A Mechanised Proof of the Time Invariance Thesis for the Weak Call-By-Value λ-Calculus

Yannick Forster
Saarland University, Saarland Informatics Campus, Saarbrücken, Germany

Fabian Kunze
Saarland University, Saarland Informatics Campus, Saarbrücken, Germany

Gert Smolka
Saarland University, Saarland Informatics Campus, Saarbrücken, Germany

Maximilian Wuttke
Saarland University, Saarland Informatics Campus, Saarbrücken, Germany

Abstract

The weak call-by-value λ-calculus L and Turing machines can simulate each other with a polynomial overhead in time. This time invariance thesis for L, where the number of β-reductions of a computation is taken as its time complexity, is the culmination of a 25-years line of research, combining work by Blelloch, Greiner, Dal Lago, Martini, Accattoli, Forster, Kunze, Roth, and Smolka. The present paper presents a mechanised proof of the time invariance thesis for L, constituting the first mechanised equivalence proof between two standard models of computation covering time complexity.

The mechanisation builds on an existing framework for the extraction of Coq functions to L and contributes a novel Hoare logic framework for the verification of Turing machines.

The mechanised proof of the time invariance thesis establishes L as model for future developments of mechanised computational complexity theory regarding time. It can also be seen as a non-trivial but elementary case study of time-complexity-preserving translations between a functional language and a sequential machine model. As a by-product, we obtain a mechanised many-one equivalence proof of the halting problems for L and Turing machines, which we contribute to the Coq Library of Undecidability Proofs.

2012 ACM Subject Classification Theory of computation → Computability; Theory of computation → Type theory

Keywords and phrases formalizations of computational models, computability theory, Coq, time complexity, Turing machines, lambda calculus, Hoare logic

Digital Object Identifier 10.4230/LIPIcs.ITP.2021.19

Supplementary Material *Software (Code Repository)*: `https://github.com/uds-psl/time-invariance-thesis-for-L`; archived at `swh:1:dir:cc13c3c6279985ae15903a24e23bd5d356b6f435`

Acknowledgements We want to thank Lennard Gäher and Dominik Kirst for valuable feedback on drafts of this paper.

1 Introduction

Computability theory – i.e. the study of which problems can be solved computationally – is invariant under the chosen model of computation: Any Turing-complete model does the job. Similarly, but less generally, computational complexity theory – i.e. the study of how efficiently problems can be solved computationally – is invariant under the chosen model: Both Turing machines and RAM machines are used and can be exchanged as long as only complexity classes closed under polynomial time reductions are of interest. This practice is based on the *invariance thesis*, introduced by Slot and van Emde Boas as "all reasonable models of computation simulate each other with a polynomially bounded overhead in time and a constant factor overhead in space" [23]. For the present paper, we dub the first half concerning time complexity the *time invariance thesis*.

While RAM machines and Turing machines are reasonable in this sense, the question how and whether the (untyped) λ-calculus [4] is reasonable was a long-standing open question. This may sound especially surprising given the fact that the λ-calculus was one of the first models for computability to be proven Turing-complete, by Turing himself [27]. On the other hand, the λ-calculus is indeed more complicated than sequential models: Terms are trees and contain binders, computation is non-local, multiple potentially non-equivalent reduction strategies can be chosen, etc. Maybe most crucially, it is not obvious what a reasonable time complexity measure is: The number of β-steps in a computation? Or does one have to account for (the size of) β-redexes? And even more unclear: What is the space complexity of a λ-calculus computation?

More or less independently of these questions, one direction of the time invariance thesis is easy to prove: The λ-calculus can simulate Turing machines.

In this paper, we focus on the weak call-by-value λ-calculus introduced by Plotkin [21], in the concrete variant L introduced by Forster and Smolka [12]. In L, only abstractions are values, and we take the number of β-steps as time complexity measure of a computation [7]. L is similar to an ML-like functional programming language: Reductions in abstractions are not allowed, and a β-reduction only applies when both sides of an application are a value.

A short timeline of time complexity for L and the full λ-calculus looks as follows:

1995 Blelloch and Greiner [3] prove the time invariance thesis for the weak call-by-value λ-calculus w.r.t. RAM machines, with the number of β-steps as time complexity measure.

2008 Dal Lago and Martini [17] prove the time invariance thesis for the weak call-by-value λ-calculus w.r.t. Turing machines, with the sum over all differences of the size of the redex and the size of the reduct as time complexity measure.

2015 Accattoli and Dal Lago [1] prove the time invariance thesis for the full λ-calculus w.r.t. RAM machines, with the number of β-steps as time complexity measure.

2020 Forster, Kunze, and Roth [7] prove the invariance thesis for L w.r.t. Turing machines, with the number of β-steps as time complexity measure and the maximum of the sizes of all intermediate terms as space complexity measure.

All of these results come with different levels of formality in their proofs. Blelloch and Greiner [3] only *state* their result, but do not give any details of a proof, much less provide details how the (RAM)-machine carrying out the simulation might look like. Dal Lago and Martini [17] give a *proof sketch*: They informally describe what a Turing machine simulating L has to do, and for instance how many tapes it has, but do not give invariants for inductions. The other, easier simulation direction is discussed in similar detail. In a later note, Dal Lago and Accattoli [16] give a *formal* proof on paper that the λ-calculus – independent from the concrete variant used – can simulate Turing machines with linear overhead. The proof of the time invariance thesis for the full λ-calculus by Accattoli and Dal Lago [1] is based on a careful and *formal* study of explicit sharing, but again the implementation in terms of machines is left out. This technique is omnipresent in theoretical computer science and mathematics: The folklore parts of results are only sketched or are entirely left out, while the interesting parts are formalised and detailed proofs are given.

When mechanising a result in an interactive theorem prover, both aspects form their own challenges: For folklore results, first a proof has to be found, then formalised, then mechanised, and each individual step can prove challenging. For a formal proof, only missing details of an argument have to be recovered, which can still impose challenges.

The proof of the invariance thesis for L [7] is accompanied by a mechanisation for one part of the novel contribution, namely two stack machine semantics for L. Since it is folklore that concrete algorithms can be implemented on Turing machines, this part is only sketched.

In the present paper, we give a mechanised proof of the time invariance thesis for L in Coq [26], providing all details left out in existing proofs. We mechanise two variants of the time invariance thesis for L. The first provides a Turing machine M_{sim}, simulating L based on a stack machine for L with a heap, and vice versa a term s_{sim} simulating Turing machines:

▶ **Theorem 1** (Time Invariance Thesis for L w.r.t. Simulation). *There are a Turing machine M_{sim}, an L-term s_{sim}, and encoding relations of L-terms and heaps on tapes, and Turing machines and tapes as L-terms s.t.*

1. *M_{sim} simulates L with polynomial overhead, i.e. there is a polynomial p s.t. for all closed s:*
   a. *Whenever s terminates in i steps with value v, M_{sim} run on an input tape encoding s terminates in $p(i, \|s\|)$ steps with an encoding of a heap containing v on its output tape, where $\|s\|$ is the size of s.*
   b. *If M_{sim} terminates on the encoding s, then s terminates.*
2. *s_{sim} simulates TMs with linear overhead, i.e. there is a constant c s.t. for all M:*
   a. *Whenever $M : \mathsf{TM}^n_\Sigma$ terminates on tapes $\boldsymbol{t}$ in i steps with result tapes $\boldsymbol{t'}$, s_{sim} run on an encoding of both M and $\boldsymbol{t}$ evaluates to the encoding of $\boldsymbol{t'}$ in $c \cdot i \cdot \|M\| + c$ steps.*
   b. *If s_{sim} terminates on the encodings M and $\boldsymbol{t}$, then M terminates on $\boldsymbol{t}$.*

As a by-product, this theorem yields the first mechanised proof that the halting problems for L and Turing machines are many-one equivalent, which we contribute as a corner stone to the Coq Library of Undecidability Proofs [11].

Two subtleties are important to point out: First, an L-computation does not have explicit input and output. Any L-term s_0 can compute on its own. Input can be realised by application to an (encoded) value $(s_0\overline{n})$ and the result of the computation can be considered its output. In contrast, a Turing machine M can not compute without the value of its tapes specified, unless one explicitly defines the canonical value of tapes to be e.g. empty. Secondly, the theorem depends on notions of encoding an L-term on a tape, of encoding heaps on a tape, and of encoding a Turing machine and tapes as L-terms. The choice of such encodings is not canonical, and they have to fulfil certain properties for the theorem to be meaningful. For instance, the unfolding of a heap to an L-term on a Turing machine has to be (at most) polynomial in time, otherwise the theorem is meaningless.

As is well-known, computation in L and the λ-calculus in general is potentially subject to so-called "size explosion", i.e. the size of a result of a computation can be exponentially larger than both the size of the input and the number of steps. Thus, in the above theorem, unfolding the heap containing v will be polynomial in the size of v and i, but the size of v might be exponential in both the size of s and i. The following version of the time invariance thesis abstracts away from such subtleties by only considering the computability of k-ary relations on boolean strings, while still being as transparent as possible:

▶ **Theorem 2** (Time Invariance Thesis for L w.r.t Computability). *Let $R \subseteq (\mathbb{LB})^k \times \mathbb{LB}$. Then*

1. *If R is L-computable with time complexity function τ_R, then there is a polynomial p s.t. R is TM-computable with time complexity function $(m, n_1, \ldots, n_k) \mapsto p(m, n_1, \ldots, n_k, \tau_R(n_1, \ldots, n_k))$.*
2. *If R is TM-computable with time complexity function τ_R, then there is a constant c s.t. R is L-computable with time complexity function $(m, n_1, \ldots, n_k) \mapsto c \cdot m \cdot n_1 \cdots n_k \cdot \tau_R(n_1, \ldots, n_k) + c$.*

We accomodate for size explosion by making the time complexity function of a relation R also depend on the size m of the output. Of course, for Turing machines m is always bounded linearly by $n_1, \ldots, n_k$ and i.

Two corollaries are immediate: If the output of a relation R is polynomially bounded by its input, R is polynomially TM-computable if and only if it is polynomially L-computable. In particular, the complexity class P (i.e. PTIME) agrees for both L and Turing machines.

Note that the theorem still depends on encodings, namely on the precise definitions of a relation R being TM- and L-computable. However, the two notions can be assessed independently: L-computability only depends on how to encode boolean strings ($\mathbb{LB}$), and similar for TM-computability. Unsurprisingly, both encodings are very simple, and it is obvious that e.g. equality checking works in linear time.

The theorems are furthermore equivalent, in the sense that one can prove one from the other without additional complex simulations: We obtain our mechanised proof of the time invariance thesis w.r.t. computability from the time invariance thesis w.r.t. simulation, by defining terms and machines dynamically exchanging between the encodings of $\mathbb{LB}$ on Turing machines, $\mathbb{LB}$ in L, L-terms on Turing machines, and Turing machines as L-terms. The other direction would be possible by verifying a universal Turing machine with polynomial overhead in time for Turing machine computation, and similarly a universal L-term.

We avoid non-polynomial overhead for terms exhibiting size explosion by relying on heaps. However, this means that the space overhead is linear-logarithmic rather than constant factor, due to terms exhibiting pointer explosion. A mechanised proof of the time and space invariance thesis for L is left for future work.

The present work is based on the certifying extraction framework for L by Forster and Kunze [6] and the Turing machine verification framework by Forster, Kunze, and Wuttke [9].

The certifying extraction framework allows extracting (by definition total) Coq functions on first-order types to L and automatically proves correctness. The user can provide a time complexity function, which is then automatically verified as well. We use an L-computability proof of Turing machine transitions, which is already a case study of the framework.

The Turing machine verification framework allows giving algorithms in the style of a register-based while-language, and a corresponding machine is automatically constructed behind the scenes. Separate correctness and verification proofs are then inclusion proofs between the automatically derived and the user-given relations for the constructed machine.

Contribution. The main contribution of the paper are mechanised proofs of the time invariance thesis for L w.r.t. both simulation and computability. As a by-product, we obtain that the halting problems for Turing machines and L are many-one equivalent, a result we contribute to the Coq Library of Undecidability Proofs [11]. We also contribute a Hoare logic framework for the verification of correctness, time, and space complexity of Turing machines.

Outline. The paper is split into four parts: First, Section 3 introduces the weak call-by-value λ-calculus L with small-step, big-step, and stack machine semantics, and Section 4 introduces Turing machines and the Turing machine verification framework from [9]. Secondly, we explain the simulation of L computations on Turing machines with polynomial overhead in Section 5, and how to obtain a simulator s_{sim} from the L-computability of Turing machine steps [6] in Section 6, which yields a mechanised proof of the time invariance thesis for L w.r.t. simulation. Thirdly, we explain how to prove that TM-computable relations $R \subseteq (\mathbb{LB})^k \times \mathbb{LB}$ are L-computable with polynomial overhead in Section 7. Lastly, we introduce a novel Hoare logic verification framework for Turing machines in Section 8, which we use to prove that L-computable relations $R \subseteq (\mathbb{LB})^k \times \mathbb{LB}$ are TM-computable with polynomial overhead in Section 9, which yields a mechanised proof of the time invariance thesis for L w.r.t. computability.

Related work. Considering only the simulation of models of computation without time complexity, we are aware of Xu et al. [29] mechanising the equivalence of register machines and Turing machines, Forster and Larchey-Wendling [10] with a mechanised compilation of register machines to binary stack machines, and Larchey-Wendling and Forster [18], proving that the halting problems of register machines and μ-recursive functions, and the solvability of diophantine equations are many-one equivalent. In unpublished work, Pous [22] mechanises an equivalence proof between counter machines and partial recursive functions in Coq.

2 Notations and Definitions

We use the following inductive types:

$$\mathbb{1} ::= () \quad \text{(unit)} \qquad o : \mathbb{O}A ::= \mathsf{None} \mid \mathsf{Some}\, a \quad \text{(options)}$$
$$n : \mathbb{N} ::= 0 \mid \mathsf{S}\, n \quad \text{(natural numbers)} \qquad l : \mathbb{L}A ::= [] \mid a :: l \quad \text{(lists)}$$
$$b : \mathbb{B} ::= \mathsf{true} \mid \mathsf{false} \quad \text{(booleans)} \qquad A + B := \mathsf{inl}\, a \mid \mathsf{inr}\, b \quad \text{(sums)}$$
$$A \times B := (a, b) \quad \text{(pairs)}$$

We use the notation **if** a **is** s **then** b_1 **else** b_2 for inline case analysis, which evaluates to b_1 if a is of the shape s (e.g. for a non-zero number $s := \mathsf{S}n$ or for a list $s := []$ or $s := x :: l$), and to b_2 otherwise.

We use the functions $\mathsf{map} : (A \to B) \to \mathbb{L}A \to \mathbb{L}B$ and $\mathsf{map}_2 : (A \to B \to C) \to \mathbb{L}A \to \mathbb{L}B \to \mathbb{L}C$, defined as follows

$$\mathsf{map}\, f\, (a :: l) := fa :: \mathsf{map}\, f\, l \qquad \mathsf{map}_2\, f\, (a :: l_1)\, (b :: l_2) := fab :: \mathsf{map}_2\, f\, l_1\, l_2$$
$$\mathsf{map}\, f\, [] := [] \qquad \mathsf{map}_2\, f\, l_1\, l_2 := []$$

We use i, n, k, and m as letters for numbers, but try to be consistent in there use: numbers of steps are i, number of tapes are n, the arity of input and relations is k, the size of inputs are $n_1, \ldots, n_k$ and the size of the output is m.

We write $\mathbb{P}$ for the type of propositions. $R \subseteq A \times B$ is short for $R : A \to B \to \mathbb{P}$, and $P \subseteq B$ is short for $P : B \to \mathbb{P}$. In particular, if $R \subseteq A \times B$ and $a : A$, then $Ra \subseteq B$.

A^k is the type of vectors of length k with elements in A. We use bold letters $(\boldsymbol{t}, \boldsymbol{u}, \ldots)$ for vectors and reuse list functions such as map_2 for vectors. The type Fin_n is inductively defined to have exactly n elements. We write the elements of the type Fins_n as $0, \ldots, n$.

A retraction $X \hookrightarrow Y$ consists of a function $I : X \to Y$ and an inverse function $R : Y \to \mathbb{O}X$ s.t. $\forall x.\ R(Ix) = \mathsf{Some}\, x$.

3 The call-by-value λ-calculus L

The call-by-value λ-calculus was introduced by Plotkin [21] as variant of Church's λ-calculus [5]. The concrete variant of the call-by-value λ-calculus we present here is called L [12]. We define syntax and semantics for L in Section 3.1 and introduce a stack machine with a heap in 3.2.

3.1 Syntax, small-step, and big-step semantics

We define the syntax of L using de Bruijn indices as the syntax of the full λ-calculus, i.e. as variables, applications, or abstractions. The size $\|s\|$ counts the number of constructors in s with unary encoded de Bruijn indices.

$$s, t, u : \mathsf{tm}_{\mathsf{L}} ::= n : \mathbb{N} \mid st \mid \lambda s \qquad \|n\| := n + 1 \qquad \|st\| := \|s\| + \|t\| + 1 \qquad \|\lambda s\| := 1 + \|s\|$$

We use names for concrete terms on paper, e.g. write $(\lambda xy.xx)(\lambda z.z)$ for $(\lambda\lambda 11)(\lambda 0)$.

We define a simple substitution operation s_t^n, agreeing with a more standard parallel substitution operation when the term substituted with is closed:

$$n_u^m := \textbf{if } n = m \textbf{ then } u \textbf{ else } n \qquad (st)_u^n := s_u^n t_u^n \qquad (\lambda s)_u^n := \lambda(s_u^{\mathsf{S}n})$$

Formally, we say that a term s is a *closed term* if $\forall nu.\ s_u^n = s$.

We now define a small-step relation $\succ$, its n-step repetition $\succ^n$, and an inductive characterisation of weak, call-by-value big-step evaluation $s \rhd v$:

$$\frac{}{(\lambda s)(\lambda t) \succ s_{\lambda t}^0} \quad \frac{s \succ s'}{st \succ s't} \quad \frac{t \succ t'}{st \succ st'} \quad \frac{}{s \succ^0 s} \quad \frac{s \succ s' \quad s' \succ^n t}{s \succ^{\mathsf{S}i} t} \quad \frac{}{\lambda s \rhd^0 \lambda s} \quad \frac{s \rhd^{i_1} \lambda u \quad t \rhd^{i_2} t' \quad u_{t'}^0 \rhd^{i_3} v}{st \rhd^{1+i_1+i_2+i_3} v}$$

Note that we have for example $(\lambda xy.xx)(\lambda z.z) \rhd^1 \lambda y.(\lambda z.z)(\lambda z.z)$. Evaluation is called weak because the bodies of abstractions are not evaluated and call-by-value because arguments are evaluated before a function is called. Evaluation agrees with evaluation in Plotkin's calculus [21] on closed terms, but does not treat variables as values.

▶ **Lemma 3.** *If s is closed, $s \rhd^i t$ if and only if $s \succ^i t$ and t is an abstraction.*

Weak call-by-value reduction is uniformly confluent [20], meaning for a terminating term s, we can talk about *the* time complexity of s without fixing a reduction path.

▶ **Fact 4.** *If $s \succ t_1$ and $s \succ t_2$, then $t_1 = t_2$ or $\exists u. t_1 \succ u \land t_2 \succ u$.*

▶ **Corollary 5.** *If $s \rhd^{n_1} t_1$ and $s \rhd^{n_2} t_2$, then $n_1 = n_2$ and $t_1 = t_2$.*

The L *halting problem* is defined as $\mathsf{Halt}_{\mathsf{L}}(s : \mathsf{tm}_{\mathsf{L}}, H : \mathsf{closed} s) := \exists t.\ s \rhd t$.

To define L-computability we introduce so-called Scott encodings [14, 19] for $\mathbb{B}$, $\mathbb{N}$, and $\mathbb{L}$, which internalise the case-analysis behaviour of the respective types.

$$\overline{\mathsf{true}} := \lambda xy.x \qquad \overline{0} := \lambda xy.x \qquad \overline{[]} := \lambda xy.x$$
$$\overline{\mathsf{false}} := \lambda xy.y \qquad \overline{\mathsf{S}n} := \lambda xy.y\overline{n} \qquad \overline{b :: l} := \lambda xy.y\ \overline{b}\ \overline{l}$$

A relation $R \subseteq \mathbb{LB}^k \times \mathbb{LB}$ is L-*computable* with time complexity relation $\tau \subseteq \mathbb{N}^{1+k} \times \mathbb{N}$ if

$$\exists s : \mathsf{tm}_{\mathsf{L}}. \forall l_1 \dots l_k. (\forall l.\ R(l_1, \dots, l_k)l \to \exists c \le \tau(|l|, |l_1|, \dots, |l_k|).\ s\,\overline{l_1} \dots \overline{l_k} \rhd^c \overline{l}) \land$$
$$\forall t.\ s\overline{l_1} \dots \overline{l_k} \rhd t \to \exists l.\ R(l_1, \dots, l_k)\overline{l}$$

Note that time complexity is a relation rather than a function. We will only consider functional time complexity relations, and write them on paper as if they were functions. However, for undecidable relations R (e.g. expressing halting problems necessary for deducing a proof of the time invariance thesis w.r.t. simulation from the one w.r.t. computability), it is crucial that τ is a relation: one could use a complexity *function* $\tau : \mathbb{N}^{1+k} \to \mathbb{N}$ to write a Coq decision function checking whether $\exists l.\ R(l_1, \dots, l_k)l$. Since we know that such a decision is impossible in the constructive type theory of Coq, time complexity functions can not be defined for undecidable problems.

3.2 Stack machine semantics

The big-step semantics allows for a compact definition, but is not ideal for implementations of L. To prepare for a simulation of L on Turing machines we introduce a stack machine for L, utilising references to a heap instead of substitution, similar to the heap machine by Kunze et al. [15]. In contrast to the results there, we give a direct correctness proof instead of a step-wise refinement via several machines. Our machine is also similar to the heap machine by Forster et al. [7], but with fewer reduction rules simplifying verification.

Instead of terms, we will work with programs $P, Q : \mathsf{Pro} := \mathbb{L}\,\mathsf{Com}$, which are lists of commands. Commands are reference, application, abstraction, or return tokens:

$$c : \mathsf{Com} ::= \mathsf{ref}\ n \mid \mathsf{app} \mid \mathsf{lam} \mid \mathsf{ret}$$

The *compilation function* $\gamma : \mathsf{Ter} \to \mathsf{Pro}$ compiles terms to programs:

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

We have $\gamma((\lambda xy.xx)(\lambda z.z)) = [\mathsf{lam}; \mathsf{lam}; \mathsf{ref}\ 1; \mathsf{ref}\ 1; \mathsf{app}; \mathsf{ret}; \mathsf{ret}; \mathsf{lam}; \mathsf{ref}\ 0; \mathsf{ret}; \mathsf{app}]$.

Compiled abstractions start with the token lam and end with ret. We can thus define a function $\phi P : \mathbb{O}(\mathsf{Pro} \times \mathsf{Pro})$ that extracts the body of an abstraction by matching the tokens like parentheses. We define $\phi P := \phi_{0,[]}\, P$, where $\phi_{k,Q}\, P$ is an auxiliary function storing the number k of unmatched lam and the processed prefix Q:

$$\phi_{k,Q}\ [] := \mathsf{None} \qquad \phi_{0,Q}\ (\mathsf{ret} :: P) := \mathsf{Some}\,(Q, P) \qquad \phi_{\mathsf{S}k,Q}\ (\mathsf{ret} :: P) := \phi_{k,Q +\!\!+ [\mathsf{ret}]}\, P$$

$$\phi_{k,Q}\ (\mathsf{lam} :: P) := \phi_{\mathsf{S}k,Q +\!\!+ [\mathsf{lam}]}\, P \qquad \phi_{k,Q}\ (c :: P) := \phi_{k,Q +\!\!+ [c]}\, P \ \text{ if } c = \mathsf{ref}\ n \text{ or } \mathsf{app}$$

The *states of the heap machine* are tuples T, V, H. The *control stack* T and the *value stack* V are lists of *closures* $g : \mathsf{Clos} := \mathsf{Pro} \times \mathbb{N}$. A closure (P, a) denotes an open program, where the reference 0 in P has to be looked up at address $a : \mathbb{N}$ in the heap when evaluating.

The *heap* H is a linked list of heap entries $e : \mathsf{Entry} := \mathbb{O}(\mathsf{Clos} \times \mathbb{N})$, i.e. an entry is either empty, or contains the head of the list and the address of its tail. Given a heap H and an address a, $H[a] : \mathbb{O}\,\mathsf{Entry}$ denotes the a-th element of H. We define $H[a, n]$ to be the n-th entry on the heap starting at address a as follows:

$$H[a, n] := \textbf{if } H[a] \textbf{ is } \mathsf{Some}\,(\mathsf{Some}\,(g, b)) \textbf{ then if } n \textbf{ is } \mathsf{S}n \textbf{ then } H[b, n] \textbf{ else } \mathsf{Some}\, g \textbf{ else } \mathsf{None}$$

We can now define the small-step semantics of the stack machine for L:

$$\begin{aligned}
(\mathsf{lam} :: P, a) :: T, V, H &\rightsquigarrow (P', a) ::_{\mathsf{tc}} T, (Q, a) :: V, H \quad \textit{if } \phi P = \mathsf{Some}\,(Q, P') \\
(\mathsf{ref}\ n :: P, a) :: T, V, H &\rightsquigarrow (P, a) ::_{\mathsf{tc}} T, g :: V, H \quad \textit{if } H[a, n] = \mathsf{Some}\, g \\
(\mathsf{app} :: P, a) :: T, g :: (Q, b) :: V, H &\rightsquigarrow (Q, |H|) :: (P, a) ::_{\mathsf{tc}} T, V, H +\!\!+ [\mathsf{Some}\,(g, b)]
\end{aligned}$$

Here, $(P, a) ::_{\mathsf{tc}} T := \textbf{if } P \textbf{ is } [] \textbf{ then } T \textbf{ else } (P, a) :: T$.

In the abstraction rule, the complete abstraction is parsed via ϕ and its body put on the value stack. In principle $::$ instead of $::_{\mathsf{tc}}$ could be used to obtain a correct machine, however the time complexity of this machine is easier to verify using this optimising operation.

Similarly, in the reference rule, the body of the abstraction corresponding to the variable n is looked up in the heap starting at address a and the result is put on the value stack.

In the application rule, the machine takes the closure of the called function (Q, b) and its argument g from the value stack. The address b is bound to g in the heap, the entry being appended to H, thus obtaining address $|H|$. The machine continues evaluating the body Q, where the value for reference 0 can be looked up at address $|H|$, where it was just placed.

Given a closed term s, the initial state of the machine is $([(\gamma s, 0)], [], [])$, i.e. an empty value stack, an empty heap, and the closure $(\gamma s, 0)$ on the task stack. In fact, for a closed term, 0 can be replaced by any address since there are no free references. An example run of the machine for $(\lambda xy.xx)(\lambda z.z)$ can be found in Figure 1, using a as start address.

To state the correctness of the stack machine we need to define an unfolding operation $\mathsf{unf}_H(P, a)$. We will use functional notation for unfolding on paper, but define it as a functional relation in Coq, since the function is not structurally recursive, and not even terminating on cyclic heaps. The function $\mathsf{unf} : \mathsf{Pro} \to \mathsf{tm_L}$ unfolds programs from the value

$$
\begin{aligned}
&[([\mathsf{lam};\mathsf{lam};\mathsf{ref}\ 1;\mathsf{ref}\ 1;\mathsf{app};\mathsf{ret}\ ;\mathsf{ret}\ ;\mathsf{lam};\mathsf{ref}\ 0;\mathsf{ret}\ ;\mathsf{app}],a)],[],[]\\
\rightsquigarrow\ &[([\mathsf{lam};\mathsf{ref}\ 0;\mathsf{ret}\ ;\mathsf{app}],a)],[([\mathsf{lam};\mathsf{ref}\ 1;\mathsf{ref}\ 1;\mathsf{app};\mathsf{ret}\ ],a)],[]\\
\rightsquigarrow\ &[([\mathsf{app}],a)],[([\mathsf{ref}\ 0],a),([\mathsf{lam};\mathsf{ref}\ 1;\mathsf{ref}\ 1;\mathsf{app};\mathsf{ret}\ ],a)],[]\\
\rightsquigarrow\ &[([\mathsf{lam};\mathsf{ref}\ 1;\mathsf{ref}\ 1;\mathsf{app};\mathsf{ret}\ ],0)],[],[\mathsf{Some}\ (([\mathsf{ref}\ 0],a),a)]\\
\rightsquigarrow\ &[],[([\mathsf{ref}\ 1;\mathsf{ref}\ 1;\mathsf{app}],0)],[\mathsf{Some}\ (([\mathsf{ref}\ 0],a),a)]
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{unf}_{[\mathsf{Some}\ (([\mathsf{ref}\ 0],a),a)]}([\mathsf{ref}\ 1;\mathsf{ref}\ 1;\mathsf{app}],0)\\
=\ &\mathsf{unf}_{[\mathsf{Some}\ (([\mathsf{ref}\ 0],a),a)],0,0}(\mathsf{unf}[\mathsf{ref}\ 1;\mathsf{ref}\ 1;\mathsf{app}])\\
=\ &\mathsf{unf}_{[\mathsf{Some}\ (([\mathsf{ref}\ 0],a),a)],0,0}(\lambda 11)\\
=\ &\lambda\mathsf{unf}_{[\mathsf{Some}\ (([\mathsf{ref}\ 0],a),a)],0,1}(11)\\
=\ &\lambda(\mathsf{unf}_{[\mathsf{Some}(([\mathsf{ref}\ 0],a),a)],0,1}1)(\mathsf{unf}_{[\mathsf{Some}(([\mathsf{ref}\ 0],a),a)],0,1}1)\\
=\ &\lambda(\mathsf{unf}_{[\mathsf{Some}\ (([\mathsf{ref}\ 0],a),a)],0,0}\\
&\qquad(\mathsf{unf}[\mathsf{ref}\ 0]))\ (\mathsf{unf}_{[\mathsf{Some}\ (([\mathsf{ref}\ 0],a),a)],0,0}(\mathsf{unf}[\mathsf{ref}\ 0]))\\
=\ &\lambda(\lambda 0)(\lambda 0) = \lambda y.(\lambda z.z)(\lambda z.z)
\end{aligned}
$$

Figure 1 Example of the execution for term $(\lambda xy.\ xx)\ (\lambda z.z)$.

stack into a term by inversing γ. It adds λ to the result, since only the bodies of abstractions are saved on the value stack. The function $\mathsf{unf}_{H,a,k} : \mathsf{tm_L} \to \mathsf{tm_L}$ substitutes free variables $n \geq k$ in a term by $H[a, n-k]$. Finally, $\mathsf{unf}_H : \mathsf{Pro} \times \mathbb{N} \to \mathsf{tm_L}$ unfolds a result using the two previous functions. The example from above is continued in Figure 1.

$$\mathsf{unf}P := \lambda t \quad \textit{(if } \gamma t = P\textit{)} \qquad\qquad \mathsf{unf}_{H,a,k}n := n \quad \textit{(if } n < k\textit{)}$$

$$\mathsf{unf}_{H,a,k}n := \mathsf{unf}_{H,b,0}(\mathsf{unf}P) \quad \textit{(if } n \geq k \textit{ and } H[a, n-k] = \mathsf{Some}\,(P, b)\textit{)}$$

$$\mathsf{unf}_{H,a,k}(st) := (\mathsf{unf}_{H,a,k}s)(\mathsf{unf}_{H,a,k}s) \qquad \mathsf{unf}_{H,a,k}(\lambda s) := \lambda(\mathsf{unf}_{H,a,\mathsf{S}k}s) \qquad \mathsf{unf}_H(P, a) := \mathsf{unf}_{H,a,0}(\mathsf{unf}P)$$

We define the size of the components of a stack machine as follows:

$$\|\mathsf{ref}\ n\| := \|n\| + 1 \qquad \|\mathsf{app}\| := \|\mathsf{lam}\| := \|\mathsf{ret}\,\| := 1 \qquad \|n : \mathbb{N}\| := n + 1$$

$$\|(a, b)\| := \|a\| + \|b\| + 1 \qquad \|[]\| := 1 \qquad \|x :: l\| := \|x\| + \|l\|$$

The final correctness theorem then reads:

▶ **Theorem 6.** *Let s be closed.*

1. *If $s \triangleright^i v$ then $([(\gamma s, 0)], [], []) \rightsquigarrow^{3\cdot i+1} ([], [(P, a)], H)$ for some P, a and H s.t. $\mathsf{unf}_H(P, a) = v$.*
2. *If $([(\gamma s, 0)], [], []) \rightsquigarrow^i (T, V, H)$ then $\|(T, V, H)\| \leq c \cdot (i+1) \cdot (i + \|s\| + 1)$ for some c and if furthermore $\neg\exists\sigma.(T, V, H) \rightsquigarrow \sigma$, then $T = []$, $V = [(P, a)]$, and $s \triangleright v$ for some P, a, and v s.t. $\mathsf{unf}_H(P, a) = v$ is defined.*

3.3 Mechanisation in Coq

The weak call-by-value λ-calculus L is a sweet spot for the mechanisation of computability and complexity theory – but only since it is engineered to be one. In principle, there is an abundance of options which λ-calculus to choose: call-by-value or call-by-name, weak or strong, variables are values or not, de Bruijn encoding with simple substitution, or with parallel substitution, or locally nameless, or parametric higher-order abstract syntax.

For the implementation of L on Turing machines it is mainly the choice of a simple substitution operation based on de Bruijn indices that is crucial. Parallel substitution is considerably more complicated to define, since it is not structurally recursive *and* a priori uses functions, i.e. uncountable types, to represent substitutions. In contrast, simple substitutions are structurally recursive and only require natural numbers for their definition.

To simulate Turing machines in L, it is crucial that also a small step semantics is available: the non-determinism of $\succ$, i.e. that it applies to both sides of an application, allows (directed) equational reasoning in correctness proofs without any overhead.

4 Turing machines

Turing machines [27] are widely used in books on computability theory, are the standard model of computation for complexity theory, and are still considered by many to be the model of computation most convincingly capturing all "effectively calculable" functions. Despite their universal use, there is however no consensus on how to formally define Turing machines in the literature. There are a multitude of definitions which can all be proved equivalent. In Appendix A, we compare the definition we use [9] to the one by Hopcroft et al. [13].[1]

In Section 4.2, we give an overview of the Turing machine verification framework from [9].

4.1 Definition

We start by defining a tape over type Σ inductively using four constructors [2]:

$$\mathsf{tp}_\Sigma ::= \mathsf{niltp} \mid \mathsf{leftof}\ r\ rs \mid \mathsf{midtp}\ ls\ m\ rs \mid \mathsf{rightof}\ l\ ls \qquad \textit{where } m, l, r : \Sigma \textit{ and } ls, rs : \mathbb{L}\Sigma$$

The representation does enforce that a tape contains a continous sequence of symbols, and that either a symbol or one of the ends of the tape is distinguished as head position. There are no explicit blank symbols, which allows for a unique representation of every tape and no well-formedness predicate is needed.

We define a type of moves Move, a function $\mathsf{mv} : \mathsf{Move} \to \mathsf{tp} \to \mathsf{tp}$ applying a move to a tape, a function $\mathsf{wr} : \mathbb{O}\Sigma \to \mathsf{tp} \to \mathsf{tp}$ writing to a tape, and a function $\mathsf{curr} : \mathsf{tp} \to \mathbb{O}\Sigma$ obtaining the current symbol of a tape in Figure 2.

We define multi-tape *Turing machines* $M : \mathsf{TM}^n_\Sigma$, where $n : \mathbb{N}$ is the number of tapes and the finite type Σ is the alphabet, as dependent pairs $(Q, \delta, q_0, \mathsf{halt})$ where Q is a finite type, $\delta : Q \times (\mathbb{O}\Sigma)^n \to Q \times (\mathbb{O}\Sigma \times \mathsf{Move})^n$, $q_0 : Q$ is the starting state, and $\mathsf{halt} : Q \to \mathbb{B}$ indicates halting states. The definition of Turing machine evaluation $M(q,t) \triangleright (q', t')$ and the *Turing machine halting problem* $\mathsf{Halt}_{\mathsf{TM}}$ are defined in Figure 2.

A relation $R \subseteq (\mathbb{L}\mathbb{B})^k \times (\mathbb{L}\mathbb{B})$ is TM-*computable* with time-complexity $\tau \subseteq \mathbb{N}^{1+k} \times \mathbb{N}$ if

$$\begin{aligned}&\exists n : \mathbb{N}.\ \exists \Sigma.\ \exists \mathsf{s}\,\mathsf{bl} : \Sigma.\ \mathsf{s} \neq \mathsf{bl} \wedge \exists M : \mathsf{TM}^{1+k+n}_\Sigma. \forall l_1 \dots l_k.\\ &\quad (\forall l. R\,(l_1, \dots, l_k)\,l \to \exists i \le \tau(|l|, |l_1|, \dots, |l_k|).\ \exists q\,\boldsymbol{t}.\ M(q_0, [\mathsf{niltp}, \overline{l_1}, \dots, \overline{l_k}, \mathsf{niltp}, \dots, \mathsf{niltp}]) \triangleright^i (q, \boldsymbol{t}) \wedge \boldsymbol{t}[0] = \overline{l}) \wedge\\ &\quad \forall q\,\boldsymbol{t}\,i.\ M(q_0, [\mathsf{niltp}, \overline{n_1}, \dots, \overline{n_k}, \mathsf{niltp}, \dots, \mathsf{niltp}]) \triangleright^i (q, \boldsymbol{t}) \to \exists l.\ R\,(l_1, \dots, l_k)\,l\end{aligned}$$

with $\overline{[x_1, \dots, x_n]} := \mathsf{midtp}\ []\ \mathsf{bl}\ [\overline{x_1}, \dots, \overline{x_n}]$ and $\overline{\mathsf{true}} := \mathsf{s}$, $\overline{\mathsf{false}} := \mathsf{bl}$.

4.2 Verified programming of Turing machines

As presented, Turing machines are not compositional: There is no canonical way how to execute a 5-tape machine over alphabet $\mathbb{B}$ after a 3-tape machine over $\mathbb{O}(\mathbb{B} \times \mathbb{B})$.

To allow for the composition of Turing machines and their verification, we first introduce labellings in order to abstract away from the state space. A *labelled Turing machine* over a type L, written $M : \mathsf{TM}^n_\Sigma(L)$, is a dependent pair (M', lab_M) of a machine $M' : \mathsf{TM}^n_\Sigma$ and a labelling function $\mathsf{lab}_M : Q_{M'} \to L$.

To prove the soundness of machines, we introduce *realisation*. A Turing machine $M : \mathsf{TM}^n_\Sigma(L)$ realises a relation $R \subseteq \mathsf{tp}^n_\Sigma \times (L \times \mathsf{tp}^n_\Sigma)$ if

$$M \vDash R := \forall \boldsymbol{t}\ q\ \boldsymbol{t'}.\ M(q_0, \boldsymbol{t}) \triangleright (q, \boldsymbol{t'}) \to R\ \boldsymbol{t}\ (\mathsf{lab}_M\ q, \boldsymbol{t'})$$

[1] chosen for instance by Wikipedia as reference definition

$$\mathsf{Move} ::= \mathsf{L} \mid \mathsf{N} \mid \mathsf{R}$$

$$\begin{aligned}
&\mathsf{mv} : \mathsf{Move} \to \mathsf{tp} \to \mathsf{tp} \\
&\mathsf{mv}\,\mathsf{L}\,(\mathsf{rightof}\,l\,ls) := \mathsf{midtp}\,ls\,l\,[] \qquad \mathsf{mv}\,\mathsf{R}\,(\mathsf{leftof}\,r\,rs) := \mathsf{midtp}\,[]\,r\,rs \\
&\mathsf{mv}\,\mathsf{L}\,(\mathsf{midtp}\,[]\,m\,rs) := \mathsf{leftof}\,m\;rs \qquad \mathsf{mv}\,\mathsf{R}\,(\mathsf{midtp}\,ls\,a\,[]) := \mathsf{rightof}\,a\;ls \\
&\mathsf{mv}\,\mathsf{L}\,(\mathsf{midtp}\,(l :: ls)\,a\,rs) := \mathsf{midtp}\,ls\,l\,(a :: rs) \qquad \mathsf{mv}\,\mathsf{R}\,(\mathsf{midtp}\,ls\,a\,(r :: rs)) := \mathsf{midtp}\,(a :: ls)\,r\,rs \\
&\mathsf{mv}\,m\,t := t \quad \textit{in all other cases}
\end{aligned}$$

$$\begin{aligned}
&\mathsf{wr} : \mathbb{O}\Sigma \to \mathsf{tp} \to \mathsf{tp} \\
&\mathsf{wr}\,\mathsf{None}\,t := t \qquad \mathsf{wr}\,(\mathsf{Some}\,a)\,\mathsf{niltp} := \mathsf{midtp}\,[]\,a\,[] \qquad \mathsf{wr}\,(\mathsf{Some}\,a)\,(\mathsf{midtp}\,ls\,b\,rs) := \mathsf{midtp}\,ls\,a\,rs \\
&\mathsf{wr}\,(\mathsf{Some}\,a)\,(\mathsf{leftof}\,r\,rs) := \mathsf{midtp}\,[]\,a\,(r :: rs) \qquad \mathsf{wr}\,(\mathsf{Some}\,a)\,(\mathsf{rightof}\,l\,ls) := \mathsf{midtp}\,(l :: ls)\,a\,[]
\end{aligned}$$

$$\begin{aligned}
&\mathsf{curr} : \mathsf{tp} \to \mathbb{O}\Sigma \\
&\mathsf{curr}(\mathsf{midtp}\,ls\,a\,rs) := \mathsf{Some}\,a \qquad \mathsf{curr}t := \mathsf{None} \quad \textit{otherwise}
\end{aligned}$$

$$\frac{\mathsf{halt}\,q = \mathsf{true}}{M(q,\boldsymbol{t}) \triangleright^i (q,\boldsymbol{t})} \qquad \frac{\mathsf{halt}\,q = \mathsf{false} \qquad \delta(q, \mathsf{map}\,\mathsf{curr}\,\boldsymbol{t}) = (q', a) \qquad M(q', \mathsf{map}_2(\lambda(c,m)\,t.\mathsf{mv}\,m\,(\mathsf{wr}\,c\,t))\,a\,t) \triangleright^i (q'', \boldsymbol{t'})}{M(q,\boldsymbol{t}) \triangleright^{\mathsf{S}i} (q'', \boldsymbol{t'})}$$

$$\mathsf{Halt}_{\mathsf{TM}^n_\Sigma}(M : \mathsf{TM}^n_\Sigma, \boldsymbol{t} : \mathsf{tp}^n_\Sigma) := \exists i q'\,\boldsymbol{t'}.\; M(q_0,\boldsymbol{t}) \triangleright^i (q', \boldsymbol{t'})$$

$$\mathsf{Halt}_{\mathsf{TM}}(n : \mathbb{N}, \Sigma, M : \mathsf{TM}^n_\Sigma, \boldsymbol{t} : \mathsf{tp}^n_\Sigma) := \mathsf{Halt}_{\mathsf{TM}^n_\Sigma}(M, \boldsymbol{t})$$

Figure 2 Definitions for Turing machines.

Dually, we introduce *termination*. $M : \mathsf{TM}^n_\Sigma(L)$ terminates in $T \subseteq \mathsf{tp}^n_\Sigma \times \mathbb{N}$ if

$$M \downarrow T := \forall \boldsymbol{t}\, i.\; T\;\boldsymbol{t}\;i \to \exists q\,\boldsymbol{t'}.\; M(t) \triangleright^i (q, \boldsymbol{t'}).$$

We call a machine *total* if $\exists c.\; M \downarrow \lambda t i.i \geq c$, i.e. if it halts on any tape in at most c steps.

▶ **Fact 7.** *The introduced predicates are (anti-)monotonic:*

1. *If $M \vDash R'$ and $\forall \boldsymbol{t}\,\ell\,\boldsymbol{t'}.\; R'\;\boldsymbol{t}\;(\ell,\boldsymbol{t'}) \to R\;\boldsymbol{t}\;(\ell,\boldsymbol{t'})$, then $M \vDash R$.*
2. *If $M \downarrow T'$ and $\forall \boldsymbol{t} i.\; T\,\boldsymbol{t}\,i \to T'\,\boldsymbol{t}\,i$, then $M \downarrow T$.*

We will use the following total machines we call *primitive machines*:

$$\mathsf{Read} : \mathsf{TM}^1_\Sigma(\mathbb{O}(\Sigma)) \vDash \lambda\;\boldsymbol{t}\;(\ell,\boldsymbol{t'}).\; \ell = \mathsf{curr}\;\boldsymbol{t}[0] \wedge \boldsymbol{t'} = \boldsymbol{t} \qquad \mathsf{Write}\;s : \mathsf{TM}^1_\Sigma(\mathbb{1}) \vDash \lambda\;\boldsymbol{t}\;\boldsymbol{t'}.\; \boldsymbol{t'}[0] = \mathsf{wr}\;s\;\boldsymbol{t}[0]$$

$$\mathsf{Move}\;d : \mathsf{TM}^1_\Sigma(\mathbb{1}) \vDash \lambda\;\boldsymbol{t}\;\boldsymbol{t'}.\; \boldsymbol{t'}[0] = \mathsf{mv}\;d\;\boldsymbol{t}[0] \qquad \mathsf{Return}\,\ell : \mathsf{TM}^n_\Sigma(L) \vDash \lambda\;\boldsymbol{t}\;(\ell',\boldsymbol{t'}).\; \boldsymbol{t'} = \boldsymbol{t} \wedge \ell' = \ell$$

The last necessary tool now are *combinators* to compose machines. Given $M : \mathsf{TM}^n_\Sigma(L)$ and $M'_\ell : \mathsf{TM}^n_\Sigma(L')$ (for $\ell : L$), we introduce the combinator $\mathsf{Switch}\;M\;M' : \mathsf{TM}^n_\Sigma(L')$, which executes M and, depending on the label ℓ returned by M, executes M'_ℓ.

$$\frac{M \vDash R \qquad \forall(\ell : L).\; M'_\ell \vDash R'_\ell}{\begin{array}{r}\mathsf{Switch}\;M\;M' \vDash \lambda \boldsymbol{t}_0\;(\ell',\boldsymbol{t'}).\; \exists \boldsymbol{t}\;(\ell : L).\\ R\;\boldsymbol{t}_0\;(\ell,\boldsymbol{t}) \wedge R'_\ell\;\boldsymbol{t}\;(\ell',\boldsymbol{t'})\end{array}} \qquad \frac{M \downarrow T \qquad M \vDash R \qquad \forall(\ell : L).M'_\ell \downarrow T'_\ell}{\begin{array}{r}\mathsf{Switch}\;M\;M' \downarrow \lambda\;\boldsymbol{t}\;k.\; \exists\;k_1\;k_2.\; T\;\boldsymbol{t}\;k_1 \wedge 1 + k_1 + k_2 \leq k\\ \wedge\;\forall\;\ell\;\boldsymbol{t'}.\; R\;\boldsymbol{t}\;(\ell,\boldsymbol{t'}) \to T'_\ell\;\boldsymbol{t'}\;k_2\end{array}}$$

Given a machine $M : \mathsf{TM}^n_\Sigma(\mathbb{O}(L))$, we introduce the combinator $\mathsf{While}\;M : \mathsf{TM}^n_\Sigma(L)$, which loops M until the result label is $\mathsf{Some}\,\ell$. The realisation relation for While is defined

inductively, whereas the termination relation is the co-inductively defined accessibility relation (the dashed line indicates coinduction).

$$\frac{M \vDash R}{\mathsf{While}\ M \vDash \mathsf{While}R\ R} \qquad \frac{R\ \boldsymbol{t}\ (\mathsf{Some}\,\ell, \boldsymbol{t'})}{\mathsf{While}R\ R\ \boldsymbol{t}\ (\ell, \boldsymbol{t'})} \qquad \frac{R\ \boldsymbol{t}\ (\mathsf{None}, \boldsymbol{t'}) \qquad \mathsf{While}R\ R\ \boldsymbol{t'}\ (\ell, \boldsymbol{t''})}{\mathsf{While}R\ R\ \boldsymbol{t}\ (\ell, \boldsymbol{t''})}$$

$$\frac{M \downarrow T \qquad M \vDash R}{\mathsf{While}\ M \downarrow \mathsf{While}T\ R\ T} \qquad \frac{\begin{array}{c} T\ \boldsymbol{t}\ k_1 \quad \forall \boldsymbol{t'}\ l.\ R\ \boldsymbol{t}\ (\mathsf{Some}\,l, \boldsymbol{t'}) \to k_1 \le k \\ \forall \boldsymbol{t'}.\ R\ \boldsymbol{t}\ (\mathsf{None}, \boldsymbol{t'}) \to \exists k_2.\ \mathsf{While}T\ R\ T\ \boldsymbol{t'}\ k_2 \wedge 1 + k_1 + k_2 \le k \end{array}}{\mathsf{While}T\ R\ T\ \boldsymbol{t}\ k}$$

Composing machines with the operator Switch only works for machines over the same alphabet and the same number of tapes. To remedy this situation we introduce lifting operations for alphabets and tapes, and also a relabelling operation Relabel.

Given a retraction $f : \Sigma \to \Gamma$, a default symbol $d : \Sigma$, and a machine $M : \mathsf{TM}^n_\Sigma$, the *alphabet lift* $\Uparrow_{(f,d)} M : \mathsf{TM}^n_\Gamma$ translates every read symbol via f^{-1}, passes it to M, and translates the symbol M writes via f. In case f^{-1} returns None, d is passed to M.

Given a retraction $I : \mathsf{Fin}_m \to \mathsf{Fin}_n$ and a machine $M : \mathsf{TM}^m_\Sigma$, the *tape lift* $\Uparrow_I M : \mathsf{TM}^n_\Sigma$ replicates the behavior of M on tape i on tape Ii, and leaves all other tapes untouched.

We only show the canonical realisation and termination relations for the tape lift here:

$$\Uparrow_I R := \lambda t\ (\ell, t').\ R\ (\mathsf{select}\ I\ \boldsymbol{t})\ (\ell, \mathsf{select}\ I\ \boldsymbol{t'}) \wedge \forall (i : \mathsf{Fin}_n).\ i \notin I \to \boldsymbol{t'}[i] = t[i]$$

$$\Uparrow_I T := \lambda t\ k.\ T\ (\mathsf{select}\ I\ \boldsymbol{t})\ k \qquad \textit{where}\ \mathsf{select}\ I\ \boldsymbol{t} := \mathsf{map}\ (\lambda i.\ t[I(i)])\ [0, \ldots, m-1]$$

$$\frac{M : \mathsf{TM}^m_\Sigma(L) \vDash R \qquad I : \mathsf{Fin}_m \hookrightarrow \mathsf{Fin}_n}{\Uparrow_I M : \mathsf{TM}^n_\Sigma \vDash \Uparrow_I R} \qquad \frac{M \downarrow T \qquad I : \mathsf{Fin}_m \hookrightarrow \mathsf{Fin}_n}{\Uparrow_I M \downarrow \Uparrow_I T}$$

Given a function $r : L_1 \to L_2$ and a machine $M : \mathsf{TM}^n_\Sigma(L_1)$, $\mathsf{Relabel}\ M\ r : \mathsf{TM}^n_\Sigma(L_2)$ behaves like M, but returns label $r\ell$ where M returned ℓ.

The last important layer of abstraction introduces the treatment of tapes as registers, based on the notion $t[i] \simeq v$ expressing that tape $t[i]$ contains an encoded value $v : V$. A type V is a TM-*encodable type* on alphabet Σ if there is a (designated) injective function $\varepsilon : V \to \mathbb{L}\Sigma$. We define such designated encoding functions for several data types, e.g. booleans, tuples, and lists of encodable types. In Coq, the implementation of encodable types relies on type classes, such that users can define their own encoding functions.

We then define *tape containment* $t \simeq_f v$, where V is TM-encodable on alphabet Σ, $v : V$, $t : \mathsf{tp}_{\Gamma^+}$, and $f : \Sigma \to \Gamma$.

$$\Gamma^+ ::= \mathsf{START} \mid \mathsf{STOP} \mid \mathsf{UNKNOWN} \mid (s : \Gamma)$$

$$t \simeq_f v := \exists ls.t = \mathsf{midtp}\ ls\ \mathsf{START}\ (\mathsf{map}\ f\ (\varepsilon v) +\!\!+ [\mathsf{STOP}])$$

Note that the position of the head is fixed in the definition of tape containment: The head must be located on the start symbol. By extending the alphabet Σ with the delimiting symbols START and END, values can effectively be copied from one tape to another tape. The symbol UNKNOWN is used as the canonical default symbol for the alphabet lift. Let $M : \mathsf{TM}^n_{\Sigma^+}$ and $f : \Sigma \to \Gamma$. Then $\Uparrow_{f^+} M : \mathsf{TM}^n_{\Gamma^+}$ (with the canonically inferrable injection $f^+ : \Sigma^+ \to \Gamma^+$) is an alphabet lift of M. In case the lifted machine reads a symbol that is not in the image of f, $\Uparrow_{f^+} M$ behaves like if M reads UNKNOWN. However, this will by design not happen if the head is under a symbol of the encoding of a value.

Void tapes (written $\mathsf{isVoid}\ t$) do not contain values. The head of the tape is located at the right-most symbol:

$$\mathsf{isVoid}\ t := \exists m\ ls.\ t = \mathsf{midtp}\ ls\ m\ []$$

A void tape can be initialised with a value by writing the encoding of a value delimited by START and END.

5 Simulating L on Turing machines

To simulate L on Turing machines, we use the stack machine semantics, meaning we have to implement the relation $\rightsquigarrow$ from Section 3.2 as multi-tape Turing machine $\mathsf{Step} : \mathsf{TM}^n_\Sigma$. The central components of Step are machines implementing the heap lookup operation $H[a, n]$ and the parsing operation ϕ. We will omit concrete implementations, but show the correctness and termination relations and briefly discuss the proof goals for such verifications. The machines will share an alphabet Σ consisting of 30 symbols, allowing to encode commands, programs, addresses, closures, task and value stack, heap entries, and heap.

The relations we display here are simplified in comparison to the actual relations in Coq w.r.t. two aspects: First, we omit the retractions $f_X : \Sigma_X \to \Sigma$ when writing $\simeq$. Since as long as concrete f_X are fixed for every type X encodable on type Σ_X, their concrete definitions do not matter. Secondly, we omit the condition $\mathsf{isVoid}\, t$ both in the premise and conclusion of rules: Any unspecified tape is always implicitly void.

▶ **Fact 8.** *There is a machine* $\mathsf{Lookup} : \mathsf{TM}^5_\Sigma(\mathbb{B})$ *and a* $c : \mathbb{N}$ *s.t.*

1. $\mathsf{Lookup} \vDash \lambda \boldsymbol{t}(\ell, \boldsymbol{t'}).\ \forall H\, a\, n.\ \boldsymbol{t}[0] \simeq H \to \boldsymbol{t}[1] \simeq a \to \boldsymbol{t}[2] \simeq n \to$
   $\textbf{if } \ell \textbf{ then } \exists g.\ H[a, b] = \mathsf{Some}\, g \wedge \boldsymbol{t'}[0] \simeq H \wedge \boldsymbol{t'}[3] \simeq g \textbf{ else } H[a, b] = \mathsf{None}$
2. $\mathsf{Lookup} \downarrow \lambda \boldsymbol{t} i.\ \exists H\, a\, n.\ \boldsymbol{t}[0] \simeq H \wedge \boldsymbol{t}[1] \simeq a \wedge \boldsymbol{t}[2] \simeq n \wedge i \geq c \cdot (n + 1) \cdot (\|H\| + \max \|H\| \|a\|)$

Proof. The machine Lookup can be defined by using building blocks like While and Switch. Once the machine is defined, an inductive relation R s.t. $\mathsf{Lookup} \vDash R$ can be automatically inferred from the relations of the building blocks.

By Fact 7 it then suffices to prove $R\, t\, (\ell, t') \to \forall H\, a\, n.\ \boldsymbol{t}[0] \simeq H \to \boldsymbol{t}[1] \simeq a \to \boldsymbol{t}[2] \simeq n \to$ $\textbf{if } \ell \textbf{ then } \exists g.\ H[a, b] = \mathsf{Some}\, g \wedge \boldsymbol{t'}[0] \simeq H \wedge \boldsymbol{t'}[3] \simeq g \textbf{ else } H[a, b] = \mathsf{None}$ by induction on R.

The termination proof is dual. ◀

▶ **Fact 9.** *There is a machine* $\mathsf{Parse} : \mathsf{TM}^5_\Sigma(\mathbb{B})$ *and a* $c : \mathbb{N}$ *s.t.*

1. $\mathsf{Parse} \vDash \lambda \boldsymbol{t}\, (\ell, t').\forall P.\boldsymbol{t}[0] \simeq P \to$
   $\textbf{if } \ell \textbf{ then } \exists Q\, P'.\ \phi P = (Q, P') \wedge \boldsymbol{t'}[0] \simeq P' \wedge \boldsymbol{t'}[1] \simeq Q \textbf{ else } \phi P = \mathsf{None}$
2. $\mathsf{Parse} \downarrow \lambda \boldsymbol{t} i.\ \exists P.\ \boldsymbol{t}[0] \simeq P \wedge i \geq \|c\| \cdot P^2$

▶ **Fact 10.** *There is a machine* $\mathsf{Step} : \mathsf{TM}^{11}_\Sigma(\mathbb{B})$ *and a* $c : \mathbb{N}$ *s.t.*

1. $\mathsf{Step} \vDash \lambda \boldsymbol{t}\, (\ell, t').\ \forall T\, V\, H.\ \boldsymbol{t}[0] \simeq T \to \boldsymbol{t}[1] \simeq V \to \boldsymbol{t}[2] \simeq H \to$
   $\textbf{if } \ell \textbf{ then } \exists T'\, V'\, H'.\ (T, V, H) \rightsquigarrow (T', V', H') \wedge \boldsymbol{t'}[0] \simeq T' \wedge \boldsymbol{t'}[1] \simeq V' \wedge \boldsymbol{t'}[2] \simeq H'$
   $\textbf{else } (\neg \exists \sigma.\ (T, V, H) \rightsquigarrow \sigma) \wedge T = [] \to \boldsymbol{t'}[0] \simeq [] \wedge \boldsymbol{t'}[1] \simeq V \wedge \boldsymbol{t'}[2] \simeq H$
2. $\mathsf{Step} \vDash \lambda \boldsymbol{t}\, i.\ \boldsymbol{t}[0] \simeq T \wedge \boldsymbol{t}[1] \simeq V \wedge \boldsymbol{t}[2] \simeq H \wedge i \geq 1 + c \cdot \textbf{if } T \textbf{ is } (a, P) :: \_ \textbf{ then}$
   $\|a\| + \|H\| + \|V\| + \|P\| \cdot (1 + \|H\| + \max\, a\, \|H\|_{\mathsf{addr}} + \|P\|) \textbf{ else } 0$

This suffices to prove one direction of the time invariance thesis w.r.t. simulation:

▶ **Theorem 11.** *There is* $M_{\mathsf{sim}} : \mathsf{TM}^{11}_\Sigma$ *and a polynomial* $p : \mathbb{N} \to \mathbb{N}$ *s.t. for closed terms* s

1. *If* $s \rhd^i v$*, then there exist* $\boldsymbol{t}$*,* H*,* P*, and* a *s.t.* $M_{\mathsf{sim}}(\overline{[(\gamma s, 0)]}, \mathsf{niltp}, \ldots, \mathsf{niltp}) \rhd^{p(i, \|s\|)} \boldsymbol{t}$ *with* $\boldsymbol{t}[1] = \overline{[]}$*,* $\boldsymbol{t}[2] = \overline{(P, a)}$*,* $\boldsymbol{t}[3] = \overline{H}$*, and* $\mathsf{unf}_H(P, a) = v$*.*
2. *If* $M_{\mathsf{sim}}(\overline{[(\gamma s, 0)]}, \mathsf{niltp}, \ldots, \mathsf{niltp})$ *terminates, so does* s*.*

Proof. Define $M_{\mathsf{sim}} := \mathsf{While}\ \mathsf{Step}$ and $p\, i\, m := c \cdot (3i + 2)^2 \cdot (3i + 1 + m) \cdot m$ for some c. ◀

▶ **Corollary 12.** $\mathsf{Halt}_{\mathsf{L}} \preceq \mathsf{Halt}_{\mathsf{TM}}$

A first version of the Coq verification of M_{sim} was also discussed in Wuttke's bachelor's thesis [28].

6 Simulating Turing machines in L

To simulate Turing machines in L, we first give an alternative, executable semantics for Turing machines based on iteration of a step-function. We then recap the central ingredients of the certifying extraction framework [6], which we use to extract the step-function to L. And lastly, we implement a (potentially non-terminating) iteration combinator in L.

We define a function $\mathsf{nxt}_M : Q_M \times \mathsf{tp}_\Sigma^n \to Q_M \times \mathsf{tp}_\Sigma^n + \mathsf{tp}_\Sigma^n$ and a polymorphic function $\mathsf{loop} : (X \to X + Y) \to X \to \mathbb{N} \to \mathbb{O}Y$ as follows:

$$\mathsf{nxt}_M(q, \boldsymbol{t}) := \textbf{if } \mathsf{halt} q \textbf{ then } \boldsymbol{t} \textbf{ else let } (q', \boldsymbol{a}) := \delta_M(q, \mathsf{curr}\,\boldsymbol{t}) \textbf{ in } (q', \mathsf{map}_2(\lambda(c, m)t.\mathsf{mv}\, m\, (\mathsf{wr}\, c\, t))\, \boldsymbol{a}\, \boldsymbol{t})$$

$$\mathsf{loop}\, f\, x\, 0 := \mathsf{None} \qquad \mathsf{loop}\, f\, x\, (\mathsf{S}n) := \mathsf{loop}\, f\, x'\, n \;\; (\text{if } fx = \mathsf{inl}\, x')$$

$$\mathsf{loop}\, f\, x\, (\mathsf{S}n) := \mathsf{Some}\, y \;\; (\text{if } fx = \mathsf{inr}\, y)$$

▶ **Fact 13.** $\mathsf{loop}\, \mathsf{nxt}_M\, (q_0, \boldsymbol{t})\, (\mathsf{S}i) = \mathsf{Some}\, \boldsymbol{t'} \leftrightarrow \exists q'.\ M(q, \boldsymbol{t}) \rhd^i (q', \boldsymbol{t'})$

The certifying extraction framework [6] automatically extracts L-terms s_f for functions f and proves that s_f computes f. Additionally, one can pass a time complexity function τ_f and is then presented with proving certain recurrence equations for τ_f. Since we do not implement higher-order functions, we can give a simplified account of the framework here.

Central in the framework are Scott encodings, which are used to encode elements of arbitrary, first-order types as L-terms. The idea behind Scott encodings is that case analysis is by application: For instance, **if** b **then** a_1 **else** a_2 corresponds to the L-term $\overline{b}\ s_{a_1}\ s_{a_2}$, where s_{a_1} and s_{a_2} compute a_1 and a_2 respectively.

In general, for types $A_1, \dots, A_n, B$ with a Scott encoding, a function $f : A_1 \to \dots A_n \to B$, is computed by a term s_f with time complexity function $\tau_f : A_1 \to \dots \to A_n \to \mathbb{N}$ if

$$\forall a_1, \dots, a_n.\ \exists k \le \tau_f\ a_1 \dots a_n.\ s_f\ \overline{a_1} \dots \overline{a_n} \rhd^k \overline{f\, a_1 \dots a_n}$$

The framework also supports higher-order functions and currying, but we omit those features complicating e.g. the definition of time complexity since we do not rely on them.

The certifying extraction framework comes with a library of computability proofs including time complexity, covering natural numbers, list, and vectors. Furthermore, one can give a general computability proof for functions with listable domain. I.e. if there is $[x_1, \dots, x_n] : \mathbb{L}X$ s.t. $\forall x : X.\, x \in [x_1, \dots, x_n]$ and $f : X \to Y$, then there is s_f and a constant c s.t. $\forall x. \exists i \le c \cdot n. s_f\ \overline{x} \rhd^i \overline{fx}$.

Since δ_M has a listable domain, we can use the certifying extraction framework to extract nxt for every M:

▶ **Fact 14.** *Let* $M : \mathsf{TM}_\Sigma^n$. *There is* $\mathsf{nxt}_M : \mathsf{tm_L}$ *and* $C_{\mathsf{nxt}} : \mathbb{N}$ *s.t.* $s_{\mathsf{nxt}_M}\, \overline{(q, \boldsymbol{t})} \rhd^{C_{\mathsf{nxt}}} \overline{\mathsf{nxt}(q, \boldsymbol{t})}$.

Proof. The framework generates the proof obligations $52 \cdot |Q_M|^2 + 56 \le C_{\mathsf{nxt}}$, and $52 \cdot |Q_M|^2 + 130 \cdot n + 216 \le C_{\mathsf{nxt}}$, where $|Q_M|$ is the number of states of M. If C is picked large enough before running extraction, the obligations can be discharged automatically by the tactic `solverec` provided by the framework. ◀

We now define a term s_{loop} which expects f and x and loops f until a value y is found, or indefinitely if not. Since the extraction framework only covers total functions, we have to manually implement s_{loop}. To do so, we rely on a recursion combinator ρ [12], also employed by the framework to use recursion:

▶ **Fact 15.** *If s and s' are closed abstractions, $\rho s s' \succ^3 s(\rho s)s'$.*

We then define s_{loop} with time complexity function $\tau_{\mathsf{loop}} : (X \to \mathbb{N}) \to X \to \mathbb{N} \to \mathbb{N}$:

$$s_{\mathsf{loop}} := \rho(\lambda r f x.\, f x (\lambda x' z. r f x')(\lambda y z. y)(\lambda z.\, z))$$
$$\tau_{\mathsf{loop}}\, \tau_f\, x\, 0 := 0 \qquad \tau_{\mathsf{loop}}\, \tau_f\, x\, (\mathsf{S} i) := \tau_f x + 11 + \mathbf{if}\ f x\ \mathbf{is}\ \mathsf{inl}\, x'\ \mathbf{then}\ \tau_{\mathsf{loop}}\, \tau_f\, x\, i\ \mathbf{else}\ 0$$

▶ **Lemma 16.** *Let f be computable by s_f with time complexity function τ_f, i.e. $\forall x. \exists i \leq \tau_f x.\ s_f \overline{x} \rhd^i \overline{fx}$. Then*
1. *If $\mathsf{loop}\, f\, x\, i = \mathsf{Some}\, y$, then $\exists k \leq \tau_{\mathsf{loop}}\, \tau_f\, x\, i.\ s_{\mathsf{loop}}\, s_f\, \overline{x}\, \overline{i} \rhd^k \overline{y}$.*
2. *If $s_{\mathsf{loop}}\, s_f\, \overline{x}$ terminates, there exist i and y s.t. $\mathsf{loop}\, f\, x\, i = \mathsf{Some}\, y$.*

This suffices to prove one direction of the time invariance thesis w.r.t. simulation:

▶ **Theorem 17.** *There is $s_{\mathsf{sim}} : \mathsf{tm}_{\mathsf{L}}$ s.t. for all $M : \mathsf{TM}^n_\Sigma$ there is $C : \mathbb{N}$ s.t. for all $\boldsymbol{t} : \mathsf{tp}^n_\Sigma$*
1. *If $M(q_0, \boldsymbol{t}) \rhd^i (q, \boldsymbol{t'})$, then $\exists j \leq C \cdot i + C.\ s_{\mathsf{sim}}\ s_{\mathsf{nxt}_M}\ \overline{\boldsymbol{t}} \rhd^j \overline{\boldsymbol{t'}}$.*
2. *If $s_{\mathsf{sim}}\ s_{\mathsf{nxt}_M}\ \overline{\boldsymbol{t}} \rhd v$, then $\exists q \boldsymbol{t'}.\ M(q_0, \boldsymbol{t}) \rhd (q, \boldsymbol{t'})$.*

7 TM-computable relations over $\mathbb{LB}$ are L-computable

Let $R \subseteq (\mathbb{LB})^k \times \mathbb{LB}$ be computable by $M : \mathsf{TM}^n_\Sigma$. We define an L-term computing R by taking $l_1, \ldots, l_k$ as input, converting them to their respective TM-encoding, and then running M with the help of s_{sim}. Step-by-step, s has to:
1. Expect input in the form $s\, \overline{l_1} \ldots \overline{l_k}$,
2. for every $1 \leq i \leq k$ compute $\overline{\mathsf{midtp}\ []\ \mathsf{bl}\, l_i}$, i.e. the L-encoding of the TM-encoding of l_i.
3. run the simulation $s_{\mathsf{sim}}\ s_{\mathsf{nxt}_M}\ \overline{[\mathsf{niltp}, t_1, \ldots, t_k, \mathsf{niltp}, \ldots, \mathsf{niltp}]}$.
4. this computation will (if it terminates) terminate with a value $\overline{(\mathsf{midtp}\ []\ b\, l, t'_2 \ldots, t'_n)}$,
5. meaning s has to output $\overline{l}$.

Three challenges arise: the term s has to be defined parametric in k, the L-encoding of the lists $l_1, \ldots, l_k$ has to be converted to the L-encoding $\overline{[\mathsf{niltp}, t_1, \ldots, t_k, \mathsf{niltp}, \ldots, \mathsf{niltp}]}$, and the L encoding of a result $\boldsymbol{t'}$ has to be analysed, and the TM-encoding of a list l contained $\boldsymbol{t'}[0]$ has to be converted to the L-encoding of l.

For the first task, we implement k-ary substitutions and combinators.

▶ **Fact 18.** *One can define functions $s^n_{\boldsymbol{u}} : \mathsf{tm}_L$ where $s : \mathsf{tm}_{\mathsf{L}}, n : \mathbb{N}, \boldsymbol{u} : \mathsf{tm}^k_{\mathsf{L}}$ and*

$$\lambda_k : \mathsf{tm}_{\mathsf{L}} \to \mathsf{tm}_{\mathsf{L}} \qquad \mathsf{app}_k : \mathsf{tm}_{\mathsf{L}} \to \mathsf{tm}^k_{\mathsf{L}} \to \mathsf{tm}_{\mathsf{L}} \qquad \mathsf{vars}_k : \mathsf{tm}^k_L$$

such that the following hold:
1. $\mathsf{vars}_{\mathsf{S}k} = k :: \mathsf{vars}_k$,
2. $(\mathsf{app}_k s(s_1, \ldots, s_k))^n_{\boldsymbol{u}} = \mathsf{app}_k(s^n_{\boldsymbol{u}})((t_1)^n_{\boldsymbol{u}}, \ldots, (t_k)^n_{\boldsymbol{u}})$,
3. *if all elements of $\boldsymbol{u}$ are closed abstractions, then $\mathsf{app}_k(\lambda_k s)\boldsymbol{u} \succ^k s^0_{\boldsymbol{u}}$.*

The second and third tasks can again be done by extraction.

▶ **Fact 19.** *There is a closed abstraction s_{prep} s.t. $s_{\mathsf{prep}}\, \overline{(l_1, \ldots, l_k)} \rhd \overline{[\mathsf{niltp}, t_1, \ldots, t_k, \mathsf{niltp}, \ldots, \mathsf{niltp}]}$, where $t_i := \mathsf{midtp}\ []\ \mathsf{bl}\, [\overline{x_1}, \ldots, \overline{x_n}]$ for $l_i = [x_1, \ldots, x_n]$.*

▶ **Fact 20.** *There is a closed abstraction $s_{\mathsf{unenc}_{\mathsf{TM}}}$ s.t. if $\boldsymbol{t}[0] = \mathsf{midtp}\ []\ \mathsf{bl}\, [\overline{x_1}, \ldots, \overline{x_m}]$ and $l = [x_1, \ldots, x_n]$ we have $s_{\mathsf{unenc}_{\mathsf{TM}}} \overline{\boldsymbol{t}} \rhd \overline{\mathsf{Some}\, l}$ and $s_{\mathsf{unenc}_{\mathsf{TM}}} \overline{\boldsymbol{t}} \rhd \overline{\mathsf{None}}$ otherwise.*

▶ **Theorem 21.** *If $R \subseteq (\mathbb{LB})^k \times \mathbb{LB}$ is TM-computable by a machine M with time complexity relation τ there are a term s_M and a constant c s.t. s_M computes R with time complexity relation $(m, n_1, \ldots, n_k) \mapsto c \cdot m \cdot n_1 \cdot \cdots \cdot n_k \cdot \tau(m, n_1, \ldots, n_k) + c$.*

Proof. Define $s_M := \lambda_k . s_{\mathsf{unenc}_{\mathsf{TM}}}(s_{\mathsf{sim}}\ s_{\mathsf{nxt}_M}\ (s_{\mathsf{prep}}(s_{\mathsf{cons}} k (\ldots (s_{\mathsf{cons}}\, 0\, \overline{[]}))))\, (\lambda x.x)\, \overline{[]}$. ◀

8 Hoare logic verification framework for Turing machines

Although the relational verification approach is quite powerful, it suffers from two crucial problems. First, realisation and termination are inherently separated proof goals, even for total machines, which form the majority of machines we are interested in. The premises of the correctness and termination relations are usually almost the same, thus manipulation of premises is duplicated. Secondly, in an interactive proof the proof context can grow very large. When two machines with say 9 tapes are sequentially composed, the proof context contains 9 assumptions on the initial tapes, 9 assumptions for the intermediate tapes between the two machines, and 9 final propositions on the tapes. Many of these assumptions are equalities. The Turing machine verification framework has naive tactics to simplify such proof goals by substitution and rewriting, which becomes quadratically slower the longer a realisation proofs takes.

We propose a verification framework based on Hoare logic, solving both problems. Correctness and termination can be proved at once, eliminating the need for repeated manipulation of premises. Furthermore, at each step in the verification of an n tape TM, the user sees exactly n specifications S for the n tapes, plus optionally a custom invariant depending on both the tapes and the label.

We here give a high-level overview. More details are in the separate Appendix B [8].

The Hoare logic is built as a new layer of abstraction for the relational framework. Given $M : \mathsf{TM}^n_\Sigma(L)$, a predicate $P \subseteq \mathsf{tp}^n_\Sigma$ and a relation $Q \subseteq L \times \mathsf{tp}^n_\Sigma$, we write *weak* Hoare triples:

$$\vDash \{P\}\ M\ \{Q\} := M \vDash (\lambda \boldsymbol{t}\ (\ell, t').\ P\ \boldsymbol{t} \to Q\ (\ell, t'))$$

To state that the machine is functionally correct and terminates in a certain time, we use *total* Hoare triples. In addition to the relation of the weak Hoare triple, a total Hoare triple asserts that the machine terminates in (at most) i steps if the precondition is satisfied. Thus, we avoid spelling out the precondition again.

$$\vDash \{P\}^i\ M\ \{Q\} := (\vDash \{P\}\ M\ \{Q\})\ \wedge\ M \downarrow (\lambda \boldsymbol{t}\ i'.\ P\ \boldsymbol{t}\ \wedge\ i \le i')$$

Similar to the old verification framework, every building block like While and Switch comes with an associated Hoare triple, shown in Figure 1 in the separate Appendix B [8]. Using these rules, an interactive verification is akin to a symbolic execution of machines with explicitly annotated invariants.

The Hoare triples of user-defined machines $M : \mathsf{TM}^n_\Sigma$ exclusively have triples with both pre and post conditions of the form $S_1 \wedge \ldots S_n \wedge I$ where S_i for $1 \le i \le n$ are either $\boldsymbol{t}[i] \simeq x$ for some x, $\mathsf{isVoid}\, \boldsymbol{t}[i]$, or $\boldsymbol{t}[i] = t_0$ for some fixed t_0, and I is a custom, user-chosen invariant. Thus, the following specification for a binary machine

$$\begin{aligned} &M \vDash \lambda \boldsymbol{t}\ (\ell, \boldsymbol{t'}).\ \forall (x : X)\ (y : Y).\ P_X x \to P_Y y \to \\ &\qquad \boldsymbol{t}[0] \simeq x \to \boldsymbol{t}[1] \simeq y \to \boldsymbol{t'}[0] \simeq f(x,y) \wedge \boldsymbol{t'}[1] \simeq g(x,y) \\ &M \downarrow \lambda \boldsymbol{t}\, i.\ \exists (x : X)\ (y : Y).\ P_X x \wedge P_Y y \wedge \boldsymbol{t}[0] \simeq x \wedge \boldsymbol{t}[1] \simeq y \wedge i \ge \tau x y \end{aligned}$$

can be compactly restated using the following triple:

$$\forall xy.\ P_X x \to P_Y x \to \vDash \{\lambda \boldsymbol{t}.\ \boldsymbol{t}[0] \simeq x \wedge \boldsymbol{t}[1] \simeq y\}^{\tau x y}\ M\ \{\lambda(\ell, \boldsymbol{t'}).\ \boldsymbol{t'}[0] \simeq f(x,y) \wedge \boldsymbol{t'}[1] \simeq g(x,y)\}$$

A typical workflow for the verification of total machines then looks as follows: First, the user defines Hoare triples for their machines, following the shape $S_1 \wedge \cdots \wedge S_n \wedge I$. Secondly, they perform the correctness proof. Thirdly, they define the time complexity function of the machine, and add the running time to the triple in the proved lemma. Fourthly, they replay the proof script and in the very last step of the verification, show that the accumulated running time is indeed bounded by the time complexity function.

9 L-computable relations over $\mathbb{LB}$ are TM-computable

To convert a term s computing a relation R, we follow the same strategy as in Section 7. We exemplarily show the specification of the machine M_{conv} corresponding to s_{unencTM}:

▶ **Fact 22.** *There is a machine* $M_{\mathsf{unencL}} : \mathsf{TM}^4_\Sigma$ *and a constant* c *s.t. for all* $l : \mathbb{LB}$ *we have* $\vDash \{P_l\}^{c\cdot|l|} M_{\mathsf{unencL}} \{Q_l\}$ *where* $P_l := \lambda \boldsymbol{t}.\ \boldsymbol{t}[0] \simeq \bar{l} \wedge \boldsymbol{t}[1] = \mathsf{niltp} \wedge \mathsf{isVoid}\,\boldsymbol{t}[2] \wedge \mathsf{isVoid}\,\boldsymbol{t}[3]$ *and* $Q_l := \lambda(\_, t').\ \boldsymbol{t}[1] = \mathsf{midtp}\,[]\,\mathsf{bl}\,\bar{l} \wedge \mathsf{isVoid}\,\boldsymbol{t}[2] \wedge \mathsf{isVoid}\,\boldsymbol{t}[3]$.

▶ **Theorem 23.** *If* $R \subseteq (\mathbb{LB})^k \times \mathbb{LB}$ *is* L*-computable by a closed term* s *with time complexity relation* τ *there is a polynomial* p *and a Turing machine* M_s *s.t.* M_s *computes* R *with time complexity relation* $(m, n_1, \ldots, n_k) \mapsto p(m, n_1, \ldots, n_k, \tau_R(m, n_1, \ldots, n_k))$.

Proof. The machine M_s is defined parametric in s and by recursion on k, i.e. we define a different machine for each k. First, M_s reads its input and writes the task stack $[(\gamma(s\,\overline{n}_1\,\overline{n}_k), 0)]$ to an auxiliary tape. As a total subroutine, we verify this part using the Hoare framework. Then M_s runs M_{sim}. If the computation terminates, the resulting heap will contain a term $\bar{l}$ for a list $l : \mathbb{LB}$. Thus, M_s runs a machine implementing unf and lastly runs M_{unencL}.

The size of the heap after i steps is in $\mathcal{O}(i \cdot (i + N))$, where $N := n_1 + \cdots + n_k$. For one reduction step, N variables might have to be looked up, resulting in a runtime of $\mathcal{O}(i \cdot (i + N + 1) \cdot (N + 1))$ per step. Unfolding a heap takes $\mathcal{O}((m + 1) \cdot (H + N + 1))$, where H is the size of the heap. In total, we can thus define the polynomial p cubic in the number of steps i, quadratic in N and linear in the size of the output m as follows, where c is a constant:

$$(m, n_1, \ldots, n_k, i) \mapsto c \cdot (i+1) \cdot (i + n_1 + \cdots + n_k + 1) \cdot ((n_1 + \cdots + n_k + 1) \cdot (i+1) + m). \quad ◀$$

10 Discussion

We have presented the first mechanised proof of an instance of the time invariance thesis, connecting L with Turing machines with a polynomial overhead in time. We prove two variants, respectively concerned with simulation of one model of computation on the other, and with the computability of relations on boolean strings. In total, including dependencies, our development consists of 30.000 LoC and takes about 18 minutes to compile on an Intel Core i7-6600U CPU @ 2.60GHz machine. The novel code for this paper still constitutes 7800 LoC, with about 40% specification and 60% proofs.

It is folklore that the two variants are equivalent. For instance, in [1, 7, 23] more emphasis is put on the simulation variant. In the interactive theorem proving community it is however well-known that "folklore" is not equivalent to "easy to mechanise". In the setting of computability and complexity theory, where one has to deal with models of computation, this fact is amplified: proofs on paper virtually never provide concrete implementations in a model of computation, but focus on the high-level invariants of a proof. This is based on the (again folklore) fact that algorithms can be implemented in models of computation provided enough time and strength to sustain the tedium to do so. However, the proof engineering time needed to show the correctness of an algorithm (as e.g. a Coq function) and to show the correctness of its implementation (as e.g. a Turing machine) are completely independent. Correctness proofs of algorithms depend on the intricacy of invariants. Correctness proofs of implementations depend on the length of the code of a Turing machine, and the size of the gap between specification language and Turing machines. If the specification is a first-order, tail-recursive Coq function, the length of the Turing machine is the main factor.

We however hope that future mechanised proofs of results in complexity theory and of the time invariance thesis for other models of computation can profit from our development.

For us, simulating Turing machines on L and simulating L on Turing machines was the only possible way. Now that L is shown reasonable for time complexity, future proofs of the time invariance thesis can choose which reasonable model to use for each direction of the proof. For the time invariance thesis for a model of computability C, one can show that C can simulate Turing machines, which are structurally simple and where computation is local. In the other direction, one can show that C can be simulated in L, which has rich structure and supports non-local computation on almost arbitrary (first-order) data-structures.

For our concrete cases of Turing machines and L, powerful abstraction layers and verification frameworks are necessary for mechanisation. We would assess that a "manual" approach to any invariance thesis involving Turing machines is completely unfeasible. The Turing machine verification framework [9] and the certifying extraction framework [6] proved very valuable in this regard. While having similar goals, they use different mathematical approaches to both verification and time complexity analysis.

First, the certifying L-extraction framework gives support to automatically prove the computability of a large subset of Coq functions. For time complexity, a user has to give a time complexity function as input, and the framework automatically generates equations the function has to fulfil and furthermore provides tactics to solve these equations. Finding a time complexity function can prove challenging, but an interactive approach where a wrong function is picked and a correct function is reverse-engineered from the equations works well. A priori, the framework does not support partial terms. The `Lsimpl` tactic used in the framework however can be used to normalise terms in manual verification.

Secondly, the Turing machine verification framework provides tools to verify the correctness, time and even space complexity of Turing machines, but the user has to implement those machines manually. For the implementation, the user can write Turing machines in the style of a register based while-language, for which a canonical realisation and termination relation are inferred automatically. Correctness and time complexity proofs w.r.t. user-defined relations are now simply inclusion proofs between the canonical and the user-defined relations. Due to the split of correctness and termination, the framework works well for the verification of partial machines. However, for total machines, termination and correctness are distinct proof goals and have to be proved separately, which leads to a mathematical duplication of similar proof goals, sometimes even to actual proof code duplication.

The novel Hoare logic verification framework we present remedies this situation: For total machines, only one proof subsuming both correctness and time complexity has to be carried out, while it still supports separate proof goals for partial machines. No canonical relations are used. The verification of Hoare triples is carried out directly on the implementation of the machine, using the proof rules for the used combinators and user-defined machines. We conjecture that the Hoare logic framework scales further and possibly far. Our simulation of L on Turing machines is reasonable on time, but if terms exhibit pointer explosion [7], the space overhead might not be constant factor. The proof of this stronger time and space invariance thesis for L [7] is considerably more complicated, but might now be in reach.

However, the assessment that "Turing machines as model of computation are inherently infeasible for the formalisation of any computability or complexity theoretic result" [9] still stands. Future developments of mechanised results in complexity theory can and should be based on L or similarly well-suited models. Our certifying extraction framework provides valuable help in doing so, but there are interesting extensions worth exploring in the future.

First, this framework does not cover space complexity, which would however be needed for a proof of the full invariance thesis for L. Secondly, exploring an automatic generation of time-complexity functions, where the user can then provide a (polynomial) upper bound

for the function in a second step might be interesting. Lastly, the framework requires user input for every single recursive function used, also for auxiliary functions, and the automatic verification of these is based on `Ltac` tactics which sometimes fail, and sometimes are slow. Here, an automatic generation of correctness proofs based on a meta-programming tool for Coq like MetaCoq [24, 25] might be a possible solution.

References

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

2 Andrea Asperti and Wilmer Ricciotti. Formalizing Turing Machines. In *Logic, Language, Information and Computation*, pages 1–25. Springer, 2012.

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

4 Alonzo Church. A set of postulates for the foundation of logic. *Annals of mathematics*, pages 346–366, 1932.

5 Alonzo Church. An unsolvable problem of elementary number theory. *American journal of mathematics*, 58(2):345–363, 1936.

6 Yannick Forster and Fabian Kunze. A Certifying Extraction with Time Bounds from Coq to Call-By-Value Lambda Calculus. In John Harrison, John O'Leary, and Andrew Tolmach, editors, *10th International Conference on Interactive Theorem Proving (ITP 2019)*, volume 141 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 17:1–17:19, Dagstuhl, Germany, 2019. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. doi:10.4230/LIPIcs.ITP.2019.17.

7 Yannick Forster, Fabian Kunze, and Marc Roth. The weak call-by-value λ-calculus is reasonable for both time and space. *Proceedings of the ACM on Programming Languages*, 4(POPL):1–23, 2019.

8 Yannick Forster, Fabian Kunze, Gert Smolka, and Maximilian Wuttke. Appendix b: A mechanised proof of the time invariance thesis for the weak call-by-value λ-calculus. Technical report, Saarland University, 2021. URL: https://www.ps.uni-saarland.de/Publications/documents/AppendixB_ForsterEtAl_2021_Mechanised-Time-Invariance-L.pdf.

9 Yannick Forster, Fabian Kunze, and Maximilian Wuttke. Verified programming of Turing machines in Coq. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*, pages 114–128, 2020.

10 Yannick Forster and Dominique Larchey-Wendling. Certified undecidability of intuitionistic linear logic via binary stack machines and minsky machines. In *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs*, pages 104–117, 2019.

11 Yannick Forster, Dominique Larchey-Wendling, Andrej Dudenhefner, Edith Heiter, Dominik Kirst, Fabian Kunze, Gert Smolka, Simon Spies, Dominik Wehr, and Maximilian Wuttke. A Coq library of undecidable problems. In *The Sixth International Workshop on Coq for Programming Languages (CoqPL 2020).*, 2020. URL: https://github.com/uds-psl/coq-library-undecidability.

12 Yannick Forster and Gert Smolka. Weak call-by-value lambda calculus as a model of computation in Coq. In *International Conference on Interactive Theorem Proving*, pages 189–206. Springer, 2017.

13 John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation (3rd Edition)*. Addison-Wesley Longman Publishing Co., Inc., USA, 2006.

14 Jan Martin Jansen. *Programming in the λ-Calculus: From Church to Scott and Back*, pages 168–180. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013. doi:10.1007/978-3-642-40355-2_12.

15 Fabian Kunze, Gert Smolka, and Yannick Forster. Formal small-step verification of a call-by-value lambda calculus machine. In *Asian Symposium on Programming Languages and Systems*, pages 264–283. Springer, 2018.

16 Ugo Dal Lago and Beniamino Accattoli. Encoding Turing machines into the deterministic lambda-calculus. *arXiv preprint arXiv:1711.10078*, 2017.

17 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.

18 Dominique Larchey-Wendling and Yannick Forster. Hilbert's tenth problem in Coq. *arXiv preprint arXiv:2003.04604*, 2020.

19 Torben Æ. Mogensen. Efficient self-interpretation in lambda calculus. *JOURNAL OF FUNCTIONAL PROGRAMMING*, 2:345–364, 1994.

20 Joachim Niehren. Uniform confluence in concurrent computation. *Journal on Functional Programming*, 10(5):453–499, 2000.

21 Gordon D. Plotkin. Call-by-name, call-by-value and the λ-calculus. *Theoretical computer science*, 1(2):125–159, 1975.

22 Damien Pous. A certified compiler from recursive functions to minski machines. Technical report, Université Paris-Diderot, PPS, 2004. URL: http://perso.ens-lyon.fr/damien.pous/recursive-minski/.

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

24 Matthieu Sozeau, Abhishek Anand, Simon Boulier, Cyril Cohen, Yannick Forster, Fabian Kunze, Gregory Malecha, Nicolas Tabareau, and Théo Winterhalter. The MetaCoq Project. *Journal of Automated Reasoning*, 2020. doi:10.1007/s10817-019-09540-0.

25 Matthieu Sozeau, Simon Boulier, Yannick Forster, Nicolas Tabareau, and Théo Winterhalter. Coq Coq correct! verification of type checking and erasure for Coq, in Coq. *Proceedings of the ACM on Programming Languages*, 4(POPL):1–28, 2019.

26 The Coq Development Team. The coq proof assistant, version 8.12.0, July 2020. doi:10.5281/zenodo.4021912.

27 Alan Mathison Turing. On computable numbers, with an application to the Entscheidungsproblem. *J. of Math*, 58(345-363):5, 1936.

28 Maximilian Wuttke. Verified programming of Turing machines in Coq. Bachelor's thesis, Saarland University, 2018.

29 Jian Xu, Xingyuan Zhang, and Christian Urban. Mechanising Turing machines and computability theory in Isabelle/HOL. In *International Conference on Interactive Theorem Proving*, pages 147–162. Springer, 2013.

A Definition of Turing machines

We compare the definition of Turing machines we use [9] to the one by Hopcroft, Motwani, and Ullman [13].

1. The logical system in [13] is classical set theory, whereas we work in constructive type theory. We discuss the impact of this foundational choice below.
2. The alphabet in [13] is separated into a set of input symbols Σ and a superset of tape symbols Γ, where we unify both into a single type.

3. The blank symbol in [13] is an explicit part of Turing machines as an element of Γ, but not of Σ. We do not specify blank symbols explicitly and instead leave it to a user to specify a blank symbol or even various blank symbols.
4. Tapes in [13] are not formally defined. It is only stated that tapes 'extend infinitely to the left and right, initially hold[ing] [...] the input'. Instantaneous descriptions of Turing machines are formally defined as strings over Γ and Q, which contain 'the portion of the tape between the leftmost and the rightmost non-blank, unless the head is to the left of the leftmost non-blank or to the right of the rightmost non-blank.' In the latter case, the blanks between the head and the non-blank content are part of the string.
5. The transition function in [13] is a partial function, whereas ours is total. If the transition function is unspecified, the computation of the machine halts, whereas we have an explicit boolean halting function. In Coq's type theory one requires classical logic and $\mathsf{AUC}_{\mathbb{N},\mathbb{N}}$ to compile Turing machines with a partial transition function into an equivalent definition with total transition functions. In general, any compilation of partial functions on finite types to total functions is non-computable and thus not definable in Coq's type theory without axioms.
6. A machine in [13] always writes a symbol and always moves to the left or right. We allow to not write a symbol and to not move the head. The first is important regarding our definition of tapes (otherwise a fully empty niltp can never stay fully empty), whereas the second is a relatively arbitrary choice to allow more freedom in the definition of concrete machines.
7. Turing machines have an explicit set of accepting states in [13]. We do not add one, because our definition is not aimed at formalising computability theory directly. Instead, our more flexible definition of labels subsumes the binary notion of accepting states, but allows for more interesting constructions like the Switch and $\mathsf{MemWhile}$ machines.

Subtle difficulties might arise when defining notions of computability theory in classical set theory. For instance, defining Turing machines with a transition function which takes as input the whole tape is problematic: Nothing permits non-computable transition functions then, making arbitrary problems decidable by encoding the decision into the transition function. When imposing the transition function to be computable one obtains a circular dependency: The notion of computability is needed to define transition functions, transition functions are needed to define Turing machines, but Turing machines are needed to define the notion of computability. The well-known solution here is to define Turing machines as finite objects, i.e. let the transition function, although partial, work on a finite domain and codomain. In classical set theory one can then always show afterwards that this transition function is computable, since every function with finite domain and codomain is. In our type-theoretic setting, we can also show that the transition function is computable, provided it is, as we defined it, a total function. We did so in Section 6.