoding is similar to postfix notation, the additional command lam makes it easier to detect subprograms representing values when traversing the encoding.

Definition 2.4. *We write $P \gg s$, read as P* represents *s, to connect programs with values in* L. *This relation is defined with the single rule $\gamma t \gg \lambda t$.*

To store the encoding of de Bruijn indices on tapes, we will use a unary encoding, motivating the following definition of the size of commands and programs:

$$\|\mathsf{var}\, n\| := 1 + n \qquad \|c\| := 1 \quad \text{if } c \text{ is not a variable} \qquad \|P\| := 1 + \sum_{c \in P} \|c\|$$

This size is compatible with term size, with factor 2 due to the two commands for abstractions:

Lemma 2.1. $1 \leq \|s\| \leq \|\gamma s\| \leq 2\,\|s\| - 1$

The use of lam and ret to encode abstraction allows to define a function $\varphi : \mathsf{Pro} \to (\mathsf{Pro} \times \mathsf{Pro})_\perp$ that extracts the body of an abstraction by matching lam with ret like parentheses. It uses an auxiliary function $\varphi_{k,Q}P$ that stores the number k of currently unmatched lam and the prefix Q already processed.

Definition 2.5. $\varphi P := \varphi_{0,P}[]$ *with*

$$\varphi_{0,Q}(\mathsf{ret} :: P) := (Q, P) \qquad \varphi_{1+k,Q}(\mathsf{ret} :: P) := \varphi_{k,Q+[\mathsf{ret}]}P$$
$$\varphi_{k,Q}(\mathsf{lam} :: P) := \varphi_{1+k,Q+[\mathsf{lam}]}P \qquad \varphi_{k,Q}(c :: P) := \varphi_{k,Q+[c]}P \text{ if } c = \mathsf{var}\, n \text{ or app}$$

Lemma 2.2. $\varphi(\gamma s + \mathsf{ret} :: P) = (\gamma s, P)$

We define a substitution operation P^k_Q on programs, analogous to substitution on terms

Definition 2.6 (Substitution on programs).

$$(\mathsf{var}\, k :: P)^k_Q := Q :: P^k_Q \qquad (\mathsf{var}\, n :: P)^k_Q := \mathsf{var}\, n :: P^k_Q$$
$$(\mathsf{lam} :: P)^k_Q := \mathsf{lam} :: P^{Sk}_Q \qquad (\mathsf{app} :: P)^k_Q := \mathsf{app} :: P^k_Q$$
$$(\mathsf{ret} :: P)^0_Q := [\mathsf{ret}] \qquad (\mathsf{ret} :: P)^{Sk}_Q := \mathsf{ret} :: P^k_Q$$
$$[]^k_Q := []$$

Term and program substitution are compatible:

Lemma 2.3. $(\gamma s)^0_{\gamma t} = \gamma(s^0_t)$

2.3 Closures and Heaps

To allow for structure sharing later, we introduce closures whose environments are stored in an explicitly modelled heap. Environments are stored as linked lists of closures on the heap, and closures $g : \mathsf{HC} := \mathsf{Pro} \times \mathsf{HA}$ contain programs and pointers $a, b : \mathsf{HA} := \mathbb{N}$ to the environment. We represent the heap $H : \mathsf{Heap} := \mathsf{HE}^*$ as a list of its cells $e : \mathsf{HE} ::= (g, a)$ that store the head and the address of the tail. To interpret the linked list structure, we define a *lookup function* $\mathsf{H[a, n]} : \mathsf{HC}_\perp$ that returns the n-th entry of the list at address a, i.e. the value bound to the de Bruijn index n in the environment a:

$$H[a, n] := \texttt{if } n = 0 \texttt{ then } g \texttt{ else } H[b, n-1] \qquad \text{where } H[a] = (g, b)$$

The operation $\mathsf{put\ H\ e} : \mathsf{Heap} \times \mathsf{HA}$ puts a heap entry on the heap and returns the new heap and the address of the new element.

$$\mathsf{put}\ H\ e := (H + [e],\ |H|)$$

In our setting, we allocate at the end of the heap and have no need for garbage collection.

A closure (P, a) represents some term if the environment a contains bindings for all free variables in P. To make this more precise, we first define the unfolding of a term relative to some environment:

$$(\mathsf{lam} :: P) :: T,\ V \ \succ\ P' ::_{\mathrm{tc}} T,\ Q :: V \qquad \text{if } \varphi P = (Q, P')$$
$$(\mathsf{app} :: P) :: T, Q :: R :: V \ \succ\ R^0_{\mathsf{lam}::Q + [\mathsf{ret}]} :: P ::_{\mathrm{tc}} T,\ V$$
$$\text{where } P ::_{\mathrm{tc}} T \ :=\ \mathtt{if}\ P = [\,]\ \mathtt{then}\ T\ \mathtt{else}\ P :: T$$

Fig. 1. Reduction rules of the substitution machine

Definition 2.7 (Unfolding). *The unfolding $s\langle a, k\rangle \downarrow_H s'$ is inductively given by the rules*

$$\frac{n < k}{n\langle a, k\rangle \downarrow_H n} \qquad \frac{n \geq k \qquad H[a, n-k] = (P, b) \qquad P \gg s \qquad s\langle b, 0\rangle \downarrow_H s'}{n\langle a, k\rangle \downarrow_H s'}$$

$$\frac{s\langle a, 1+k\rangle \downarrow_H s'}{\lambda s\langle a, k\rangle \downarrow_H \lambda s'} \qquad \frac{s\langle a, k\rangle \downarrow_H s' \qquad t\langle a, k\rangle \downarrow_H t'}{st\langle a, k\rangle \downarrow_H s't'}$$

Intuitively, $s\langle a, k\rangle \downarrow_H s'$ holds if s' is obtained by recursively substituting all free variables in s by their values in the environment a. The index k is an artefact of the de Bruijn representation and denotes which variables in s are locally bound during the traversal of s.

The first rule states that bound variables are left unchanged. The second rule states that for free variables, the environment a binds n to some value s' that can be looked up in H. The third rule descends under an abstraction and thus one more variable is considered bound in s. The last rule descends under application.

Definition 2.8. *The relation $g \gg_H s$, read as g* represents *s relative to H, is defined by the single rule*

$$\frac{P \gg t \qquad t\langle a, 0\rangle \downarrow_H s}{(P, a) \gg_H s}$$

3 ABSTRACT MACHINES

In order to analyse the two mentioned strategies on a more semantic level than just as implementations on Turing machines we introduce two abstract machines implementing these strategies – based on substitutions and based on heaps. The machines are variants of the ones presented in [Kunze et al. 2018]. Both machines will take $O(\|s\|_{\mathrm{T}})$ abstract steps to evaluate a term s, but differ in the size of intermediate states and in the complexity of their respective implementations as Turing machines, which we construct in Section 4.

3.1 Substitution Machine

We define an abstract machine that uses substitution on programs. The implemented strategy is close to the small-step semantics for L. One important property is that the size of machine states during the machine run is linear in the size of the intermediate terms. Therefore, the substitution-based Turing machine will have constant factor overhead for space.

The abstract machine executes terms using two stacks of programs T and V called *task* and *value* stack. The task stack holds the parts of the program yet to be executed, and the value stack holds the already fully evaluated parts.

The semantics of the substitution machine is defined in Figure 1. The machine executes the first command of the topmost program of the task stack. In the *lambda rule*, the command lam marks the start of an abstraction. The sub-program corresponding to the body of the abstraction is moved to the value stack. In the *application rule*, the topmost values are applied to each other: The program R is instantiated with the argument Q to obtain a new task to be evaluated.

We need tail call optimisation $::_{\mathrm{tc}}$ to guarantee that the size of the machine state is linear in the size of the represented term. Without it, the application rule could pile up return-tasks $P = []$ inside the task stack, invalidating Lemma 3.2. While we only need tail call optimisation for the application rule, adding it to the lambda rule as well streamlines proofs and allows us to avoid a rule to discard empty programs. The *initial state* τ_s for a term s is $\tau_s := ([\gamma s], [])$.

The machine evaluates L with a number of steps linear in the time-measure:

Lemma 3.1 (Substitution machine runtime). *If $s \Downarrow_k^T t$, then $\tau_s \succ^{3k+1} ([], [P])$ for some P with $P \gg t$.*

The size $\|T\|$ of T is defined to be just the sum of the sizes of the elements in T, and similar for V. The size of a state is defined by $\|(T, V)\| = \|T\| + \|V\|$. We write $\tau \succ_m^* \tau'$ for a sequence of machine reductions where the largest state has size m.

The maximal machine state size when evaluating s is asymptotically as large as $\|s\|_S$.

Lemma 3.2 (Substitution machine state size). *If $s \Downarrow_m^S t$, then $\tau_s \succ_{m'}^* ([], [P])$ for some P and m' with $P \gg t$ and $m \leq m' \leq 2m$.*

$$
\begin{aligned}
(\mathrm{var}\, n :: P, a) :: T,\ V,\ H &\succ (P, a) :: T,\ g :: V,\ H && \text{if } H[a, n] = g \\
(\mathrm{lam} :: P, a) :: T,\ V,\ H &\succ (P', a) :: T,\ (Q, a) :: V,\ H && \text{if } \varphi P = (Q, P') \\
(\mathrm{app} :: P, a) :: T,\ g :: (Q, b) :: V,\ H &\succ (Q, b') :: (P, a) :: T,\ V,\ H' && \text{if } \mathrm{put}\, H\, (g, b) = (H', b') \\
([], a) :: T,\ V,\ H &\succ T,\ V,\ H
\end{aligned}
$$

Fig. 2. Reduction rules of the heap machine

3.2 Heap Machine

This machine uses the heap described in Subsection 2.3 to enable sharing of environments. One important feature of this machine is that the size of intermediate states only depends on the size of the initial term s and the number of machine steps, but not on $\|s\|_S$.

The machine is defined in Figure 2. Its task and value stacks contain closures. The *variable rule* loads the value bound to a variable to the value stack. The *lambda rule* copies a subprogram representing an abstraction to the value stack. The *application rule* calls the subprogram Q after adding the value g as argument to the environment of Q. The *return rule* drops finished tasks. The use of closures instead of programs allows an explicit variable rule instead of program-level substitution. The *initial state* σ_s for a closed term s is $([(\gamma s, 0)], [], [])$ as $s\langle 0, 0\rangle \downarrow_H s$ by Lemma B.3.

The machine evaluates L with a number of steps linear in the time-measure:

Lemma 3.3 (Heap machine runtime). *If $s \Downarrow_k^T t$ and s is closed, then $\sigma_s \succ^{4k+2} ([], [g], H)$ for some g, H with $g \gg_H t$.*

We define the size of a closure as $\|(P, a)\| := \|P\| + a$ and the size of a heap entry to be $\|(g, a)\| := \|g\| + a$. The size of a state is the sum of the sizes of all elements in T, V and H.

The size of the k-th state starting from σ_s is a polynomial in k and $\|s\|$:

Lemma 3.4 (Heap machine state size). *If $\sigma_s \succ^k \sigma$, then $\|\sigma\| \leq (k+1)(3k + 4\,\|s\|)$*

4 SIMULATING L WITH TURING MACHINES

We now sketch how to construct the Turing machine that simulates L with polynomially bounded overhead in time and constant factor overhead in space. We construct those machines by implementing Turing machines that compute a single step of the abstract machines from Section 3, and

then looping those machines, thus essentially implementing the rewriting systems from Figure 1 and Figure 2. The considered Turing machines will operate on various kinds of data (e.g. natural numbers, programs, heap closures, heap entries, heaps, ...). For programs, we use a symbol for each of the four constructors and a fifth symbol to encode de Bruijn indices in unary. All other natural numbers will also be encoded in unary, unless explicitly stated. The encoding of the further structures on tapes is straightforward.

4.1 The Substitution-Based Turing Machine Simulating L

We construct a Turing machine M_{subst} that executes the substitution-based strategy from Subsection 3.1 for k steps, where k is an input. The Turing machine takes an additional input m and aborts if the abstract machine would reach a state of size greater than m.

Theorem 4.1. *There is a Turing machine M_{subst} that, given two binary numbers k, m and a term s, halts in time $O(k \cdot \mathit{poly}(\min(m, \|s\|_S)))$ and space $O(\min(m, \|s\|_S) + \log m + \log k)$ s.t. one of the following holds:*

- *The machine outputs a term t, then s has normal form t and $m \geq \|s\|_S$ and $k \geq 3 \cdot \|s\|_T + 1$.*
- *The machine halts in a state named* space bound reached *and $m \leq 2 \cdot \|s\|_S$ holds.*
- *The machine halts in a state named* space bound not reached *and $k < 3 \cdot \|s\|_T + 1$ holds.*

Proof. The Turing machine can be constructed by iterating the rules of the abstract substitution machine from Figure 1 on the initial state τ_s. The machine has to keep track of the size of the abstract machine state, even *during* the execution of the substitution: As soon as the size of the next state to be computed is known to exceed m, it aborts *before* consuming more than $\Theta(m)$ space. This is necessary because the result of a substitution P^0_Q with $\|P\| + \|Q\| \in O(m)$ could have quadratic size $O(m^2)$, e.g. if P applies the variable 0 to itself m times and Q has size m as well. The function φP can be implemented via the tail-recursive $\varphi_{k,Q} P$, which takes space and time $O(\|Q\| + k + \|P\|)$, as it just traverses P and accumulates the result. The argument k during the run is bounded by $\|s\|_S$. Then the size of all intermediate states and the overall space consumption follow from Lemma 3.2. The existence of the result for large enough k follows in combination with Lemma 3.1. □

The precise specification of the machine is subtle: Intuitively, the machine state size is as large as the 'current' term, but we don't know if a state larger m is reached in the first k steps. Therefore, we don't specify which of the last two cases occurs if *both* bounds on k and m are exceeded.

If s diverges, Theorem 4.1 states that M_{subst} can only halt in the two special final states (with $\|s\|_T = \infty$ for diverging terms s).

4.2 The Heap-Based Turing Machine Simulating L

We construct a Turing machine executing the heap-based strategy from Subsection 3.2 for k steps:

Theorem 4.2. *There is a Turing machine M_{heap} that, given a number k and a closed term s, halts in time $O(\mathit{poly}(\|s\|, k))$ and space $O(\|s\| \cdot \mathit{poly}(k))$. If s has a normal form t and $k \geq 4 \cdot \|s\|_T + 2$, it computes a heap H and a closure g such that $g \gg_H t$. Otherwise, it halts in a distinguished final state (denoting* 'failure'*).*

Proof. The Turing machine can be constructed by iterating the rule of the abstract substitution machine on the initial state σ_s. We already argued on the runtime of φ for Theorem 4.1. And $H[a, n]$ can be computed by iterating over H for at most n times. So each abstract step $(T, V, H) \succ (T', V', H')$ can be implemented in time $O(\text{poly}(\|(T, V, H)\|))$ and space

$$O(\max(\|(T, V, H)\|, \|(T', V', H')\|)).$$

The space consumption of all involved operations in Figure 2 is bounded by their input or output. Using Lemma 3.4, the size of all intermediate (T, V, H) can be bound by k and $\|s\|$ to derive the claimed resource bounds. The successful computation of g and H for large enough k follows with Lemma 3.3. □

4.3 The Combined Turing Machine Simulating L

We now combine the machines from the last two sections to execute the heap-machine only if we know that its space consumption is bounded by the space measure of the simulated term:

Theorem 4.3. *There is a Turing machine M_{L} that, given a closed term s that has a normal form t, computes a heap H and a closure g such that $g \gg_H t$ in time $O(poly(\|s\|, \|s\|_T))$ and space $O(\|s\|_S)$.*

Proof. Let p be the polynomial such that the machine from Theorem 4.2 runs in space $O(\|s\| \cdot p(k))$. Then the combined machine executes the following algorithm:

(1) Initialise $k := 0$ (in binary)
(2) Compute $m := \|s\| \cdot p(k)$ (in binary)
(3) Run M_{subst} on s, k and m.
   - If M_{subst} computes the normal form t, output $(\gamma t, 0)$ and an empty heap [] and halt.
   - If M_{subst} halts with *space bound not reached*, set $k := k + 1$ and go to **2**.
   - If M_{subst} halts with *space bound reached*, continue at **4**.
(4) Run M_{heap} on s and k.
   - If this computed a closure g and a heap H representing t, output H and g and halt.
   - Otherwise, set $k := k + 1$ and go to **2**.

First, we show that if this machine halts, its output is a closure-heap pair representing the normal form t of s: If the machine halts during **3**, the output is a representation of the normal form by Theorem 4.1 and Lemma B.3. If it halts during **4**, it does so by Theorem 4.2.

Second, we analyse termination and the time complexity of this machine. As intermediate step, we analyse the run time for a fixed k. Step **2** takes time $O(\mathrm{poly}(\|s\|, k))$, and the size of s can be computed from its encoding in straightforward fashion. Using Theorem 4.1, Step **3** takes time

$$\begin{aligned} O(k \cdot \mathrm{poly}(\min(m, \|s\|_{\mathrm{S}}))) &\subseteq O(k \cdot \mathrm{poly}(m)) \\ &= O(k \cdot \mathrm{poly}(\|s\| \cdot p(k))) \\ &\subseteq O(k \cdot \mathrm{poly}(\|s\|, k)) && p \text{ is a polynomial} \\ &\subseteq O(\mathrm{poly}(\|s\|, k)) \end{aligned}$$

If Step **4** is executed, this takes time $O(\mathrm{poly}(\|s\|, k))$ by Theorem 4.2. This means for arbitrary k, one iteration of the described algorithm can be computed in time $O(\mathrm{poly}(\|s\|, k))$.

The algorithm will eventually halt: We consider $k = 4\|s\|_{\mathrm{T}} + 2$, which is larger than the two values required in Theorem 4.1 and Theorem 4.2: By Theorem 4.1, the machine does halt during Step **3**, unless $m < \|s\|_{\mathrm{S}}$. In the latter case, **4** is tried. Then, by Theorem 4.2, as k is large enough, we have that M_{heap} indeed halts with a closure-heap pair.

Summing up the run time of each iteration, we have that the machine terminates in time

$$O\left(\sum_{k=0}^{4\|s\|_{\mathrm{T}}+2} (\mathrm{poly}(\|s\|, k))\right) \subseteq O(\|s\|_{\mathrm{T}} \cdot (\mathrm{poly}(\|s\|, \|s\|_{\mathrm{T}}))) \subseteq O(\mathrm{poly}(\|s\|, \|s\|_{\mathrm{T}}))$$

Third, we analyse the space complexity of this machine. Again, we first analyse one iteration for a fixed k. Step **2** takes space $O(\log(m))$, since we use binary numbers. By Theorem 4.1, Step **3** takes space $O(\min(m, \|s\|_{\mathrm{S}}) + \log m + \log k) \subseteq O(\|s\|_{\mathrm{S}} + \log m + \log k)$. If Step **4** is executed, then

$m < \|s\|_S$. By Theorem 4.2, this step runs in space $O(m) \subseteq O(\|s\|_S)$. So, we can compute the space consumption of a single iteration as:

$$\begin{aligned}
&O(\log m + \log k + \|s\|_S) \\
&= O(\log(\|s\| \cdot p(k)) + \log k + \|s\|_S) && \text{definition } m \\
&\subseteq O(\log \|s\| + \log(p(k)) + \log k + \|s\|_S) \\
&= O(\log(p(k)) + \log k + \|s\|_S) && \text{as } \|s\| \leq \|s\|_S \\
&\subseteq O(\log k + \|s\|_S) && \log(p(k)) \in O(\log k) \text{ as } p \text{ polynomial}
\end{aligned}$$

Overall, we have that the whole machine runs in space (the last equation is by Lemma 4.1):

$$O\left(\max_{0 \leq k \leq 4\|s\|_T+2} (\log k + \|s\|_S)\right) \subseteq O(\log \|s\|_T + \|s\|_S) = O(\|s\|_S) \quad \square$$

Note that the machine only terminates for terminating terms, making this a full simulation also for diverging terms. For terms with $\|s\|_T \notin O(\|s\|_S)$ it is crucial that the machine tracks the step number k in binary, because it would need $\Omega\|s\|_T$ space otherwise. This suffices due to the following theorem:

Lemma 4.1. $\log \|s\|_T \in O(\|s\|_S)$.

Proof. The main insight is that for any given size, there are only exponentially many terms smaller than that size. As reduction is deterministic, a terminating term s can not contain the same intermediate term twice. This bounds $\|s\|_T$ by the number of terms with size smaller than $\|s\|_S$, i.e. $\|s\|_T \leq c^{\|s\|_S}$ for a constant c.

Now, we show that the number of terms smaller than a certain size m is an exponential. We use the encoding γ to allow us to count linear strings (programs) instead of trees (terms):

$$\begin{aligned}
&\#\{t \mid \|t\| \leq m\} \\
&= \#\{\ t \mid 2 \cdot \|t\| \leq 2 \cdot m\} \\
&\leq \#\{t \mid \|\gamma t\| \leq 2 \cdot m\} && \text{Lemma 2.1} \\
&= \#\{\gamma t \mid \|\gamma t\| \leq 2 \cdot m\} && \text{Lemma B.1} \\
&\leq \#\{P \mid \|P\| \leq 2 \cdot m\} \\
&\leq 5^{2 \cdot m}
\end{aligned}$$

In the last step, we use that $\#\{P \mid \|P\| \leq n\} \leq 5^{n-1}$ for all $n > 0$ by induction on n, where the intuition behind the 5 is that there are four different symbols with which a program can start, and that variables require a fifth symbol to encode the index in unary. Thus the claim holds for $c = 5^2$. $\square$

Note that this suffices to prove our main theorem:

Theorem 1.1: *There is an algorithm which takes as input a closed* L*-term t and which, in time polynomial in the number of β-steps of t and the size of t* and space linear in the size of the largest term in the reduction *outputs a heap containing a term u such that the unfolding of u is the normal form of t.*

Proof. We prove that there is an algorithm which takes as input a closed L-term t and which, in time polynomial in the number of β-steps of t and the size of t *and space linear in the size of the largest term in the reduction* outputs a heap containing a term u such that the unfolding of u is the normal form of t.

The machine M_L from Theorem 4.3 is an implementation of such an algorithm. $\square$

Furthermore, the simulation of L on Turing machines computes the normal form as pair of closure and heap, as defined in Definition 2.8. It is possible to unfold this heap into a program:

Lemma 4.2. *There is a machine M_{unf} that, given a heap H and a closure g with $g \gg_H s$, computes s (explicitly encoded as γs) in time $O(poly(\|s\|, \|H\|, \|g\|))$ and space $O(\|s\| \cdot (\|g\| + \|H\|))$.*

5 SIMULATING TURING MACHINES IN L

The remaining direction of the proof of the strong invariance thesis requires us to prove that Turing machines can be simulated with L consuming only a constant overhead in space and a polynomial overhead in time with respect to our measures $\|\cdot\|_S$ and $\|\cdot\|_T$.

Accattoli and Dal Lago [Accattoli and Dal Lago 2012] show that counting head-reductions is an invariant time measure. In the associated technical report, they give a linear simulation of Turing machines in the deterministic λ-calculus, a fragment of the λ-calculus where all weak evaluation strategies coincide. Although they treat variables as values, reduction in L also coincides, because all considered terms are closed. The construction uses standard Scott encodings $\ulcorner x \urcorner$ for strings x and is explained in all detail in [Dal Lago and Accattoli 2017], spelling out all intermediate terms during simulation explicitly.

It turns out that this construction also only has a constant factor overhead in space w.r.t our measure $\|\cdot\|_S$. This can easily be verified by checking all intermediate terms spelled out in the proofs of [Dal Lago and Accattoli 2017]. One has to take care that a linear amount of steps (i.e. all steps annotated with $O(\cdot)$ or $\Theta(\cdot)$ instead of constants) does not introduce a super-linear space overhead. This is the case, because all such sequences of steps only use substitutions where the substituted variable occurs at most once, effectively decreasing the term size. Note that since names in the simulation are all distinct, the translation to de Bruijn indices has no overhead. Thus the simulation is linear in time *and space*:

Theorem 5.1. *Let $f : \Sigma^* \to \Sigma^*$ be a function that is computable by a Turing machine $\mathcal{M}$ in time $\mathcal{T}$ and in space $\mathcal{S}$. Then there exists an* L*-term $\overline{\mathcal{M}}$ such that for every $x \in \Sigma^*$ we have that*

(1) $\overline{\mathcal{M}}\ \ulcorner x \urcorner >^* \ulcorner f(x) \urcorner$,
(2) $\left\| \overline{\mathcal{M}}\ \ulcorner x \urcorner \right\|_S \in O(|x| + \mathcal{S}(|x|))$, and
(3) $\left\| \overline{\mathcal{M}}\ \ulcorner x \urcorner \right\|_T \in O(|x| + \mathcal{T}(|x|))$.

Proof. Take $\overline{\mathcal{M}}$ as in Theorem 5.5. in [Dal Lago and Accattoli 2017]. □

6 THE WEAK CALL-BY-BALUE λ-CALCULUS IS REASONABLE

We explain how existing simulations of Turing machine in the λ-calculus already have polynomial time and constant factor space overhead in Section 5. With both the simulations, we are now able to show Theorem 1.2, that is, the invariance thesis for the weak call-by-value λ-calculus.

Theorem 1.2 (L is reasonable): *Let Σ be a finite alphabet such that $\{true, false\} \subseteq \Sigma$ and let $f : \Sigma^* \to \{true, false\}$ be a function. Furthermore, let $\mathcal{T}, \mathcal{S} \in \Omega(n)$.*

(1) If f is L*-computable in time $\mathcal{T}$ and space $\mathcal{S}$, then f is computable by a Turing machine in time $O(poly(\mathcal{T}(n)))$ and space $O(\mathcal{S}(n))$.*
(2) If f is computable by a Turing machine in time $\mathcal{T}$ and space $\mathcal{S}$, then f is L*-computable in time $O(poly(\mathcal{T}(n)))$ and space $O(\mathcal{S}(n))$.*

Proof of Theorem 1.2. Let Σ be a finite alphabet such that $\{true, false\} \subseteq \Sigma$ and let $f : \Sigma^* \to \{true, false\}$ be a function. Furthermore, let $b = \max\{\left\|\ulcorner true \urcorner\right\|, \left\|\ulcorner false \urcorner\right\|\}$ and $\mathcal{T}, \mathcal{S} \in \Omega(n)$. Note that b is a constant only depending on the fixed alphabet Σ.

For the first direction, we assume that f is L-computable in time $\mathcal{T}$ and space $\mathcal{S}$. By definition, there is hence a term s_f such that for all $x \in \Sigma^*$ we have that

$$s_f \ulcorner x \urcorner \succ^* \ulcorner f(x) \urcorner \text{ and } \left\| s_f \ulcorner x \urcorner \right\|_{\mathrm{T}} \leq \mathcal{T}(|x|) \text{ and } \left\| s_f \ulcorner x \urcorner \right\|_{\mathrm{S}} \leq \mathcal{S}(|x|)\,.$$

We construct a Turing machine M_f as follows. On input x, M_f executes M_{L} on the (closed) term $s := s_f \ulcorner x \urcorner$, which computes a heap H and a closure g such that $g \gg_H \ulcorner f(x) \urcorner$ in time $O(\mathrm{poly}(\|s\|, \|s\|_{\mathrm{T}}))$ and space $O(\|s\|_{\mathrm{S}})$, by Theorem 4.3 – note that s_f as well as M_{L} are hard-coded in M_f. We observe that

$$\|g\| + \|H\| \in O(\mathrm{poly}(\|s\|, \|s\|_{\mathrm{T}})) \quad \text{and} \quad \|g\| + \|H\| \in O(\|s\|_{\mathrm{S}})\,, \tag{1}$$

where the former holds as writing down g and H cannot take more time than the overall running time bound $O(\mathrm{poly}(\|s\|, \|s\|_{\mathrm{T}}))$ and the latter is due to the space bound $O(\|s\|_{\mathrm{S}})$ of M_f. After that, M_f executes M_{unf} on H and g which yields $\ulcorner f(x) \urcorner$ and finally, depending on whether $\ulcorner f(x) \urcorner = \ulcorner \mathit{true} \urcorner$ or $\ulcorner f(x) \urcorner = \ulcorner \mathit{false} \urcorner$, M_f outputs *true* or *false* accordingly. By Lemma 4.2, the final steps take time $O(\mathrm{poly}(b, \|H\|, \|g\|))$ and space $O(b \cdot (\|g\| + \|H\|))$. Now the final time consumption is given by

$$\begin{aligned} O(\mathrm{poly}(\|s\|, \|s\|_{\mathrm{T}}) + \mathrm{poly}(b, \|H\|, \|g\|)) &\leq O(\mathrm{poly}(\|s\|, \|s\|_{\mathrm{T}}) + \mathrm{poly}(b, \|s\|, \|s\|_{\mathrm{T}})) && (2)\\ &\leq O(\mathrm{poly}(|x|, \mathcal{T}(|x|))) && (3)\\ &\leq O(\mathrm{poly}(\mathcal{T}(|x|)))\,, && (4) \end{aligned}$$

where (2) is due to Equation (1), (3) holds as $\left\|s_f\right\|$ and b are constants and (4) follows from the fact that $\mathcal{T} \in \Omega(n)$. The overall space consumption is bounded by

$$\begin{aligned} O(\|s\|_{\mathrm{S}} + b \cdot (\|H\| + \|g\|)) &\leq O((b+1) \cdot \|s\|_{\mathrm{S}}) && (5)\\ &\leq O(\mathcal{S}(|x|))\,, && (6) \end{aligned}$$

where (5) is due to Equation (1) and (6) holds as b is a constant.

For the converse direction, we assume that f can be computed by a Turing machine $\mathcal{M}$ in time $\mathcal{T}$ and space $\mathcal{S}$. We invoke Theorem 5.1 to obtain a term $\overline{\mathcal{M}}$ which shows that f is L-computable in space $O(|x| + \mathcal{S}(|x|))$, and time $O(|x| + \mathcal{T}(|x|))$. We conclude the proof by observing that $O(|x| + \mathcal{S}(|x|)) = O(\mathcal{S}(|x|))$ and $O(\mathcal{T}(|x|)) = O(|x| + \mathcal{T}(|x|))$ as both, $\mathcal{S}$ and $\mathcal{T}$ are contained in $\Omega(n)$. □

The restrictions $\mathcal{S}, \mathcal{T} \in \Omega(n)$ are due to several factors. First, they are restrictions imposed by the simulation of Turing machines in L (Theorem 5.1). Secondly, they also occur naturally in the simulation of L on Turing machines (Theorems 4.3 and 1.1): We exclude sublinear space (i.e. require $\mathcal{S} \in \Omega(n)$), because we treat the input as part of the computation and thus our simulation needs at least as much space as the size of the input. We exclude sublinear time (i.e. require $\mathcal{T} \in \Omega(n)$) because our simulation needs time polynomial not only in the number of steps, but also in the size of the initial term containing the input.

7 RELATED WORK

We have already mentioned the recent long line of work by Accattoli, Dal Lago, Sacerdoti Coen, Guerrieri and Martini (for an overview see [Accattoli 2018]) analysing reasonable time measures and implementations of several λ-calculi.

Type systems for call-by-name and call-by-value λ-calculi can be used to logically characterise complexity classes (*P* [Asperti and Roversi 2002], *LOGSPACE* [Schöpp 2006], *PSPACE* [Gaboardi et al. 2008]).

There is recent work in investigating strategies to evaluate open terms, for instance open call-by-value, which is reasonable for time [Accattoli and Coen 2015; Accattoli and Guerrieri 2017], but the question for space is open. On the more applied side, there is work on time and space profiling based on lazy graph reduction [Sansom and Peyton Jones 1995] in Haskell. More recent work uses a graph-based cost-semantics used for space-profiling [Spoonhower et al. 2008], based on earlier measures in [Blelloch and Greiner 1995]. Moreover, computation in sub-linear space with an external memory has been studied [Dal Lago and Schöpp 2010], which we do not cover in this paper.

8 FUTURE WORK

As mentioned before, our result does not extend to sublinear time or space classes. By connecting our result to the insights from related work, e.g. the logical characterisations of classes or the treatment of sublinear space using external memory, it might be possible to extend our result to sublinear classes as well. We are interested in finding measures that still allow for compositional reasoning and are not defined as the resource consumption of some machine.

Another line of extension would be to cover more versions of the λ-calculus. For instance, the full λ-calculus can be translated into weak call-by-value e.g. using a CPS translation. Potentially, translations like this combined with our result can contribute to answers how reasonable complexity measures for other λ-calculi look.

Our main motivation for this work was to define complexity classes in terms of the λ-calculus. While this goal has succeeded, much follow-up work backing our claim that our measures are actually feasible to use for mechanisation has to be done. We would like to mechanise basic complexity theory in Coq, starting with results like the time- and space hierarchy theorems, Savitch's theorem, Cook's theorem, or the inclusions $P \subseteq NP \subseteq PSPACE \subseteq EXP$.

While our results imply that $P \subseteq PSPACE$, the proof relies on Turing machines. We would like to investigate whether this result can be proved without reference to Turing machines, for instance by implementing our space-aware interleaving simulation in a universal λ-term.

And finally, it remains open whether there exists a higher-order interpretation of space complexity that does not exhibit (exponential) size explosion. It is well-known that RAM-machines also exhibit size explosion, however only polynomially instead of exponentially [Slot and van Emde Boas 1984]. Recall that in Subsection 1.1 we discuss a transformation (using s_E and $\hat{s_E}$ as examples) which optimises terms exhibiting size explosion in case the normal form of this terms is of polynomial size again, i.e. in cases where the size explosion was unnecessary. Consequently, our space measure is a severe overestimation on such terms. We could thus define the space measure of a term t with a normal form of polynomial size as the minimum of our space measure and a polynomial in the initial size and the time measure of t. Depending on the perspective, this could even be seen as a more faithful measure.

However, as argued before, the contribution of this paper is the insight that (some form of) the λ-calculus can serve as formal basis to define time and space complexity classes, based on machine-independent natural measures that one can work well with. The observation that that size explosion can be neglected on some terms would complicate the definition of space complexity, we thus refrain from incorporating it formally. The observation may however still turn out to be helpful in the future to find a natural space measure for the λ-calculus without exponential size explosion.

A BIG-STEP CHARACTERISATION OF REDUCTION

While the characterisation of our time and space measure in terms of a normalising reduction $s_0 \succ \ldots \succ s_k$ are intuitive, a big-step characterisation allows for easy, inductive analyses of the abstract machines evaluating L in Section 3.

Definition A.1 (Time Measure).

$$\frac{}{\lambda s \Downarrow_0^T \lambda s} \qquad \frac{s \Downarrow_{k_1}^T \lambda s' \qquad t \Downarrow_{k_2}^T \lambda t' \qquad s'^0_{\lambda t'} \Downarrow_{k_3}^T u}{st \Downarrow_{k_1+k_2+1+k_3}^T u}$$

Definition A.2 (Space Measures).

$$\frac{}{\lambda s \Downarrow_{|\lambda s|}^S \lambda s}$$

$$\frac{s \Downarrow_{m_1}^S \lambda s' \qquad t \Downarrow_{m_2}^S \lambda t' \qquad s'^0_{\lambda t'} \Downarrow_{m_3}^S u \qquad m = \max(1 + m_1 + |t|, 1 + |\lambda s'| + m_2, m_3)}{st \Downarrow_m^S u}$$

In the second rule, each of the three recursive assumptions could contain the largest subterm, so we take the maximum of each m_i, while accounting for the size of the remaining part of the term, e.g. during the reduction of s in st, the t and the application itself contribute to $1 + \|t\|$ additional size by definition of the term siz.

The following lemmas allow us to use the two characterisations interchangeably:

Lemma A.1. *$s \Downarrow_k^T t$ iff $s \succ^k t$ and t is an abstraction.*

Note that especially if $\|s\|_{\mathrm{T}} = k$, then $s \Downarrow_k^{\mathrm{T}} t$ for some t.
We write $s \succ_m^* t$ if s reduces to t where the largest intermediate term has size m.

Lemma A.2. *$s \Downarrow_m^S t$ iff $s \succ_m^* t$ and t is an abstraction.*

Note that especially, if $\|s\|_{\mathrm{S}} = m$, then $s \Downarrow_m^{\mathrm{S}} t$ for some t.

B TECHNICAL DEFINITIONS AND LEMMAS

Lemma B.1. *γ is injective.*

Adding a value t' to an environment a results in substitution in the unfolded term:

Lemma B.2. *If $H[a'] = (g, a)$ with $g \gg_H t'$ and $s\langle a, 1\rangle \downarrow_H s'$, then $s\langle a', 0\rangle \downarrow_H s'^0_{t'}$.*

Unfolding only changes de Bruijn indices starting at k:

Lemma B.3. *If s is bounded by k, then $s\langle a, k\rangle \downarrow_H s$.*

Proof. Induction on $s < k$. □

In particular, closed terms are invariant under unfolding.
The unfolding relation only holds if all de Bruijn indices up to k are bound in a.

Lemma B.4. *If $s\langle a, k\rangle \downarrow_H s'$, then $s' < k$*

Proof. Induction on $s\langle a, k\rangle \downarrow_H s'$. □

In particular for $k = 0$, unfolding results in closed terms.
A heap is extended by another heap if the latter contains a superset of the entries:

Definition B.1. $H \subseteq H' := \forall a, H[a] \neq \bot \to H[a] = H'[a]$

Lemma B.5. *Heap extension $H \subseteq H'$ is transitive and reflexive.*

Heap extension does not change the result of certain operations:

Lemma B.6. *Assume $H \subseteq H'$*

(1) If $H[a,n] \neq \bot$, then $H[a,n] = H'[a,n]$.
(2) If $s\langle a,k\rangle \downarrow_H s'$, then $s\langle a,k\rangle \downarrow_{H'} s'$.
(3) If $g \gg_H s$, then $g \gg_{H'} s$.

Proof. The first claim follow by induction on n. The second claim follows by induction on $s\langle a,k\rangle \downarrow_H s'$. The only interesting case is the one where $s = n \geq k$, which requires the first claim. The third claim follows from the second by definition of $\gg_H$. □

Lemma B.7. *Assume $\sigma_s \succ^k (T,V,H) = \sigma$ for some term s.*

(1) $|T| + |V| \leq k+1$
(2) $|H| \leq k$
(3) $\|P\| \leq \|s\|$ and $a \leq |H|$ for all $P/a \in T + V$
(4) $\|P\| \leq \|s\|$ and $a \leq |H|$ and $b \leq |H|$ for all $((P,a),b) \in H$

C PROOFS

Proof of Lemma 2.1:

Proof. Induction on s. □

Proof of Lemma 2.2:

Proof. The generalisation $\varphi_{k,Q}(\gamma s + P) = \varphi_{k,Q+\gamma s}P$ follows by induction on s. □

Proof of Lemma 2.3:

Proof. The generalisation $(\gamma s + P)^k_{\gamma t} = \gamma(s^k_t) + P^k_{\gamma t}$ holds by induction on s. □

Proof of Lemma 3.1:

Proof. We show the generalisation, *if $s \Downarrow^T_k t$, then for all Q,T,V we have $((\gamma s + Q) :: T, V) \succ^{3k+1} (Q ::_{tc} T, P :: V)$ for some P with $P \gg t$*, from which the claim follows for $Q = R = V = []$.

Proof by induction on $s \Downarrow^T_k t$ as defined in Definition A.1.

In the case $\lambda s \Downarrow^T_0 \lambda s$, we have

$$\begin{aligned}((\gamma(\lambda s) + Q) :: T, V) &= ((\mathsf{lam} :: \gamma s :: \mathsf{ret} + Q) :: T, V)\\ &\succ (Q ::_{\mathrm{tc}} T, \gamma s :: V) && \text{Lemma 2.2}\end{aligned}$$

and $\gamma s \gg s$ holds by definition.

In the case $st \Downarrow^T_{k_1+k_2+1+k_3} u$ with all names as in Definition A.1, we have

$$\begin{aligned}((\gamma(st) + Q) :: T, V) &= ((\gamma s + \gamma t + \mathsf{app} :: Q) :: T, V)\\ &\succ^{3k_1+1} ((\gamma t + \mathsf{app} :: Q) ::_{\mathrm{tc}} T, \gamma s' :: V) && \text{IH for } s \Downarrow^T_{k_1} \lambda s'\\ &\succ^{3k_2+1} ((\mathsf{app} :: Q) ::_{\mathrm{tc}} T, \gamma t' :: \gamma s' :: V) && \text{IH for } t \Downarrow^T_{k_2} \lambda t'\\ &\succ (((\gamma s')^0_{\gamma(\lambda t')} :: Q ::_{\mathrm{tc}} T, \gamma t' :: \gamma s' :: V)\\ &= (\gamma(s'^0_{\lambda t'}) :: Q ::_{\mathrm{tc}} T, V) && \text{Lemma 2.3}\\ &\succ^{3k_3+1} (Q ::_{\mathrm{tc}} T, \gamma u :: V) && \text{IH for } s'^0_{\lambda t'} \Downarrow^T_{k_3} \lambda u\end{aligned}$$

Note that $::_{\text{tc}}$ is $::$ in the first two reductions.

The claim follows as $3(k_1 + k_2 + 1 + k_3) + 1 = (3k_1 + 1) + (3k_2 + 1) + 1 + (3k_3 + 1)$ and $\gamma u \gg \lambda u$ by definition. □

Proof of Lemma 3.2:

Proof. We show a generalisation, *if* $s \Downarrow^S_m t$, *then for all* Q, T, V *we have* $((\gamma s + Q) :: T, V) \succ^*_{m'} (Q ::_{tc} T, P :: V)$ *for some* P,m' *with* $P \gg t$ *and* $m + \|P ::_{tc} T\| + \|V\| \leq m' \leq 2m + \|P ::_{tc} T\| + \|V\|$, from which the claim follows with $Q = T = V = []$.

Proof by induction on $s \Downarrow^S_m t$ as defined in Definition A.2. By definition of $\Downarrow^S$, this proof is very similar to the one for Lemma 3.1. The only difference is the needed equalities between the various space-measures m_i. Those are proven by tedious, but straightforward computations when using the facts that $\|P\| + \|T\| \leq \|P ::_{\text{tc}} T\| \leq \|P\| + \|T\| + 1$ and Lemma 2.3 and Lemma 2.1 and the fact that $s \Downarrow^S_m t$ implies $\|s\| \leq m \geq \|t\|$. □

Proof of Lemma 3.3:

Proof. We show a generalisation, *If* $s \Downarrow^T_k t$ *and* $s_0\langle a, 0\rangle \downarrow_H s$, *then there are* g *and* H' *with* $g \gg_{H'} t$ *such that* $((\gamma s_0 + P, a) :: T, V, H) \succ^{4k+1} ((P, a) :: T, g :: V, H')$ *for any* P, T, V, *and* $H \subseteq H'$. Here $H \subseteq H'$ is meant as in Definition B.1. The original claim follows with $P = T = V = H = []$, Lemma B.3 and the reduction rule for empty tasks.

Proof by induction on $s \Downarrow^T_m t$ as in Definition A.1. In the case of $\lambda s \Downarrow^T_0 \lambda s$, a case distinction on $s_0\langle a, 0\rangle \downarrow_H \lambda s$ yields two cases: s_0 is either a variable with a value bound in a, or s_0 is an abstraction.

In the case $s_0 = n$, we obtain Q, b, s_1 such that $H[a, n] = (Q, b)$ with $Q \gg s_1$ and $s_1\langle b, 0\rangle \downarrow_H \lambda s$. The claim holds as $(Q, b) \gg_H \lambda s$ and $((\gamma n + P, a) :: T, V, H) = ((\text{var } n :: P, a) :: T, V, H) \succ ((P, a) :: T, (Q, b) :: V, H)$.

In the case $s_0 = \lambda s_1$, we have that $s_1\langle a, 1\rangle \downarrow_H s$. The claim holds as $(\gamma s_1, a) \gg_H \lambda s$ and $((\gamma(\lambda s_1) + P, a) :: T, V, H) = ((\text{lam} :: \gamma s_1 + \text{ret} :: P, a) :: T, V, H) \succ ((P, a) :: T, (\gamma s_1, a) :: V, H)$.

In the other case of the induction, $st \Downarrow^T_{k_1+k_2+1+k_3} u$, we have $s \Downarrow^T_{k_1} \lambda s'$ and $t \Downarrow^T_{k_2} \lambda t'$ and $s'^0_{\lambda t'} \Downarrow^T_{k_3} u$ and an inductive hypothesis for each of those. We also have $s_0\langle a, 0\rangle \downarrow_H st$. Now $s_0 = s_1 t_1$ must be an application. Note that even in the second rule of Definition 2.7, the definition of $\gg$ on programs implies that the unfolded term would be an abstraction.

So we have $s_1\langle a, 0\rangle \downarrow_H s$ and $t_1\langle a, 0\rangle \downarrow_H t$ for some s_1, t_1. We now construct the reduction of the machine using the inductive hypothesis. We will explain where the new objects in the following reduction come from in the next paragraph.

$$
\begin{aligned}
((\gamma(s_1 t_1) + P, a) :: T, V, H) &= (\gamma s_1 + \gamma t_1 + \text{app} :: P, a) :: T, V, H) && (7)\\
&\succ^{4k_1+1} ((\gamma t_1 + \text{app} :: P, a) :: T, (\gamma s_2, a_2) :: V, H_1) && \text{IH} \quad (8)\\
&\succ^{4k_2+1} ((\text{app} :: P, a) :: T, g_t :: (\gamma s_2, a_2) :: V, H_2) && \text{IH} \quad (9)\\
&\succ ((\gamma s_2, a'_2) :: (P, a) :: T, V, H'_2) && (10)\\
&\succ^{4k_3+1} (([], a'_2) :: (P, a) :: T, g_u :: V, H_3) && \text{IH} \quad (11)\\
&\succ ((P, a) :: T, g_u :: V, H_3) && (12)
\end{aligned}
$$

In this reduction, the inductive hypothesis for s_1 in (8) yields s_2, a_2 and H_1 such that $H \subseteq H_1$ and $s_2\langle a_2, 1\rangle \downarrow_{H_1} s'$. The inductive hypothesis on t_1 in (9) yields g_t and H_2 such that $H_1 \subseteq H_2$ and $g_t \gg_{H_2} \lambda t'$. In the step for beta reduction, (10), we have $(H'_2, a'_2) = \text{put} H_2(g_t, a_2)$ and $H_2 \subseteq H'_2$. With Lemma B.2, this implies $s_2\langle a'_2, 0\rangle \downarrow_{H'_2} s'^0_{\lambda t'}$. This now allows the use of the third inductive hypothesis in (11), obtaining g_u and H_3 with $g_u \gg u$ and $H'_2 \subseteq H_3$. Note that we use Lemma B.5 to transfer several properties along the changing heaps. Now the claim holds for g_u and H_3. □

Proof of Lemma 3.4:

Proof. Follows from Lemma B.7 □

Proof of Lemma 4.2:

Proof. We first consider a partial function $f_H Pak$ that computes the unfolding, but on programs instead of terms:

$$\begin{aligned}
f_H([]) ak &:= [] \\
f_H(\mathsf{app} :: P) ak &:= \mathsf{app} :: f_H Pak \\
f_H(\mathsf{ret} :: P) a(1+k) &:= \mathsf{ret} :: f_H Pak \\
f_H(\mathsf{lam} :: P) ak &:= \mathsf{lam} :: f_H Pa(1+k) \\
f_H(\mathsf{var}\, n :: P) ak &:= \mathsf{lam} :: f_H Qb1 + \mathsf{ret} :: f_H Pak && \text{if } n \geq k \text{ and } H[a, n-k] = (Q, b) \\
f_H(\mathsf{var}\, n :: P) ak &:= \mathsf{var}\, n :: f_H Pak && \text{if } n < k
\end{aligned}$$

For this set of equations, we can show *if* $k\langle s, a\rangle \downarrow_H s'$, *then* $f_H(\gamma s + Q)ak = \gamma s' + f_H Qak$ by induction on s'. With $Q = []$, this means that $(P, a) \gg_H \lambda s'$ implies $f_H Pa1 = \gamma s'$. So f indeed computes the unfolding on programs.

Implementing f in Turing machines, we first note that during execution, all considered k, a and P are bound is bound by $\|g\| + \|H\|$, as all addresses come from g or H and k can not be larger than the largest program in g or H. In the equation using $H[\cdot, \cdot]$, an additional explicit stack is needed to remember P for after the recursive call on Q. This stack is bound in length by $O(\|s\|)$, as every recursive call computes at least one symbol of the result. This means that the algorithm runs in space $O(\|s\| \cdot (\|g\| + \|H\|))$.

Furthermore, each equation produces a symbol of the result, so the total number of calls on f is bound by $\|s\|$. Every equation, except the one where $H[\cdot, \cdot]$ occurs, performs a constant number of operations. This other equation needs to traverse H at most n times before recurring, where n is the largest de Bruijn index occurring. In total, this means that the algorithm runs in time $O(\mathrm{poly}(\|s\|, \|g\|, \|H\|))$. □

Proof of Lemma A.1:

Proof. For the direction assuming $s \Downarrow^{\mathrm{T}}_k t$, the claim follows by induction on $\Downarrow^{\mathrm{T}}$ using two compatibility lemmas of $\succ^k$ with term-level application, the first beeing that $s \succ^k s'$ implies $st \succ^k s't$, and the second that $t \succ^k t'$ implies $(\lambda s)t \succ^k (\lambda s)t'$.

For the other direction, we first show

Claim C.1. *If* $s \succ s'$ *and* $s' \Downarrow^T_k t$, *then* $s \Downarrow^T_{1+k} t$.

This claim follows by induction in $s \succ s'$.

Now, assuming $s \succ^k \lambda t$, we can show $s \Downarrow^{\mathrm{T}}_k t$ by induction on k using Claim C.1 in the case where $k > 0$. □

Proof of Lemma A.2:

Proof. For the direction assuming $s \Downarrow^{\mathrm{S}}_{m'} t$, the claim follows by induction on $\Downarrow^{\mathrm{S}}$. In the inductive case, two compatibility lemmas of $\succ^*_m$ with term-level application are helpful: the first beeing that $s \succ^*_m s'$ implies $st \succ^*_{1+m+\|t\|} s't$, and the second that $t \succ^*_m t'$ implies $(\lambda s)t \succ^*_{1+m+\|\lambda s\|} (\lambda s)t'$. Furthermore, the fact that $t \Downarrow^{\mathrm{S}}_{m_2} \lambda t'$ implies $\|\lambda t'\| \leq m_2$ is needed.

For the other direction, we first show

Claim C.2. *If $s \succ s'$ and $s' \Downarrow^S_m t$, then $s \Downarrow^S_{\max(\|s\|,m)} t$.*

This claim follows by induction in $s \succ s'$. The equalities between expressions involving max are tedious to check, but follow only using the inductive hypothesis, the definition of $\Downarrow^S$, the definition of the size of terms and that $s \Downarrow^S_m t$ implies $\|s\| \leq m \geq \|t\|$.

Now, assuming $s \succ^k_m \lambda t$, we can show $s \Downarrow^S_m \lambda t$ by induction on k. In the base case, $\lambda t \succ^0_m \lambda t$ implies $m = \|\lambda t\|$ implies $\lambda t \Downarrow^S_m \lambda t$.

In the inductive case, $s \succ^{1+k}_m \lambda t$ implies a decomposition $s \succ s' \succ^k_{m'} \lambda t$ for some s', m' with $m = \max(\|s\|, m')$. Then the inductive hypothesis for k is $s' \Downarrow^S_{m'} \lambda t$, which together with Claim C.2 implies $s \Downarrow^S_m \lambda t$. □

Proof of Lemma B.1:

Proof. We define an inverse $\delta : \mathbb{N} \to \mathsf{Pro} \to \mathsf{Ter}^* \to {\mathsf{Ter}^*}_\perp$ of γ:

$$\begin{aligned} \delta k(\mathsf{var}\, n :: P)A &:= \delta k P(n :: P) & \delta k(\mathsf{app} :: P)(t :: s :: A) &:= \delta k P(st :: A) \\ \delta k(\mathsf{lam} :: P)A &:= \delta(1+k)PA & \delta(1+k)(\mathsf{ret} :: P)(s :: A) &:= \delta k P(\lambda s. :: A) \\ \delta k[]A &:= A \end{aligned}$$

Now $\delta k(\gamma s + P)A = \delta k P(s :: A)$ holds by induction on s. □

Proof of Lemma B.2:

Proof. Let $H[a'] = (g, a)$ and $g \gg_H t'$. We show a generalisation: *If $s\langle a, 1+k\rangle \downarrow_H s'$, then $s\langle a', k\rangle \downarrow_H {s'}^k_{t'}$*, by induction on $s\langle a, 1+k\rangle \downarrow_H s'$.

In the case $s = n < 1 + k$ and $s' = n$, there are two subcases: Assuming $n < k$, $n\langle a', k\rangle \downarrow_H n = {s'}^k_{t'}$ holds by definition. Otherwise, we have $n = k$. Since $H[a'] = (g, a)$, we have $H[a', n-k] = H[a', 0] = g$. With P, b such that $g = (P, b)$, we have $n\langle a', k\rangle \downarrow_H n^k_{t'} = t'$ by the second rule since $(P, b) \gg_H t'$ implies $(P \gg t)$ and $(t, b) \gg_H t'$ for some t.

In the case $s = n \geq 1 + k$, we have $H[a, n-(1+k)] = (P, b)$ and $P \gg u$ with $u\langle b, 0\rangle \downarrow_H s'$ for some P, b. As $H[a'] = (g, a)$, we have $H[a', n-k] = H[a, n-(1+k)] = (P, b)$. Therefore $n\langle a, k\rangle \downarrow_H {s'}^k_{t'} = s'$ by the second rule, where the equality holds as s' is closed by Lemma B.4.

In the other cases, i.e. application and abstraction, the claim follows by the inductive hypothesis and the definition of $\cdot\langle\cdot, k\rangle \downarrow_H \cdot$. □

Proof of Lemma B.5:

Proof. Transitivity and reflexivity follow from the same properties for equality. □

Proof of Lemma B.7:

Proof. All claims follow by induction on k. The third claim uses that φP always returns a sublist of P. □

ACKNOWLEDGMENTS

We would like to thank Beniamino Accattoli for encouraging us to write down our results and his support in explaining our results in the most transparent way.

Dominik Kirst, Dominique Larchey-Wendling, Simon Spies, Maximilian Wuttke, and the anonymous reviewers gave useful hints on the presentation, which we are very grateful for.

We would like to thank Gert Smolka and Holger Dell for many fruitful and encouraging discussions over the last 3 years this project has spanned.

REFERENCES

Beniamino Accattoli. 2016. The Complexity of Abstract Machines. In *Proceedings Third International Workshop on Rewriting Techniques for Program Transformations and Evaluation, WPTE@FSCD 2016, Porto, Portugal, 23rd June 2016*. 1–15. https://doi.org/10.4204/EPTCS.235.1

Beniamino Accattoli. 2018. (In)Efficiency and Reasonable Cost Models. *Electr. Notes Theor. Comput. Sci.* 338 (2018), 23–43. https://doi.org/10.1016/j.entcs.2018.10.003

Beniamino Accattoli and Claudio Sacerdoti Coen. 2015. On the Relative Usefulness of Fireballs. In *30th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2015, Kyoto, Japan, July 6-10, 2015*. 141–155. https://doi.org/10.1109/LICS.2015.23

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

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

Beniamino Accattoli and Giulio Guerrieri. 2017. Implementing Open Call-by-Value. In *Fundamentals of Software Engineering - 7th International Conference, FSEN 2017, Tehran, Iran, April 26-28, 2017, Revised Selected Papers*. 1–19. https://doi.org/10.1007/978-3-319-68972-2_1

Andrea Asperti and Wilmer Ricciotti. 2015. A formalization of multi-tape Turing machines. *Theoretical Computer Science* 603 (Oct. 2015), 23–42. https://doi.org/10.1016/j.tcs.2015.07.013

Andrea Asperti and Luca Roversi. 2002. Intuitionistic Light Affine Logic. *ACM Trans. Comput. Log.* 3, 1 (2002), 137–175. https://doi.org/10.1145/504077.504081

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

Nicolaas Govert De Bruijn. 1972. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. In *Indagationes Mathematicae (Proceedings)*, Vol. 75. Elsevier, 381–392. Issue 5.

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

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

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

Nachum Dershowitz and Evgenia Falkovich-Derzhavetz. 2015. The Invariance Thesis. *Logical Methods in Computer Science (to appear)* (2015). http://www.cs.tau.ac.il/~nachumd/papers/InvarianceThesis.pdf

Yannick Forster, Fabian Kunze, and Marc Roth. 2017. The strong invariance thesis for a λ-calculus. *Workshop on Syntax and Semantics of Low-Level Languages (LOLA)* (2017).

Yannick Forster, Fabian Kunze, and Maximilian Wuttke. 2019. Verified Programming of Turing Machines in Coq. (2019). https://github.com/uds-psl/tm-verification-framework/

Yannick Forster and Gert Smolka. 2017. Weak Call-by-Value Lambda Calculus as a Model of Computation in Coq. In *Interactive Theorem Proving - 8th International Conference, ITP 2017, Brasília, Brazil, September 26-29, 2017, Proceedings*. 189–206. https://doi.org/10.1007/978-3-319-66107-0_13

Marco Gaboardi, Jean-Yves Marion, and Simona Ronchi Della Rocca. 2008. A logical account of PSPACE. In *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, San Francisco, California, USA, January 7-12, 2008*. 121–131. https://doi.org/10.1145/1328438.1328456

Fabian Kunze, Gert Smolka, and Yannick Forster. 2018. Formal Small-Step Verification of a Call-by-Value Lambda Calculus Machine. In *Programming Languages and Systems - 16th Asian Symposium, APLAS 2018, Wellington, New Zealand, December 2-6, 2018, Proceedings*. 264–283. https://doi.org/10.1007/978-3-030-02768-1_15

Julia L. Lawall and Harry G. Mairson. 1996. Optimality and Inefficiency: What Isn't a Cost Model of the Lambda Calculus?. In *Proceedings of the 1996 ACM SIGPLAN International Conference on Functional Programming, ICFP 1996, Philadelphia, Pennsylvania, USA, May 24-26, 1996*. 92–101. https://doi.org/10.1145/232627.232639

Michael Norrish. 2011. Mechanised Computability Theory. In *Interactive Theorem Proving - Second International Conference, ITP 2011, Berg en Dal, The Netherlands, August 22-25, 2011. Proceedings*. 297–311. https://doi.org/10.1007/978-3-642-22863-6_22

Gordon D. Plotkin. 1975. Call-by-Name, Call-by-Value and the lambda-Calculus. *Theor. Comput. Sci.* 1, 2 (1975), 125–159. https://doi.org/10.1016/0304-3975(75)90017-1

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

Ulrich Schöpp. 2006. Space-Efficient Computation by Interaction. In *Computer Science Logic, 20th International Workshop, CSL 2006, 15th Annual Conference of the EACSL, Szeged, Hungary, September 25-29, 2006, Proceedings*. 606–621. https://doi.org/10.1007/11874683_40

Cees F. Slot and Peter van Emde Boas. 1984. On Tape Versus Core; An Application of Space Efficient Perfect Hash Functions to the Invariance of Space. In *Proceedings of the 16th Annual ACM Symposium on Theory of Computing, April 30 - May 2, 1984, Washington, DC, USA*. 391–400. https://doi.org/10.1145/800057.808705

Daniel Spoonhower, Guy E. Blelloch, Robert Harper, and Phillip B. Gibbons. 2008. Space profiling for parallel functional programs. *J. Funct. Program.* 20, 5-6 (2008), 417–461. https://doi.org/10.1017/S0956796810000146

The Coq Proof Assistant. 2019. http://coq.inria.fr.