A Verified Cost Model for Call-By-Push-Value

Zhuo Zoey Chen
University of Melbourne, Australia

Johannes Åman Pohjola
University of Gothenburg, Sweden

Christine Rizkallah
University of Melbourne, Australia

Abstract

The call-by-push-value λ-calculus allows for syntactically specifying the order of evaluation as part of the term language. Hence, it serves as a unifying language for embedding various evaluation strategies including call-by-value and call-by-name. Given the impact of call-by-push-value, it is remarkable that its adequacy as a model for computational complexity theory has not yet been studied. In this paper, we show that the call-by-push-value λ-calculus is *reasonable* for both time and space complexity. A reasonable cost model can encode other reasonable cost models with polynomial overhead in time and constant factor overhead in space. We achieve this by encoding call-by-push-value λ-calculus into Turing machines, following a simulation strategy by Forster et al.; for the converse direction, we prove that Levy's encoding of the call-by-value λ-calculus has reasonable complexity bounds. The main results have been formalised in the HOL4 theorem prover.

2012 ACM Subject Classification Theory of computation → Lambda calculus

Keywords and phrases lambda calculus, formalizations of computational models, computability theory, HOL, call-by-push-value reduction, time and space complexity, abstract machines

Digital Object Identifier 10.4230/LIPIcs.ITP.2025.7

Supplementary Material *Software (Mechanised Proof)*: `https://github.com/ZhuoZoeyChen/cbpv-reasonable-HOL/` [5], archived at `swh:1:dir:df18377e9fa5e35255f2687ad66ddbc2f010b934`

Acknowledgements We thank Yannick and Fabian for taking the time to meet with us and discuss their work on the weak call-by-value λ-calculus. We are also grateful to the anonymous reviewers for their valuable feedback.

1 Introduction

The λ-calculus [7] is a fundamental model of computation that represents functions as abstractions over variables. It provides a foundation for computability, mathematical logic, and functional programming. Functional programming supports a concise, declarative style of programming that is ideal for reasoning about functional correctness properties.

Besides functional correctness, another important property of a program is its computational complexity. It addresses the vital questions: How fast does my program produce an output? How much space will my program use in order to produce that output?

In complexity analysis, we describe the asymptotic behaviour of *cost functions* that model the cost of the program mathematically. But where do cost functions come from? In practice they are often constructed in an ad hoc manner with no formal connection to any program semantics. *Cost models* bridge that gap.

Creating cost models for functional programming languages is thus an essential topic. It is also a significant challenge due to the abstract nature of functional programming. As λ-calculus is the basis of functional programming languages, a vital question is whether we can create cost models for the λ-calculus. There has been a large body of research [18, 8, 2, 1, 11, 12, 3] dedicated to solving this problem. An important parameter in a cost model for the λ-calculus is the choice of evaluation strategy. One strategy is *call-by-value*, where function calls are applied once the arguments are fully evaluated, as in Standard ML [23]. Another strategy is *call-by-name*, where the evaluation of arguments is deferred and the function call happens first.

The call-by-push-value λ-calculus (CBPV) [19] is a variant of λ-calculus where the evaluation strategy can be set on a call-by-call basis. In particular, CBPV has syntactic constructs that enable delaying or forcing the evaluation of specific terms. Various evaluation strategies, including call-by-value and call-by-name, can be encoded within this subsuming paradigm.

This high expressive power of CBPV has resulted in a long line of work on the language since its inception [10, 9, 24, 22, 13, 15, 6, 16]. Prior work has studied and extended the calculus [16, 27, 22], related it to other calculi [10, 6, 14], and formalised it [24, 13]. The fine-grained control CBPV provides has proved useful to verify compiler optimisations [24].

There are foundational results that demonstrate that CBPV aids in recurrence extraction, which can in turn be used for analysing the complexity of functional programs, with various evaluation strategies [15].

This demonstrates that CBPV can serve as a basis for further research on analysing the complexity of functional programs with various evaluation strategies. As such, it is vital to establish time and space cost models for CBPV. Naturally, such cost models must satisfy some property that demonstrates that they are fit for purpose.

Reasonable Machines

Turing machines are the standard computational model for complexity theory. They have obvious cost models for time (the number of steps) and space (the number of tape cells used). Cost models are less obvious for λ-terms, but the readability and convenience of using λ-terms are higher. Can cost analysis performed in one carry over to the other?

A *reasonable* cost model [26, 28] answers this question in the positive. Reasonableness is a standard requirement for assessing the suitability of computational models for reasoning about complexity, by relating them to Turing machines (which are considered reasonable by definition). The *invariance thesis* [28] states that:

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

Hence the definition of standard complexity classes like P, PSPACE, and EXP are independent of which (reasonable) substrate they are defined on.

Contribution

This paper contributes, to the best of our knowledge, the first reasonable time and space cost models for CBPV. We further provide machine-checked proofs in HOL4 [25] for the core parts thereof. In doing so we build on prior work on formally verified time and space cost models for the weak call-by-value λ-calculus (WCBV) that was formalised in Coq [17, 11].

Related Work

In 1996, Lawall and Mairson [18] proved that the full λ-calculus is reasonable for both time and space using the measures *total ink used* and *maximum ink used.* But the time measure of *total ink used* is too general and hard to apply. In 2008, Dal Lago and Martini [8] provided a different time measure for WCBV, which counts the number of β-steps while taking account of the size of β-redexes. This was further strengthened by Accattoli and Dal Lago [2] in 2016, showing that counting (leftmost-outermost) β-steps makes the full λ-calculus reasonable for time. Continuing on this line, Forster, Kunze, and Roth [11] proved in 2020 that WCBV is reasonable with respect to natural measures, accompanied with a partial formalisation. They define a natural measure to be the number of β-reductions for time, and the size of the biggest intermediate term for space. They proved that WCBV is reasonable by interleaving two evaluation strategies: a substitution-based strategy and heap-based strategy, which we adapt and implement for CBPV in our paper. Forster, Kunze, Smolka, and Wuttke [12] provided a complete formalisation in 2021, showing that WCBV is reasonable for time. The complete formal verification of the space invariance thesis for the same calculus still remains open. One limitation of this line of work, as well as ours, is that we do not consider sublinear time or space classes. In contrast, Accattoli, Dal Lago, and Vanoni [3] presented a reasonable space cost model for the λ-calculus that works for LOGSPACE by using a variant of the Krivine abstract machine. It remains open whether this approach can be extended to CBPV.

CBPV, developed by Levy in 1999 [19], has been increasingly popular in recent decades. There are also various extensions of CBPV, including with stacks [21], with probability [10] and with call-by-need [22]. On the formalisation side, there is a formal equational theory [24] for CBPV; there is another formalisation [13] that includes proofs for its operational, equational, and denotational theory. On the applied side, there are projects such as extracting recurrences [15] using CBPV. There is a similar λ-calculus called the Bang-calculus [9], which can be regarded as an untyped version of CBPV without any side-effects.

2 Overview

Our goal is to show that CBPV is reasonable. This section is a high-level overview of our proof strategy, which is detailed in Section 6. The main theorems involved are:

▶ **Theorem 1** (Turing Machines Simulating CBPV). *Let T, $S \in \Omega(n)$. For a CBPV term s, if s reduces to a normal form t in time n and space m, then one can construct a Turing machine P_s simulating s that halts with output P_t simulating t, in time $\mathcal{O}(poly(T(n)))$ and space $\mathcal{O}(S(m))$.*

▶ **Theorem 2** (CBPV Simulating Turing Machines). *Let T, $S \in \Omega(n)$. For a Turing machine P_s that halts with output P_t in time n and space m, one can construct a CBPV term s simulating P_s which can be reduced to a normal form t simulating P_t in time $\mathcal{O}(poly(T(n)))$ and space $\mathcal{O}(S(n))$.*

That is, we must model the cost of CBPV using Turing machines and prove that the resulting simulation is cost-bounded (Theorem 1). Moreover, we must show that CBPV provides sufficient expressivity to reasonably simulate any Turing machine (Theorem 2).

Turing Machines Simulating CBPV

Inspired by the strategy used for proving that WCBV is reasonable [11], we verify that CBPV can be simulated by Turing machines with reasonable overhead by using two intermediate abstract machines: the *substitution machine* (Section 5.2) and the *heap machine* (Section 5.3).

It is well-known that the λ-calculus has the *size explosion problem*, where linear time can lead to exponential growth in space [26]. That is, with $\mathcal{O}(n)$ β-reduction steps, the largest intermediate term can be of size $\mathcal{O}(2^n)$. Turing machines, by contrast, need at least one unit of time to consume one unit of space, so space cost cannot exceed time cost. Hence, if one adopts a substitution-based strategy alone, the overhead in time will be exponential. To solve the size explosion problem, a shared memory structure is required to store values. This motivates why the heap-based strategy is incorporated into our simulation.

But the heap-based strategy has a *pointer explosion problem* [26], which makes space overhead non-constant. Luckily, size explosion and pointer explosion problems don't overlap [11], so we can use the heap-based strategy when a size explosion happens, and vice versa.

We formalise the simulation of CBPV by each of the two abstract machines, and verify that it respects the desired bounds in terms of time and space overhead. The cost bounds turn out to be similar to those for WCBV, enabling the adoption of an existing algorithm [11] to obtain a Turing machine simulation by interleaving the substitution and heap machines.

CBPV Simulating Turing Machines

For this direction, we use WCBV as an intermediate model. We first formalise the translation from WCBV to CBPV provided by Levy [19]. We then show that CBPV can simulate WCBV with reasonable time and space overheads. Since WCBV can simulate Turing machines with reasonable overheads [11, Theorem 5.1], we obtain our result for this direction as a corollary.

Formalisation

We verify in HOL4 that our time and space cost models for CBPV have the desired overheads. The formalisation covers all material presented in Section 3, Section 4, Section 5 and Section 6.2. An overview is given in Figure 6 in Section 6. The interleaving strategy, and the connection between abstract machines and Turing machines is done using pen-and-paper proofs adapted from the literature, in Section 6.

3 Call-By-Push-Value λ-Calculus

The CBPV λ-calculus [19, 20] allows encoding the order of evaluation as part of the syntax of a program. Hence, it serves as a subsuming paradigm that enables studying evaluation strategies, and combinations thereof, using a single set of reduction rules. Levy provides semantic-preserving translations from call-by-name and call-by-value into CBPV [20].

For simplicity, we use a core fragment of CBPV that is sufficient for demonstrating reasonability. As such we omit, for instance, the general pair types, and instead introduce a simpler double sequencing operation that will be discussed later. Furthermore, while most presentations of CBPV are typed, ours is untyped CBPV. We consider types an orthogonal concern to cost: the well-typed CBPV terms are a strict subset of the untyped CBPV terms, so a cost model for the latter immediately suggests a cost model for the former.

The CBPV terms are defined below as two mutually recursive sets: the *values* `V` and the *computations* `M`. The mutual recursion adds some technical difficulties in our formalisation as all relevant functions need to be mutually recursive too. For instance, the substitution function for CBPV, and the compilation function for compiling CBPV terms into programs are both defined mutually recursively, which complicates proofs. To simplify the presentation, we will often use a single overloaded name for two such mutually recursive functions.

$$\begin{array}{lll} \text{Values} & V & := \mathsf{var}\, x \mid \mathsf{thunk}\, M \\ \text{Computations} & M & := \lambda.\, M \mid \mathsf{app}\, M\, V \mid \mathsf{force}\, V \mid \mathsf{ret}\, V \mid \mathsf{seq}\, M\, M \mid \mathsf{pseq}\, \mathtt{M}\,\mathtt{M}\,\mathtt{M} \mid \mathsf{let}\, V.\, \mathsf{in}\, M \end{array}$$

Fine-grained control over evaluation can be achieved using the force and thunk operators. thunk suspends a computation, and force resumes a suspended computation.

Note that we have an extra pseq that is absent in the standard presentation of CBPV. An example of pseq computation is pseq $m_2\ m_1\ n$. It allows us to evaluate two computations m_1 and m_2 and use the results in a third computation n. Note that the notation within pseq follows the convention and binds to the right. This can of course be encoded with nested seq, but at the cost of higher binding depth: $\mathsf{seq}\, m_2\, (\mathsf{seq}\, m_1\, n)$. Avoiding this higher binder depth will turn out to be crucial for obtaining constant space overhead in Section 6.2. Including pairs in the language would have solved the problem too, but pseq suffices for our purposes.

We then formalise the big-step cost semantics of closed CBPV provided by Levy [20]; that is, we only consider terms with no free variables at the top level. Similar to the change we made in the syntax, we also add pseq as a special case of the pair type into our semantics.

In order to define the semantics for closed CBPV, we need to first provide a closed substitution function for β-reductions. The following function m^i_v substitutes all the variables with de Bruijn index i by v in m by recursively visiting all the inner terms of m:

$$\begin{array}{lcl@{\qquad}lcl}
(\lambda.\, m)^i_u & = & \lambda.\, (m^{i+1}_u) & (\mathsf{var}\, x)^i_u & = & u \text{ (if } x = i) \\
(\mathsf{app}\, m\, v)^i_u & = & \mathsf{app}\, (m^i_u)\, (v^i_u) & (\mathsf{var}\, x)^i_u & = & \mathsf{var}\, x \text{ (if } x \neq i) \\
(\mathsf{ret}\, v)^i_u & = & \mathsf{ret}\, (v^i_u) & (\mathsf{thunk}\, m)^i_u & = & \mathsf{thunk}\, (m^i_u) \\
(\mathsf{seq}\, m\, n)^i_u & = & \mathsf{seq}\, (m^i_u)\, (n^{i+1}_u) & (\mathsf{force}\, v)^i_u & = & \mathsf{force}\, (v^i_u) \\
(\mathsf{pseq}\, m_2\, m_1\, n)^i_u & = & \mathsf{pseq}\, ({m_2}^i_u)\, ({m_1}^i_u)\, (n^{i+2}_u) & (\mathsf{let}\, v.\, \mathsf{in}\, m)^i_u & = & \mathsf{let}\, (v^i_u).\, \mathsf{in}\, (m^{i+1}_u)
\end{array}$$

Note the special cases such as $(\lambda.\, m)^i_u$, where we need to increment the targeted variable index i accordingly because we are entering extra layers of abstractions. A more special case is $(\mathsf{pseq}\, m_2\, m_1\, n)^i_u$, we increment i by two for this substitution because n needs to leave two free variable names for the two computations m_1 and m_2.

We then define time cost and space cost semantics for CBPV. For the time cost semantics, we use a judgement $m \Downarrow_k n$ to mean that the computation m reduces to n in k steps. The rules are given in Figure 1.

$$\frac{}{\lambda.\, m \Downarrow_0 \lambda.\, m} \qquad \frac{}{\mathsf{ret}\, v \Downarrow_0 \mathsf{ret}\, v} \qquad \frac{m \Downarrow_k m'}{\mathsf{force}\, (\mathsf{thunk}\, m) \Downarrow_{k+2} m'} \qquad \frac{m^0_v \Downarrow_k m'}{\mathsf{let}\, v.\, \mathsf{in}\, m \Downarrow_{k+1} m'}$$

$$\frac{m \Downarrow_{k_1} \lambda.\, n \qquad n^0_v \Downarrow_{k_2} n'}{\mathsf{app}\, m\, v \Downarrow_{k_1+k_2+1} n'} \qquad \frac{m \Downarrow_{k_1} \mathsf{ret}\, v \qquad n^0_v \Downarrow_{k_2} n'}{\mathsf{seq}\, m\, n \Downarrow_{k_1+k_2+1} n'}$$

$$\frac{m_1 \Downarrow_{k_1} \mathsf{ret}\, v_1 \qquad m_2 \Downarrow_{k_2} \mathsf{ret}\, v_2 \qquad (n^0_{v_1})^1_{v_2} \Downarrow_{k_3} n'}{\mathsf{pseq}\, m_2\, m_1\, n \Downarrow_{k_1+k_2+k_3+1} n'}$$

Figure 1 The Rules Defining Big-Step Semantics of CBPV Terms with Time Cost.

For space, the judgement $m \Downarrow^s n$ says that m reduces to n with space cost s. Note that the time and space cost semantics judgements coincide if the cost annotations are ignored.

We define a size function $\|m\|$ for CBPV terms m as follows. Note that we account for the size of a de Bruijn index x in the term size.

$$\begin{array}{lcl@{\qquad}lcl}
\|\mathsf{var}\, x\| & = & 1 + x & \|\mathsf{thunk}\, m\| & = & 1 + \|m\| \\
\|\mathsf{force}\, v\| & = & 1 + \|v\| & \|\mathsf{let}\, v.\, \mathsf{in}\, m\| & = & 1 + \|v\| + \|m\| \\
\|\lambda.\, m\| & = & 1 + \|m\| & \|\mathsf{app}\, m\, v\| & = & 1 + \|m\| + \|v\| \\
\|\mathsf{ret}\, v\| & = & 1 + \|v\| & \|\mathsf{seq}\, m\, n\| & = & 1 + \|m\| + \|n\| \\
\|\mathsf{pseq}\, m_2\, m_1\, n\| & = & 1 + \|m_2\| + \|m_1\| + \|n\| & & &
\end{array}$$

Figure 2 gives inference rules of the space cost semantics. It tracks the maximum intermediate term size of an evaluation. For instance, for the pseq case, there are three different evaluation stages: (1). Evaluating m_1; (2). Evaluating m_2 (3). Substituting results v_1 and v_2 into n. The space cost is the maximum size among these stages.

$$\frac{}{\lambda.\, m \Downarrow^{\|m\|+1} \lambda.\, m} \qquad \frac{}{\mathsf{ret}\, v \Downarrow^{\|v\|+1} \mathsf{ret}\, v} \qquad \frac{m \Downarrow^{s} m'}{\mathsf{force}\,(\mathsf{thunk}\ m) \Downarrow^{\max(s,\, \|m\|+2)} m'}$$

$$\frac{m^0_v \Downarrow^{s} m'}{\mathsf{let}\, v.\, \mathsf{in}\, m \Downarrow^{\max(s,\, \|v\|+\|m\|+1)} m'} \qquad \frac{m \Downarrow^{s_1} \lambda.\, n \qquad n^0_v \Downarrow^{s_2} n'}{\mathsf{app}\, m\, v \Downarrow^{\max(s_1+\|v\|+1,\, s_2)} n'}$$

$$\frac{m \Downarrow^{s_1} \mathsf{ret}\, v \qquad n^0_v \Downarrow^{s_2} n'}{\mathsf{seq}\, m\, n \Downarrow^{\max(s_1+\|n\|+1,\, s_2)} n'}$$

$$\frac{m_1 \Downarrow^{s_1} \mathsf{ret}\, v_1 \qquad m_2 \Downarrow^{s_2} \mathsf{ret}\, v_2 \qquad (n^0_{v_1})^1_{v2} \Downarrow^{s_3} n'}{\mathsf{pseq}\, m_2\, m_1\, n \Downarrow^{\max(s_1+\|m_2\|+\|n\|+1,\, \|v_1\|+s_2+\|n\|+1, s_3)} n'}$$

Figure 2 The Rules Defining Big-Step Semantics of CBPV Terms with Space Cost.

4 Compiling CBPV Terms to Programs

As a first step in bridging the gap between CBPV and Turing machines, we define a flat data structure to represent *programs* that correspond to CBPV terms. A program P is formed of a list of *tokens*, Tok that are defined as follows:

$$\begin{array}{lcl} \mathsf{t} \in \mathsf{Tok} & := & \mathsf{varT}\, x \mid \mathsf{thunkT} \mid \mathsf{endThunkT} \mid \mathsf{lamT} \mid \mathsf{endLamT} \mid \mathsf{appT} \mid \mathsf{forceT} \mid \\ & & \mathsf{retT} \mid \mathsf{endRetT} \mid \mathsf{seqT} \mid \mathsf{endSeqT} \mid \mathsf{pseqT} \mid \mathsf{endPseqT} \mid \mathsf{letT} \mid \mathsf{endLetT} \end{array}$$

▶ **Definition 3** (Size of Tokens and Programs).

$$\begin{array}{lcl} |\mathsf{varT}\, x| & = & 1 + x \qquad\qquad |t| \;=\; 1 \qquad \textit{otherwise} \\ \|P\| & = & 1 + \sum_{t_i \in P} |t_i| \end{array}$$

The de Bruijn index x counts towards the token size because larger indices require more tape cells to store on Turing machines. The size of a program is simply the sum of the size of its tokens plus 1 (which is the size of the empty program on a Turing machine).

Definitions 4 and 5 define compilation to the substitution and heap machines, respectively.

▶ **Definition 4** (Compilation Function for Substitution Machine).

$$\begin{array}{lcl} \gamma(\mathsf{var}\, x) & = & \mathsf{varT}\, x \\ \gamma(\mathsf{thunk}\ m) & = & \mathsf{thunkT} :: \gamma(m) +\!\!+ [\mathsf{endThunkT}] \\ \gamma(\mathsf{force}\, v) & = & \gamma(v) +\!\!+ [\mathsf{forceT}] \\ \gamma(\mathsf{ret}\, v) & = & \mathsf{retT} :: \gamma(v) +\!\!+ [\mathsf{endRetT}] \\ \gamma(\lambda.\, m) & = & \mathsf{lamT} :: \gamma(m) +\!\!+ [\mathsf{endLamT}] \\ \gamma(\mathsf{app}\, m\, v) & = & \gamma(m) +\!\!+ \gamma(v) +\!\!+ [\mathsf{appT}] \\ \gamma(\mathsf{seq}\, m\, n) & = & \gamma(m) +\!\!+ [\mathsf{seqT}] +\!\!+ \gamma(n) +\!\!+ [\mathsf{endSeqT}] \\ \gamma(\mathsf{pseq}\, m_2\, m_1\, n) & = & \gamma(m_1) +\!\!+ \gamma(m_2) +\!\!+ [\mathsf{pseqT}] +\!\!+ \gamma(n) +\!\!+ [\mathsf{endPseqT}] \\ \gamma(\mathsf{let}\, v.\, \mathsf{in}\, m) & = & \gamma(v) +\!\!+ [\mathsf{letT}] +\!\!+ \gamma(m) +\!\!+ [\mathsf{endLetT}] \end{array}$$

▶ **Definition 5** (Compilation Function for Heap Machine). *We define γ' exactly as γ, except:*

$$\gamma'(\mathsf{ret}\, v) = \gamma'(v) \mathbin{+\!\!+} [\mathsf{retT}\,]$$

We use pairs of delimiter tokens (like seqT and endSeqT) when necessary, to preserve the tree structure of the original term. The development of these compilers was fiddly, since the right balance needs to be struck between including enough structure to prevent different subterms from being conflated or evaluated prematurely, yet not too much structure, because extra structure takes space, and must be accounted for in the space cost bound proofs. γ and γ' make slightly different tradeoffs in this respect. We could change the substitution machine to use γ', but the overall proof does not require the machines to use the same syntax.

The following lemma is useful for our space cost analysis. It states that the size of a compiled program is linear wrt. term size. (Note that the same lemma also works for γ'.)

▶ **Lemma 6** (Program Size Bounds). $1 \le \|m\| \le \|\gamma(m)\| + 1 \le 2 * \|m\|$

We write $P \gg m$ to state that P is the corresponding program for m.

▶ **Definition 7** (Program-Term Correspondence). *$P \gg m$ holds if $\gamma(m) = P$.*

5 Abstract Machines

Recall from Section 2 that our proof strategy relies on interleaving two simulation strategies to achieve reasonability in time and space. The substitution-based strategy has reasonable overhead for space, but not for time due to the size explosion problem. The heap-based strategy has reasonable overhead for time, but not for space due to the pointer explosion problem. These two explosion problems do not occur at the same time. Thus, by interleaving the respective Turing machines for each of these two strategies, we can obtain a reasonable simulation. In order to achieve this, we first implement two abstract machines that represent these two strategies respectively in this section. We then construct the corresponding Turing machines and finish the rest of the proofs in Section 6. Note that the size of intermediate terms and the complexity differs between the abstract machines and their corresponding Turing machines.

In this section, we first introduce an auxiliary extraction function that is used by both abstract machines (Section 5.1). We then introduce the substitution machine (Section 5.2) and the heap machine (Section 5.3), and investigate their cost in relation to CBPV.

5.1 Extraction Function

For each pair of delimiter tokens, we define a φ function to scan a program until the corresponding end delimiter. To simplify the presentation, we overload φ to account for all operands. The idea is that $\varphi P = (M, Q)$ strips the argument body out of P and returns it as M, where Q is the rest of the program. We show the extraction function for lamT - endLamT below. The intuition is similar to finding matching parentheses pairs. When φ is applied to a well-formed P, we have $\varphi P = \varphi(M :: [\mathsf{endLamT}\,] \mathbin{+\!\!+} Q) = (M, Q)$, where M has balanced lamT - endLamT pairs.

▶ **Definition 8** (Extraction Function for lamT - endLamT). *$\varphi P = \varphi^{0}_{[]} P$ where:*

$$\begin{aligned}
\varphi^{0}_{M}(\mathsf{endLamT}\, :: Q) &= (M, Q) & \qquad \varphi^{k}_{M}(t :: Q) &= \varphi^{k}_{M \mathbin{+\!\!+} [t]} Q \\
\varphi^{k}_{M}(\mathsf{lamT}\, :: Q) &= \varphi^{k+1}_{M \mathbin{+\!\!+} [\mathsf{lamT}\,]} Q & \qquad \varphi^{k}_{M} [] &= \textit{undefined} \\
\varphi^{k+1}_{M}(\mathsf{endLamT}\, :: Q) &= \varphi^{k}_{M \mathbin{+\!\!+} [\mathsf{endLamT}\,]} Q
\end{aligned}$$

5.2 Substitution Machine

In this section, we develop a machine that implements a substitution-based evaluation strategy. Before diving into the transition rules, we need a helper function $::_{tc}$ that is used to prevent empty lists from accumulating on the stack. It is defined recursively as follows:

$$[\,] ::_{tc} C \quad = \quad C \qquad\qquad c ::_{tc} C \quad = \quad c ::_{tc} C \quad (c \neq [\,])$$

The substitution machine performs substitutions immediately as they appear at the top of the current stacks. The machine state consists of two stacks: the task stack and the value stack. In the initial state, the value stack is empty and the task stack contains the $\gamma(m)$ for a CBPV computation m. On successful termination, a final value is produced on the value stack, and the task stack is empty. Note that the value stack (despite its name) will sometimes contain computations in non-final states. An alternative presentation would be to add an extra stack for suspended computations, but we found no need for this.

Substitution on programs, written, P^i_Q is similar to that for CBPV terms (Section 3); its definition is elided. Figure 3 shows the transition rules for the substitution machine. The three columns represent the task stack, the value stack, and the assumptions. Each $\rhd$ represents one transition step, where the row above $\rhd$ represents the current state of the machine, and the row on the same level as $\rhd$ represents the next state of the machine after the transition. For example, the transition rule for thunkT strips one layer of thunkT-$\mathsf{endThunkT}$ off the task stack and places it on the value stack (thus suspending it). Note that the transition rules for seqT and pseqT can strip retT components from the value stack directly without the extraction function φ. For instance, in the seqT rule, $\mathsf{retT} :: U +\!\!+ [\mathsf{endRetT}]$ is just the first element on the value stack. We can strip it with a simple list operation. Furthermore, since there is no extra subsequent programs after $\mathsf{endRetT}$ in $\mathsf{retT} :: U +\!\!+ [\mathsf{endRetT}]$, we can obtain U by removing retT and $\mathsf{endRetT}$ using simple list operations.

The transition rule for varT is not strictly necessary: we only consider closed terms, so the rule will never be exercised when running the compilation output. Nonetheless, including it appears to make the proofs more ergonomic, by making it unnecessary to carry around a closedness side condition. For example, consider the useful technical lemma

$$((\gamma(v) :: P) :: T, V) \rhd (P ::_{tc} T, \gamma(v) :: V)$$

which holds unconditionally when the varT rule is present in the substitution machine semantics. If we remove the rule, it only holds when v is a thunk.

Multiple transitions are written $(T, V) \rhd^\sigma_k (T', V')$, where T is the current task stack and V is the current value stack. We obtain a new state (T', V') after applying the transition rules k times on the current state (T, V), with the size of the biggest intermediate state being $\frac{1}{2}\sigma$. We elide σ or k when irrelevant. We write $\rhd_*$ to represent 0 or more transition steps.

The substitution machine simulates CBPV with constant time and space overhead:

▶ **Lemma 9** (Substitution Machine Time Simulation). *If $m \Downarrow_k n$, then there exists k' such that $(P_m, []) \rhd_{k'} ([], P_n)$ where $k' \leq 3 * k + 1$ and $P_m \gg m$ and $P_n \gg n$.*

Proof. By rule induction on the big-step semantics of CBPV. ◀

▶ **Lemma 10** (Substitution Machine Space Simulation). *If $m \Downarrow^s n$ then there exists σ such that $(P_m, []) \rhd^\sigma_* ([], P_n)$, where $s \leq \sigma \leq 9 * s$ and $P_m \gg m$ and $P_n \gg n$.*

Proof. Similar to the time simulation proof, we induct on the structure of the big-step semantics of CBPV and show that this theorem is true for all the transition rules. ◀

Task Stack	Value Stack	Assumption
$(\mathsf{varT}\, n :: P) :: T$	V	
$\triangleright$ $P ::_{tc} T$	$\mathsf{varT}\, n :: V$	
$(\mathsf{thunkT} :: P) :: T$	V	$\varphi P = (M, Q)$
$\triangleright$ $Q ::_{tc} T$	$\mathsf{thunkT} :: M +\!\!+ [\mathsf{endThunkT}] :: V$	
$(\mathsf{forceT} :: P) :: T$	$(\mathsf{thunkT} :: K) :: V$	$\varphi K = (M, [\,])$
$\triangleright$ $(M +\!\!+ P) ::_{tc} T$	V	
$(\mathsf{lamT} :: P) :: T$	V	$\varphi P = (M, Q)$
$\triangleright$ $Q ::_{tc} T$	$(\mathsf{lamT} :: M +\!\!+ [\mathsf{endLamT}]) :: V$	
$(\mathsf{appT} :: P) :: T$	$Q :: (\mathsf{lamT} :: M +\!\!+ [\mathsf{endLamT}]) :: V$	
$\triangleright$ $M^0_Q :: (P ::_{tc} T)$	V	
$(\mathsf{retT} :: P) :: T$	V	$\varphi P = (U, Q)$
$\triangleright$ $Q ::_{tc} T$	$(\mathsf{retT} :: U +\!\!+ [\mathsf{endRetT}]) :: V$	
$(\mathsf{seqT} :: P) :: T$	$(\mathsf{retT} :: U +\!\!+ [\mathsf{endRetT}]) :: V$	$\varphi P = (N, Q)$
$\triangleright$ $N^0_U :: (Q ::_{tc} T)$	V	
$(\mathsf{pseqT} :: P) :: T$	$(\mathsf{retT} :: U_2 +\!\!+ [\mathsf{endRetT}]) :: (\mathsf{retT} :: U_1 +\!\!+ [\mathsf{endRetT}]) :: V$	$\varphi P = (N, Q)$
$\triangleright$ $(N^0_{U_1})^1_{U_2} :: (Q ::_{tc} T)$	V	
$(\mathsf{letT} :: P) :: T$	$K :: V$	$\varphi P = (M, Q)$
$\triangleright$ $M^0_K :: (Q ::_{tc} T)$	V	

Figure 3 Transition Rules for the Substitution Machine.

5.3 Heap Machine

In this section, we introduce an environment-based abstract machine. Free variables are interpreted as pointers into a heap that store their values. But first, some auxiliary definitions.

▶ **Definition 11** (Closure). *A closure $\langle P, a\rangle$ is a pair consisting of a program P and pointer a. The pointer a binds the free variables in the program P to the values in the heap.*

▶ **Definition 12** (Heap). *A heap is defined as a list of heap cells where each cell $\{C, a\}$ consists of a closure C and an additional pointer a. The pointer a points to the previous cell in the heap, providing a linked list representation of the heap.*

Let $+\!\!+$ be list concatenation, and *len* be the standard length function for lists.

▶ **Definition 13** (Put and Lookup). *In lookup, let $H[a] = \{C, a'\}$.*

$$\begin{aligned} &put\ H\ e &&= (H +\!\!+ [e],\ len(H)) \\ &lookup\ H\ a\ 0 &&= C \\ &lookup\ H\ a\ x &&= lookup\ H\ a'\ (x-1) \quad (if\ x \neq 0) \end{aligned}$$

The states of the heap machine are triplets consisting of a task stack, a value stack and a heap. We store values in the heap, and replace variables with pointers instead of directly substituting values in-place.

The transition rules for the heap machine are shown in Figure 4. Compared to Section 5.2, there are some minor differences that are not directly related to substitution, but are convenient in the proofs. For example, we spell out the thunkT structure for the forceT case, so this rule does not have to use the φ function.

In our proofs, we write heap machine transitions as $(T, V, H) \triangleright^{\sigma}_{k} (T', V', H')$, where (T, V, H) is a triple representing the current task stack, value stack, and heap. We obtain a new state (T', V', H') after applying the transition rules k times on the current state, with the size of the biggest intermediate state being σ. We elide σ or k when irrelevant.

	Task Stack	Value Stack	Heap	Assumption
	$\langle \mathsf{varT}\, x :: P,\, a\rangle :: T$	V	H	$\mathsf{lookup}\, H\, a\, x = g$
$\triangleright$	$\langle P,\, a\rangle :: T$	$g :: V$	H	
	$\langle \mathsf{thunkT} :: P,\, a\rangle :: T$	V	H	$\varphi P = (M, Q)$
$\triangleright$	$\langle Q,\, a\rangle :: T$	$\langle \mathsf{thunkT} :: M +\!\!+ [\mathsf{endThunkT}],\, a\rangle :: V$	H	
	$\langle \mathsf{forceT} :: P,\, a\rangle :: T$	$\langle \mathsf{thunkT} :: M +\!\!+ [\mathsf{endThunkT}],\, b\rangle :: V$	H	
$\triangleright$	$\langle M,\, b\rangle :: \langle P,\, a\rangle :: T$	V	H	
	$\langle \mathsf{lamT} :: P,\, a\rangle :: T$	V	H	$\varphi P = (M, Q)$
$\triangleright$	$\langle Q,\, a\rangle :: T$	$\langle \mathsf{lamT} :: M +\!\!+ [\mathsf{endLamT}],\, a\rangle :: V$	H	
	$\langle \mathsf{appT} :: P,\, a\rangle :: T$	$Q :: \langle \mathsf{lamT} :: M +\!\!+ [\mathsf{endLamT}],\, b\rangle :: V$	H	$\mathsf{put}\ H\{Q,\, b\} = (H', c)$
$\triangleright$	$\langle M,\, c\rangle :: \langle P,\, a\rangle :: T$	V	H'	
	$\langle \mathsf{retT} :: P,\, a\rangle :: T$	$\langle M,\, b\rangle :: V$	H	
$\triangleright$	$\langle P,\, a\rangle :: T$	$\langle M,\, b\rangle :: V$	H	
	$\langle \mathsf{seqT} :: P,\, a\rangle :: T$	$\langle M,\, b\rangle :: V$	H	$\varphi P = (N, Q)$ $\mathsf{put}\ H\{\langle M,\, b\rangle,\, a\} = (H', c)$
$\triangleright$	$\langle N,\, c\rangle :: \langle Q,\, a\rangle :: T$	V	H'	
	$\langle \mathsf{pseqT} :: P,\, a\rangle :: T$	$\langle M_2,\, b_2\rangle :: \langle M_1,\, b_1\rangle :: V$	H	$\varphi P = (N, Q)$ $\mathsf{put}\ H\{\langle M_2,\, b_2\rangle,\, a\} = (H_1, c_1)$ $\mathsf{put}\ H_1\{\langle M_1,\, b_1\rangle,\, a\} = (H_2, c_2)$
$\triangleright$	$\langle N,\, c_2\rangle :: \langle Q,\, a\rangle :: T$	V	H_2	
	$\langle \mathsf{letT} :: P,\, a\rangle :: T$	$K :: V$	H	$\varphi P = (M, Q)$ $\mathsf{put}\ H\{K,\, a\} = (H', b)$
$\triangleright$	$\langle M,\, b\rangle :: \langle Q,\, a\rangle :: T$	V	H'	
	$\langle [\,],\, a\rangle :: T$	V	H	
$\triangleright$	T	V	H	

Figure 4 Transition Rules for the Heap Machine.

Correspondence between programs and CBPV terms is here relative to an environment. We therefore use the *unfolding* judgement, written $(H,\, a|\, m_1) \rightsquigarrow_k m_2$, which takes as inputs a heap H, a pointer a, a variable bound number k, and two CBPV terms m_1 and m_2. It returns true when m_2 is identical to m_1 with all of its free variables replaced by their values in H. A representative subset of the unfolding rules are shown in Figure 5. The two variable rules are the interesting ones. The first variable rule unfolds a bound variable to itself. The second variable rule (at the bottom) unfolds free variables to their value on the heap.

We define a heap-aware correspondence using the unfolding function as follows:

▶ **Definition 14** (Closure-Term Correspondence with Heap). *: For any closure $\langle P,\, a\rangle$ with heap H and CBPV term n, $\langle P,\, a\rangle$ is the corresponding closure for n (written as $\langle P,\, a\rangle \gg_H n$) if and only if there exists a CBPV term m such that $(H,\, a|\, m) \rightsquigarrow_0 n$ and $\gamma'(m) = P$.*

We prove in HOL4 that the heap machine can simulate CBPV with constant overhead in time:

▶ **Lemma 15** (Heap Machine Time Simulation). *If $m \Downarrow_k n$ then there exists k' such that $(\langle P_m,\, 0\rangle, [\,], [\,]) \triangleright_{k'} ([\,], \langle P_n,\, a\rangle, H)$, where $k' \leq 10 * k + 3$ and $\langle P_m,\, 0\rangle \gg m$ and $\langle P_n,\, a\rangle \gg_H n$.*

Proof. By rule induction on the big-step semantics of CBPV. ◀

The space cost is unrelated to that of the CBPV reduction. Instead, the size of the k^{th} state is bounded by the size of the original CBPV term, and the number of steps:

▶ **Lemma 16** (Heap Machine Space Simulation). *If $(P_m, [\,], [\,]) \triangleright_k (T, N, H)$ and $P_m \gg m$ then $\|T +\!\!+ N +\!\!+ H\| \leq (3k+1) * (4 * k + 2 * \|m\|)$.*

Proof. The proof proceeds by showing each stack's length and that all elements in each stack are bounded by the size of the original term m and the number of transitions taken k. ◀

$$\frac{x < k}{(H,\ a|\,\mathsf{var}\,x) \rightsquigarrow_k \mathsf{var}\,x} \qquad \frac{(H,\ a|\,m) \rightsquigarrow_{k+1} m'}{(H,\ a|\,\lambda.\,m) \rightsquigarrow_k \lambda.\,m'} \qquad \frac{(H,\ a|\,v) \rightsquigarrow_k v'}{(H,\ a|\,\mathsf{ret}\,v) \rightsquigarrow_k \mathsf{ret}\,v'}$$

$$\frac{(H,\ a|\,m_1) \rightsquigarrow_k m_1' \qquad (H,\ a|\,m_2) \rightsquigarrow_k m_2' \qquad (H,\ a|\,n) \rightsquigarrow_{k+2} n'}{(H,\ a|\,\mathsf{pseq}\,m_2\,m_1\,n) \rightsquigarrow_k \mathsf{pseq}\,m_2'\,m_1'\,n'}$$

$$\frac{k \le x \qquad \mathsf{lookup}\,H\,a\,(x-k) = \langle \gamma(\mathsf{thunk}\,m),\,b\rangle \qquad (H,\ b|\,\mathsf{thunk}\,m) \rightsquigarrow_0 v}{(H,\ a|\,\mathsf{var}\,x) \rightsquigarrow_k v}$$

Figure 5 A Selection of Unfolding Rules.

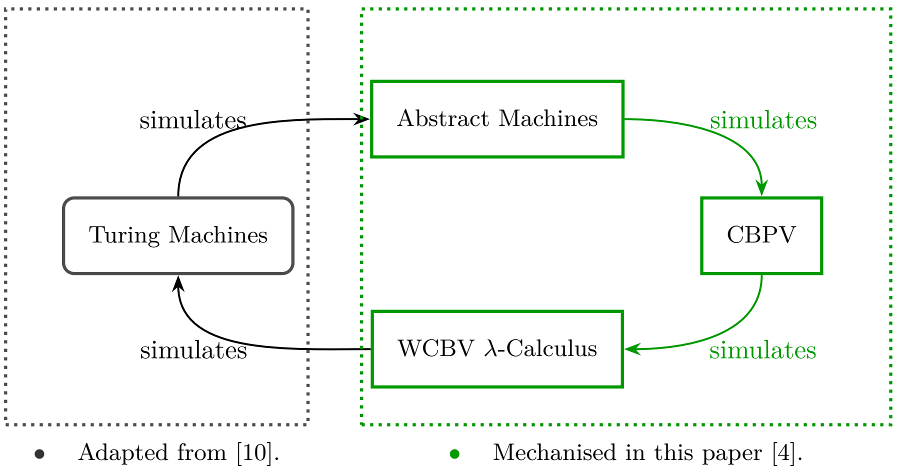

Figure 6 Simulation between Turing Machines and CBPV with Intermediate Models.

6 CBPV is Reasonable for Both Time and Space

In this section, we use the results from Section 5 to prove that CBPV is reasonable. We achieve this by providing two simulations between CBPV and Turing machines, as elaborated in Figure 6.

Section 6.1 describes how Turing machines can simulate CBPV with polynomial time overhead and constant factor space overhead, fulfilling Theorem 1 in Section 2. Note that this simulation goes through the abstract machines we implemented and formalised in Section 5.

Section 6.2 describes how CBPV can reasonably simulate WCBV. We adapt the result that WCBV can reasonably simulate Turing machines from existing literature [11]. Together, we have that CBPV can simulate Turing machines with polynomial time overhead and constant factor space overhead, fulfilling Theorem 2 in Section 2.

6.1 Turing Machines Simulating CBPV

In this section we show that it is always possible to construct a Turing machine M_{CBPV} that simulates CBPV with polynomially bounded time overhead and constant factor space overhead. The results from Section 5, specifically, Lemmas 9, 10, 15, and 16, will be used here.

Our proof is very similar to a proof from the literature [11], where similar heap and substitution machines are interleaved to obtain a simulation of WCBV rather than CBPV. Their proof relies on formalised results similar to our formalised results, and proves that the interleaving strategy always obeys the required time and space bounds. We demonstrate that their argument extends to the more general case of CBPV. While our abstract machines are much more involved, they have similar time and space bounds, which simplifies adaptation.

We need to construct Turing machines M_{subst} and M_{heap} that simulate the respective abstract machines. We must then prove that one can always construct a Turing machine M_{CBPV} that interleaves M_{subst} and M_{heap} such that they always obey the required bounds.

The substitution-based Turing machine M_{subst} is constructed by iterating over smaller Turing machines each simulating an individual transition rule from Figure 3. Similarly, the heap-based Turing machine M_{heap} is constructed by iterating over Turing machines simulating the rules of Figure 4. In addition to the original CBPV term s, each of these machines takes as input a number of steps k that they are meant to simulate. The M_{subst} takes an additional input σ and aborts if the amount of space used is larger than σ. The encoding of the machine transition rules in Turing machines is straightforward, where we use 13 symbols to represent the 13 constructors(tokens) in the substitution machine. Similarly, we use 12 symbols to represent the 12 constructors(tokens) in the heap machine.

The algorithm then constructs M_{CBPV} by interleaving the above two machines, M_{subst} and M_{heap}. Provided an input s which has a normal form t, the algorithm starts by applying M_{subst} over (the program equivalent of) s to reduce it. If a size explosion occurs during this reduction, execution then switches to use M_{heap} before the explosion happens. In this case, the heap-based strategy is guaranteed not to encounter the pointer explosion problem [11]. That is because the terms that cause size explosions result in a space cost of $\mathcal{O}(n^2)$, which easily accommodates the $\log n$ space cost for pointer storage in the heap machine. This algorithm guarantees termination and reasonable time and space overhead.

Now let's construct the Turing machines. In the following theorems, we write $\|s\|_T$ for the number of transition steps and $\|s\|_S$ for size of the biggest intermediate term.

We first consider the substitution machine. M_{subst} takes as inputs a term s, two numbers k and σ. s is the term to be reduced, k represents the number of transition steps to perform, and σ represents the space threshold over which the machine should abort. We show that this machine must satisfy one of the following three conditions: (1) it returns a desired value t within k steps; (2) it reaches the space bound and halts; (3) it finishes k reductions and halts (within both space and time bounds). This is formally stated below as Theorem 17.

▶ **Theorem 17.** *There exists a Turing machine M_{subst} that takes as inputs k, σ and a term s. It halts in time $\mathcal{O}(k\cdot$ poly(min $(\sigma, \|s\|_S)))$ and space $\mathcal{O}($min $(\sigma, \|s\|_S)$ + log σ + log $k)$ while one of the following statements holds:*

- *The machine outputs a term t, then s has normal form t and $\sigma \geq \|s\|_S$ and $k \geq 3\cdot\|s\|_T+1$.*
- *The machine halts in a state named space bound not reached and $k \leq 3\cdot\|s\|_T + 1$ holds.*
- *The machine halts in a state named space bound reached and $\sigma \leq 9\cdot\|s\|_S$ holds.*

Proof. We can construct a Turing machine M_{subst} by looping Turing machines that implement the individual steps of the abstract substitution machine. We add an extra rule where this machine has to halt if it were to reach a state with size larger than σ. Note that the extra rule requires M_{subst} to halt even if it is in the middle of execution, in order to avoid the size explosion problem. With an initialisation function τ that converts s from a λ-term into a program $\tau(s)$, we now have the desired Turing machine M_{subst}.

The time and space cost from the substitution machine mostly carry over directly, but the extraction functions φ cannot be implemented in constant time. For instance, $\varphi^k_Q P$ takes time and space $\mathcal{O}(\|Q\| + \|P\| + k)$.

From Lemma 10, we know that if $s \Downarrow^{\|s\|_S} t$ then there exists σ such that $(P_s, []) \triangleright^{\sigma}_{*} ([], P_t)$, where $\|s\|_S \leq \sigma \leq 9 * \|s\|_S$ and $P_s \gg s$ and $P_t \gg t$. Thus the size of all intermediate states and the overall space consumption lie within $9 * \|s\|_S$.

From Lemma 9, we know that if $s \Downarrow_{\|s\|_T} t$, then there exists k such that $(P_s, []) \triangleright_k ([], P_t)$ where $k \leq 3 * \|s\|_T + 1$ and $P_s \gg s$ and $P_t \gg t$. Thus there must exist a k that is large enough to simulate the reduction while still lying within the bound $3 * \|s\|_T + 1$. ◀

The next step is to construct the heap-based Turing machine M_{heap}:

▶ **Theorem 18.** *There exists a Turing machine M_{heap} that, given a number k and a closed term s, halts in time $\mathcal{O}(poly(\|s\|, k))$ and space $\mathcal{O}(\|s\| \cdot poly(k))$. If s has a normal form t and $k \geq 10 \cdot \|s\|_T + 3$, 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. We can implement the abstract substitution machine from Section 5.3 by looping Turing machines that implement the individual steps of the abstract machine.

Time and space cost of the φ functions is as in Theorem 17. We also have to consider the lookup function. lookup iterates through the heap for n indices using the heap headers (a in this case) as pointers, resulting in at most $\mathcal{O}(n)$ time cost.

Thus each abstract step $(T, V, H) \triangleright (T', V', H')$ can be implemented in $\mathcal{O}(poly(\|(T, V, H)\|))$ time and $\mathcal{O}(max(\|(T, V, H)\|, \|(T', V', H')\|))$ space. The space consumption of all involved operations in Figure 4 is bounded by their input or output. Using Lemma 16, 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 15. ◀

Before constructing M_{CBPV}, we need a lemma to say that unfolding only changes de Bruijn indices starting at k. In particular, closed terms are invariant under unfolding.

▶ **Lemma 19.** *If s is bounded by k, then $(s, a|\, H) \rightsquigarrow_k s$.*

Proof. Induction on $s \leq k$. ◀

We now combine everything together to form our final theorem.

▶ **Theorem 20.** *There is a Turing machine M_{CBPV} 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 $\mathcal{O}(poly(\|s\|, \|s\|_T))$ and space $\mathcal{O}(\|s\|_S)$.*

Proof. By using the interleaving algorithm (Algorithm 1) adapted from WCBV [11].

Algorithm 1 Interleaving Strategy Algorithm.

```
Let p be the polynomial such that the machine from Theorem 18 runs in space $\mathcal{O}(\|s\| \cdot p(k))$.
1. Initialise $k := 0$ (in binary)
2. Compute $\sigma := \|s\| \cdot p(k)$ (in binary)
3. Run $M_{subst}$ on $s$, $k$ and $\sigma$.
   a. If $M_{subst}$ computes the normal form $t$, output $(\gamma(t), 0)$ and an empty heap [ ] and halt.
   b. If $M_{subst}$ halts with space bound not reached, set $k := k + 1$ and go to 2
   c. If $M_{subst}$ halts with space bound reached, continue at 4.
4. Run $M_{heap}$ on $s$ and $k$.
   a. If this computed a closure $g$ and a heap $H$ representing $t$, output $H$ and $g$ and halt.
   b. Otherwise, set $k := k + 1$ and go to 2.
```

There are four things to prove about the algorithm. We will go through them one by one.

Halting states. M_{CBPV} has only two halting states: when M_{subst} returns (state 3(a) in Algorithm 1), and when M_{heap} returns (state 4(a)). In both cases, M_{CBPV} returns a closure-heap pair representing the normal form t of s. In the first case, Theorem 17 shows that M_{subst} will return a normal form t of s and Lemma 19 shows that the closure-heap pair we constructed in state 3(a) indeed represents t. The second case is immediate Theorem 18.

Termination. The machine will terminate for terminating terms s and diverge on non-terminating CBPV terms. For the terminating case, we need to show two things: (1). for all k, each iteration eventually finishes and goes to the next iteration; (2). there exists a k such that the machine halts and returns a closure-heap pair. We consider (1) first, and fix k. Time cost in step 1 is constant. Binary computation in Turing machines have polynomial cost, thus the time cost in step 2 is $\mathcal{O}(poly(\|s\|,(k)))$. Using Theorem 17, step 3 takes time

$$\begin{array}{rcll}\mathcal{O}(k\cdot poly(min(\sigma,\|s\|_S))) & \subseteq & \mathcal{O}(k\cdot poly(\sigma)) & \\ & = & \mathcal{O}(k\cdot poly(\|s\|\cdot p(k))) & \text{from step 2}\\ & \subseteq & \mathcal{O}(k\cdot poly(\|s\|,k)) & p \text{ is a polynomial}\\ & \subseteq & \mathcal{O}(poly(\|s\|,k)) & \end{array}$$

If Step 4 is executed, this takes time $\mathcal{O}(poly(\|s\|,k))$ by Theorem 18. Since each of the four steps has at most time complexity $\mathcal{O}(poly(\|s\|,k))$, one iteration has time cost $\mathcal{O}(poly(\|s\|,k))$, which suffices for (1). For (2), consider $k = 10\|s\|_T + 3$, which is larger than the two values required in Theorem 17 and Theorem 18. By Theorem 17, the machine does halt during Step 3, unless $\sigma \le \|s\|_S$. In the latter case, 4 is tried. Then, by Theorem 18, as k is large enough, we have that M_{heap} indeed halts with a closure-heap pair.

Time Complexity. We have proved the time cost for each iteration, so we just need to sum up all the iterations for the overall time complexity for M_{CBPV}:

$$\mathcal{O}\Bigg(\sum_{k=0}^{10\|s\|_T+3}(poly(\|s\|,k))\Bigg) \subseteq \mathcal{O}(\|s\|_T\cdot poly(\|s\|,\|s\|_T)) \subseteq \mathcal{O}(poly(\|s\|,\|s\|_T))$$

Space Complexity. We first analyse the space cost for one iteration with an arbitrary k. Step 1 is constant. Step 2 takes $\mathcal{O}(log(\sigma))$ space since the computation is in binary. By Theorem 17, Step 3 takes space $\mathcal{O}(min(\sigma,\|s\|_S)+\log\sigma+\log k) \subseteq \mathcal{O}(\|s\|_S+\log\sigma+\log k)$. If step 3(c) reaches the space bound ($\sigma \le 3\cdot\|s\|_S$), step 4 is tried. Together with Theorem 18 and definition of m, the space cost for step 4 is $\mathcal{O}(\|s\|\cdot p(k)) \subseteq \mathcal{O}(\sigma) \subseteq \mathcal{O}(\|s\|_S)$.

Thus we have the space cost of one iteration:

$$\begin{array}{rcll}\mathcal{O}(\|s\|_S+\log\sigma+\log k) & = & \mathcal{O}(\|s\|_S+\log(\|s\|\cdot p(k))+\log k) & \text{(definition of } \sigma)\\ & \subseteq & \mathcal{O}(\|s\|_S+\log\|s\|+\log(p(k))+\log k) & \\ & = & \mathcal{O}(\|s\|_S+\log(p(k))+\log k) & (\text{as } \|s\|\le\|s\|_S)\\ & = & \mathcal{O}(\|s\|_S+\log(k^c)+\log k) & (c \text{ const., } p \text{ poly.})\\ & = & \mathcal{O}(\|s\|_S+c\log(k)+\log k) & \\ & \subseteq & \mathcal{O}(\|s\|_S+\log k) & \end{array}$$

The space cost for all iterations is as follows, where the last equation is by Lemma 21:

$$\mathcal{O}(\max_{0\le k\le 10\|s\|_T+3}(\|s\|_S+\log k)) \subseteq \mathcal{O}(\|s\|_S+\log\|s\|_T) = \mathcal{O}(\|s\|_S) \qquad \blacktriangleleft$$

By combining our formalisation with results above, we obtain Theorem 2.

For terms with $\|s\|_T \notin \mathcal{O}(\|s\|_S)$ it is crucial that the machine tracks the step number k in binary, because it would need $\Omega(\|s\|_T)$ space otherwise. This suffices because of Lemma 21:

▶ **Lemma 21.** $\log \|s\|_T \in \mathcal{O}(\|s\|_S)$.

Proof. As the vocabulary is finite, there are at most exponentially many terms for a given size. A reduction from s cannot visit the same term twice since reduction is deterministic. $\|s\|_S$ is the biggest intermediate term, which means that all the terms in the reduction for s will be smaller than $\|s\|_S$. This implies that $\|s\|_T$ will be at most equal to the total number of terms smaller than $\|s\|_S$. Formally: $\|s\|_T \leq c^{\|s\|_S}$ for a constant c.

To see that the number of terms smaller than a given size σ is at most exponential, note that $\#\{t|\ \|t\| \leq \sigma\} = \#\{\|t\|\ |\ \|\gamma(t)\| \leq 2 \cdot \sigma\}$ by Lemma 6. Because γ is injective we have $\#\{\|t\|\ |\ \|\gamma(t)\| \leq 2 \cdot \sigma\} = \#\{P|\ \|P\| \leq 2 \cdot \sigma\}$, which is $\leq \#\{P|\ \|P\| \leq 2 \cdot \sigma\}$. Finally, $\#\{P|\ \|P\| \leq 2 \cdot \sigma\} \leq 5n - 1$ follows by induction on n. The 5 is because there are four different symbols a program can start with, and variables indices use a fifth symbol. ◀

6.2 CBPV Simulating WCBV

In this section, we prove that CBPV can simulate Turing machines with reasonable time and space overhead. We achieve this by using WCBV as an intermediate model. There is existing work showing WCBV can simulate Turing machines with reasonable time and space overhead [11]. Thus what remains to be shown in this section is a reasonable simulation of WCBV using CBPV. With prior work, this suffices to prove Theorem 2.

Let T denote the set of WCBV λ-terms constructed from the following grammar:

$$t, u \in T \quad := \quad \mathsf{var}\, x \mid \mathsf{app}\, T\, T \mid \lambda.\, T$$

WCBV does not allow reduction under λ. Time and space cost semantics are as in [11]:

$$\frac{}{\lambda.\, t \mathrel{\dot{\Downarrow}_0} \lambda.\, t} \qquad \frac{t \mathrel{\dot{\Downarrow}_{k_1}} \lambda.\, t' \qquad u \mathrel{\dot{\Downarrow}_{k_2}} \lambda.\, u' \qquad {t'}^{0}_{\lambda.\, u'} \mathrel{\dot{\Downarrow}_{k_3}} t'}{\mathsf{app}\, t\, u \mathrel{\dot{\Downarrow}_{k_1+k_2+k_3+1}} t''}$$

$$\frac{}{\lambda.\, t \mathrel{\dot{\Downarrow}^{\|\lambda.\, t\|}} \lambda.\, t} \qquad \frac{t \mathrel{\dot{\Downarrow}^{s_1}} \lambda.\, t' \qquad u \mathrel{\dot{\Downarrow}^{s_2}} \lambda.\, u' \qquad {t'}^{0}_{\lambda.\, u'} \mathrel{\dot{\Downarrow}^{s_3}} t''}{\mathsf{app}\, t\, u \mathrel{\dot{\Downarrow}^{\max(s_1+1+\|u\|,\, \|\lambda.\, t'\|+1+s_2,\, s_3)}} t''}$$

The compilation function $c(t)$ is then defined as follows:

$$\begin{array}{lcl} c(\mathsf{var}\, x) & = & \mathsf{var}\, x \qquad\qquad c(\lambda.\, t) \;=\; \mathsf{ret\, thunk}\ \lambda.\, c(t) \\ c(\mathsf{app}\, t\, u) & = & \mathsf{pseq}\ (c(u))\ (c(t))\ (\mathsf{app}\ (\mathsf{force\, var}\, 0)\ (\mathsf{var}\, 1)) \end{array}$$

We prove that CBPV can simulate WCBV with constant overheads, by a routine induction:

▶ **Theorem 22.** *For each closed WCBV term t, we have:*

1. *If* $t \mathrel{\dot{\Downarrow}_k} u$*, then there exists* k' *such that* $c(t) \Downarrow_{k'} c(u)$ *and* $k' \leq 5 * k$*; and*

2. *If* $t \mathrel{\dot{\Downarrow}^s} u$*, then there exists* s' *such that* $c(t) \Downarrow^{s'} c(u)$ *and* $s' \leq 6 * s$*.*

Note that the standard translation uses seq twice instead of pseq once, like this:

$$\begin{array}{lcl} c'_\eta(\mathsf{var}\, x) & = & \mathsf{var}\, \eta(x) \qquad\qquad c'_\eta(\lambda.\, t) \;=\; \mathsf{ret\, thunk}\ \lambda.\, c'_{\eta^{\Uparrow}[0 \mapsto 0]}(t) \\ c'_\eta(\mathsf{app}\, t\, u) & = & \mathsf{seq}\ (c'_\eta(u))\ (\mathsf{seq}\ c'_{\eta^{\uparrow}}(t)\ (\mathsf{app}\ (\mathsf{force\, var}\, 0)\ (\mathsf{var}\, 1))) \end{array}$$

Some de Bruijn arithmetic is needed to account for the extra binder in the app case. *Environments* η are (partial) functions from $\mathbb{N}$ to $\mathbb{N}$. We define lifting operators as follows:

$$\eta^{\uparrow} \equiv x \mapsto \eta(x) + 1 \qquad \eta^{\Uparrow} \equiv x \mapsto \eta(x+1) + 1$$

$$\eta[x \mapsto y](z) \equiv \begin{cases} y & \text{if } x = z \\ \eta(z) & \text{otherwise} \end{cases}$$

However, it turns out that c' has linear space overhead, because there exists terms t such that $\|c'(t)\| = \Omega(\|t\|^2)$, as shown by the following example. Consider a term t_n consisting of n right-associated applications, followed by n occurrences of the variable 0. For example, we'd have the following, where I denotes the identity function $\lambda.\,\mathsf{var}\,0$:

$$\begin{aligned} t_1 &\equiv \mathsf{app}\,\mathsf{I}\,(\mathsf{var}\,0) \\ t_2 &\equiv \mathsf{app}\,\mathsf{I}\,(\mathsf{app}\,\mathsf{I}\,(\mathsf{app}\,(\mathsf{var}\,0)\,(\mathsf{var}\,0))) \\ t_3 &\equiv \mathsf{app}\,\mathsf{I}\,(\mathsf{app}\,\mathsf{I}\,(\mathsf{app}\,\mathsf{I}\,(\mathsf{app}\,(\mathsf{app}\,(\mathsf{var}\,0)\,(\mathsf{var}\,0))\,(\mathsf{var}\,0)))) \ldots \end{aligned}$$

We have $size(t_n) = \mathcal{O}(n)$. But if we consider $c'(t_n)$, the n occurrences of $\mathsf{var}\,0$ in t_n become n occurrences of $\mathsf{var}\,n$, and hence $size(t_n) = \Omega(n^2)$.

This technicality arises because de Bruijn indices count towards term size. We must count like so because numbers are not representable in constant space on Turing machines.

Fortunately, the problematic additional bindings introduced by applications are vacuous over the operand. Translating CBPV to WCBV requires introducing intermediary bindings, and we cannot solve the issue by shuffling arguments: it is necessary for either the operator or operand of an application to be in the scope of a vacuous binding. Based on this observation, we conjecture that without pseq (or products), CBPV is not reasonable for space.

6.2.1 Do we need to go to Turing machines?

Since WCBV is known to be reasonable, the reader may wonder if we must go all the way to Turing machines to prove Theorem 2. Wouldn't going to WCBV be simpler? This turns out to be straightforward for time cost, but unfortunately the natural encoding of CBPV in WCBV has linear space overhead (cf. Section 6.2). Consider this compilation function:

$$\begin{aligned} d(\mathsf{var}\,x) &= \mathsf{var}\,x & d(\mathsf{thunk}\,m) &= \lambda.\,d(m)^{\uparrow} \\ d(\mathsf{force}\,v) &= \mathsf{app}\,(d(v))\,\lambda.\,\mathsf{var}\,0 & d(\mathsf{ret}\,v) &= d(v) \\ d(\lambda.\,m) &= \lambda.\,d(m) & d(\mathsf{app}\,m\,v) &= \mathsf{app}\,(d(m))\,(d(v)) \\ d(\mathsf{seq}\,m\,n) &= \mathsf{app}\,(\lambda.\,d(m))\,(d(n)) & d(\mathsf{let}\,v.\,\mathsf{in}\,m) &= \mathsf{app}\,(\lambda.\,d(m))\,(d(v)) \\ d(\mathsf{pseq}\,m_2\,m_1\,n) &= \mathsf{app}\,(\mathsf{app}\,(\lambda.\,\lambda.\,d(n))\,(d(m_2)))\,(d(m_1)) \end{aligned}$$

We flatten the distinction between values and computations, and to suspend computations we use the one mechanism on offer: λ-abstraction. We have proved that this compilation strategy is reasonable for time cost. Unfortunately, it does not yield a reasonable space cost model. To see why, consider the following variation on the example from the previous section.

$$\begin{aligned} t_1 &\equiv \mathsf{app}\,(\mathsf{force}\,\mathsf{var}\,0)\,(\mathsf{var}\,0) \\ t_2 &\equiv \mathsf{force}\,\mathsf{thunk}\,\mathsf{app}\,(\mathsf{app}\,(\mathsf{force}\,\mathsf{var}\,0)\,(\mathsf{var}\,0))\,(\mathsf{var}\,0) \\ t_3 &\equiv \mathsf{force}\,\mathsf{thunk}\,\mathsf{force}\,\mathsf{thunk}\,\mathsf{app}\,(\mathsf{app}\,(\mathsf{app}\,(\mathsf{force}\,\mathsf{var}\,0)\,(\mathsf{var}\,0))\,(\mathsf{var}\,0))\,(\mathsf{var}\,0) \\ &\ldots \end{aligned}$$

That is, term t_n contains $\mathcal{O}(n)$ mentions of $\mathsf{var}\,0$ under $\mathcal{O}(n)$ layers of thunks. We have $size(t_n) = \mathcal{O}(n)$. But when we consider $d(t_n)$, the n occurrences of $\mathsf{var}\,0$ in t_n become n occurrences of $\mathsf{var}\,n$, and hence $size(d(t_n)) = \Omega(n^2)$.

Clearly a reasonable encoding in this direction is impossible: the detour via Turing machines would yield one. But the choice of encoding function would not be obvious.

7 Conclusion and Future Work

In this paper, we establish the first time and space cost models for the CBPV λ-calculus and formally verify that CBPV relates to intermediate abstract machines. These intermediate machines are interleaved to maintain the desired time and space bounds in relation to Turing machines by extending known results about weak call-by-value [11, 12]. Together, this gives the first proof that CBPV is a reasonable model of computation. Hence, CBPV can serve as a basis for reasoning about the computational complexity of functional programs.

In future work we plan to investigate cost models for extensions to CBPV that support call-by-need evaluation. Moreover, it is unclear how the Bang calculus [9] relates to CBPV in terms of time and space cost, and whether the Bang calculus is reasonable. It would also be interesting to consider sublinear complexity classes. Finally, it would be interesting to extend our work into cost models for λ-calculus variants with different evaluation strategies, thus laying the foundation for a unifying approach of complexity analysis for the λ-calculus.

References

1 Beniamino Accattoli. (in)efficiency and reasonable cost models. In Sandra Alves and Renata Wasserman, editors, *12th Workshop on Logical and Semantic Frameworks, with Applications, LSFA 2017, Brasília, Brazil, September 23-24, 2017*, volume 338 of *Electronic Notes in Theoretical Computer Science*, pages 23–43. Elsevier, 2017. doi:10.1016/j.entcs.2018.10.003.

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

4 Zhuo Chen. cbpv-reasonable-hol, June 2025. URL: https://github.com/ZhuoZoeyChen/cbpv-reasonable-HOL.

5 Zhuo Zoey Chen, Johannes Åman Pohjola, and Christine Rizkallah. cbpv-reasonable-HOL. Software, swhId: swh:1:dir:df18377e9fa5e35255f2687ad66ddbc2f010b934 (visited on 2025-09-11). URL: https://github.com/ZhuoZoeyChen/cbpv-reasonable-HOL/, doi:10.4230/artifacts.24718.

6 Jules Chouquet and Christine Tasson. Taylor expansion for call-by-push-value. In Maribel Fernández and Anca Muscholl, editors, *28th EACSL Annual Conference on Computer Science Logic, CSL 2020, January 13-16, 2020, Barcelona, Spain*, volume 152 of *LIPIcs*, pages 16:1–16:16. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2020. doi:10.4230/LIPICS.CSL.2020.16.

7 Alonzo Church. A set of postulates for the foundation of logic. *Annals of Mathematics*, 33(2):346–366, 1932. URL: http://www.jstor.org/stable/1968337.

8 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.

9 Thomas Ehrhard and Giulio Guerrieri. The bang calculus: an untyped lambda-calculus generalizing call-by-name and call-by-value. In James Cheney and Germán Vidal, editors, *Proceedings of the 18th International Symposium on Principles and Practice of Declarative Programming, Edinburgh, United Kingdom, September 5-7, 2016*, pages 174–187. ACM, 2016. doi:10.1145/2967973.2968608.

10 Thomas Ehrhard and Christine Tasson. Probabilistic call by push value. *CoRR*, abs/1607.04690, 2016. arXiv:1607.04690.

11 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.

12 Yannick Forster, Fabian Kunze, Gert Smolka, and Maximilian Wuttke. A mechanised proof of the time invariance thesis for the weak call-by-value λ-calculus. In Liron Cohen and Cezary Kaliszyk, editors, *12th International Conference on Interactive Theorem Proving, ITP 2021, June 29 to July 1, 2021, Rome, Italy (Virtual Conference)*, volume 193 of *LIPIcs*, pages 19:1–19:20. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPIcs.ITP.2021.19.

13 Yannick Forster, Steven Schäfer, Simon Spies, and Kathrin Stark. Call-by-push-value in coq: Operational, equational, and denotational theory. In Assia Mahboubi and Magnus O. Myreen, editors, *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2019, Cascais, Portugal, January 14-15, 2019*, pages 118–131. ACM, 2019. doi:10.1145/3293880.3294097.

14 Dmitri Garbuzov, William Mansky, Christine Rizkallah, and Steve Zdancewic. Structural operational semantics for control flow graph machines. *CoRR*, abs/1805.05400, 2018. arXiv:1805.05400.

15 G. A. Kavvos, Edward Morehouse, Daniel R. Licata, and Norman Danner. Recurrence extraction for functional programs through call-by-push-value. *Proc. ACM Program. Lang.*, 4(POPL):15:1–15:31, 2020. doi:10.1145/3371083.

16 Delia Kesner and Andrés Viso. The power of tightness for call-by-push-value. *CoRR*, abs/2105.00564, 2021. arXiv:2105.00564.

17 Fabian Kunze, Gert Smolka, and Yannick Forster. Formal small-step verification of a call-by-value lambda calculus machine. In Sukyoung Ryu, editor, *Programming Languages and Systems - 16th Asian Symposium, APLAS 2018, Wellington, New Zealand, December 2-6, 2018, Proceedings*, volume 11275 of *Lecture Notes in Computer Science*, pages 264–283. Springer, 2018. doi:10.1007/978-3-030-02768-1_15.

18 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.

19 Paul Blain Levy. Call-by-push-value: A subsuming paradigm. In Jean-Yves Girard, editor, *Typed Lambda Calculi and Applications, 4th International Conference, TLCA'99, L'Aquila, Italy, April 7-9, 1999, Proceedings*, volume 1581 of *Lecture Notes in Computer Science*, pages 228–242. Springer, 1999. doi:10.1007/3-540-48959-2_17.

20 Paul Blain Levy. *Call-by-Push-Value.* PhD thesis, Queen Mary University of London, UK, 2001. URL: https://ethos.bl.uk/OrderDetails.do?uin=uk.bl.ethos.369233.

21 Paul Blain Levy. Adjunction models for call-by-push-value with stacks. In Richard Blute and Peter Selinger, editors, *Category Theory and Computer Science, CTCS 2002, Ottawa, Canada, August 15-17, 2002*, volume 69 of *Electronic Notes in Theoretical Computer Science*, pages 248–271. Elsevier, 2002. doi:10.1016/S1571-0661(04)80568-1.

22 Dylan McDermott and Alan Mycroft. Extended call-by-push-value: Reasoning about effectful programs and evaluation order. In Luís Caires, editor, *Programming Languages and Systems - 28th European Symposium on Programming, ESOP 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings*, volume 11423 of *Lecture Notes in Computer Science*, pages 235–262. Springer, 2019. doi:10.1007/978-3-030-17184-1_9.

23 Robin Milner, Mads Tofte, and Robert Harper. *Definition of standard ML.* MIT Press, 1990.

24 Christine Rizkallah, Dmitri Garbuzov, and Steve Zdancewic. A formal equational theory for call-by-push-value. In Jeremy Avigad and Assia Mahboubi, editors, *Interactive Theorem Proving - 9th International Conference, ITP 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 9-12, 2018, Proceedings*, volume 10895 of *Lecture Notes in Computer Science*, pages 523–541. Springer, 2018. doi:10.1007/978-3-319-94821-8_31.

25 Konrad Slind and Michael Norrish. A brief overview of HOL4. In Otmane Aït Mohamed, César A. Muñoz, and Sofiène Tahar, editors, *Theorem Proving in Higher Order Logics, 21st International Conference, TPHOLs 2008, Montreal, Canada, August 18-21, 2008. Proceedings*, volume 5170 of *Lecture Notes in Computer Science*, pages 28–32. Springer, 2008. doi:10.1007/978-3-540-71067-7_6.

26 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. In Richard A. DeMillo, editor, *Proceedings of the 16th Annual ACM Symposium on Theory of Computing, April 30 - May 2, 1984, Washington, DC, USA*, pages 391–400. ACM, 1984. doi:10.1145/800057.808705.

27 Cassia Torczon, Emmanuel Suárez Acevedo, Shubh Agrawal, Joey Velez-Ginorio, and Stephanie Weirich. Effects and coeffects in call-by-push-value (extended version). *CoRR*, abs/2311.11795, 2023. doi:10.48550/arXiv.2311.11795.

28 Peter van Emde Boas. Chapter 1 - machine models and simulations. In Jan Van Leeuwen, editor, *Algorithms and Complexity*, Handbook of Theoretical Computer Science, pages 1–66. Elsevier, Amsterdam, 1990. doi:10.1016/B978-0-444-88071-0.50006-0.