Verified Programming of Turing Machines in Coq

Yannick Forster
Saarland University
Saarbrücken, Germany
forster@ps.uni-saarland.de

Fabian Kunze
Saarland University
Saarbrücken, Germany
kunze@ps.uni-saarland.de

Maximilian Wuttke
Saarland University
Saarbrücken, Germany
s8mawutt@stud.uni-saarland.de

Abstract

We present a framework for the verified programming of multi-tape Turing machines in Coq. Improving on prior work by Asperti and Ricciotti in Matita, we implement multiple layers of abstraction. The highest layer allows a user to implement nontrivial algorithms as Turing machines and verify their correctness, as well as time and space complexity compositionally. The user can do so without ever mentioning states, symbols on tapes or transition functions: They write programs in an imperative language with registers containing values of encodable data types, and our framework constructs corresponding Turing machines.

As case studies, we verify a translation from multi-tape to single-tape machines as well as a universal Turing machine, both with polynomial time overhead and constant factor space overhead.

CCS Concepts • **Theory of computation → Turing machines**; **Type theory**.

Keywords Turing machines, verification, universal machine, Coq

ACM Reference Format:
Yannick Forster, Fabian Kunze, and Maximilian Wuttke. 2020. Verified Programming of Turing Machines in Coq. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP '20), January 20–21, 2020, New Orleans, LA, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3372885.3373816

1 Introduction

Turing machines are, at least on paper, the foundation of modern computability and complexity theory, in part due to the conceptual simplicity of their definition. However, this simplicity leads to a lack of structure, which is also one of their biggest disadvantages: When it comes to detailed or formal reasoning, Turing machines soon become very hard to treat. This is maybe best reflected by the fact that while many basic areas of computer science, like logic, grammars, automata, or programming languages theory have been formalised in proof assistants, formalisations of even basic complexity-theoretic results[1] are not available. While constructing Turing machines on paper might be possible, verifying a non-trivial machine defined in terms of states and transition functions in a proof assistant seems entirely infeasible.

There were several attempts of formalising Turing machines in proof assistants. Asperti and Ricciotti [2015] and Xu, Zhang, and Urban [2013] verify universal Turing machines in Matita and Isabelle/HOL, respectively, and Ciaffaglione [2016] formalises the undecidability of Turing machine halting in Coq. However, none of these results analyse time or space complexity of their machines.

The main difficulty for detailed reasoning about Turing machines is their lack of compositionality. For example, it is not clear at all how to compose a two-tape Turing machine with a three-tape Turing machine that works on a different alphabet. Therefore, it is common to rely on pseudo code or prose describing the intended behaviour. The exact implementation as well as its correctness or resource analysis is left as an exercise to the reader. In a mechanised proof, those details cannot be left out. Luckily, it is possible to hide those details behind suitable abstractions.

We present a framework that aims to have the cake and eat it too when it comes to mechanising computation in terms of Turing machines: Algorithms are stated in the style of a register based while-language; a corresponding Turing machine is automatically constructed behind the scene. Our framework furthermore characterises the semantics by deriving two relations for each machine, one witnessing partial correctness, which can subsume a space-consumption analysis, and one witnessing termination, which can subsume a running time analysis. These relations are similar to relations a Hoare-like logic would derive for the algorithm, especially in that they follow the internal structure of the program. The only task left for the user is to simplify those synthesised relations into a more high-level, hand-written description of the semantics.

CPP '20, January 20–21, 2020, New Orleans, LA, USA

ACM ISBN 978-1-4503-7097-4/20/01…$15.00
https://doi.org/10.1145/3372885.3373816

[1] For example the Cook–Levin theorem (SAT is NP-complete), the inclusions $P \subseteq NP \subseteq PSPACE \subseteq EXP$, or the time and space hierarchy theorems.

Our imperative abstractions for Turing machines are shallowly embedded into Coq's type theory: Primitive operations are predefined Turing machines performing primitive tasks. Control-flow operators like If or While are Coq-functions constructing new Turing machines from existing ones.

To make reasoning and programming of Turing machines feasible, we introduce three layers of abstractions, L_1 - L_3. The lower layers are heavily inspired by the definitions Asperti and Ricciotti [2015] use to formalise multi-tape Turing machines in the proof assistant Matita.

On the lowest layer L_0 (which is actually not an abstraction), we define n-tape Turing machines $M : \mathsf{TM}^n_\Sigma$ over finite alphabets Σ and their semantics in Coq, based on the definitions by Asperti and Ricciotti.

Layer L_1 introduces labelled machines $M : \mathsf{TM}^n_\Sigma(L)$, which additionally contain an arbitrary finite type L together with a function labelling every state of the machine with an element of this type. Labels can be seen as a partitioning of states, abstracting away from implementation details. Based on this notion we define two verification primitives: *realisation* (partial correctness) and *termination*. A labelled machine $M : \mathsf{TM}^n_\Sigma(L)$ realises a relation $R \subseteq \mathsf{Tape}^n_\Sigma \times (L \times \mathsf{Tape}^n_\Sigma)$, written $M \vDash R$, if for every terminating computation, the input tapes are in relation with the label of the terminating state and the output tapes. Dually, a machine *terminates* in a *running time relation* $T \subseteq \mathsf{Tape}^n_\Sigma \times \mathbb{N}$, written $M \downarrow T$, if for every related pair of input tapes t and step counts k, the machine actually terminates in k steps on input t. Realisation and termination can be used to express the correctness of a machine.

Programming machines on layer L_1 is quite hard because we have to define concrete transition functions. We thus introduce layer L_2 with primitive machines that can read, write and move the head as well as lifting operators and control-flow operators.

Lifting operators and control-flow operators can be used to compose machines. For example, given machines $M_1 : \mathsf{TM}^n_\Sigma(L)$ and $M_2 : \mathsf{TM}^m_\Gamma(L')$, we can use tape-lifting operators to obtain machines $M_1' : \mathsf{TM}^{n+m}_\Sigma(L)$ and $M_2' : \mathsf{TM}^{n+m}_\Gamma(L')$, and alphabet-liftings to obtain machines $M_1'' : \mathsf{TM}^{n+m}_{\Sigma+\Gamma}(L)$ and $M_2'' : \mathsf{TM}^{n+m}_{\Sigma+\Gamma}(L')$. We can then use control-flow operators to compose them into a machine $M_1''; M_2'' : \mathsf{TM}^{n+m}_{\Sigma+\Gamma}(L')$. From layer L_2 onward, we never need to consider machine states or transition functions again; we just compose machines and use the labelling function as an abstraction in the verification process.

On layer L_2, programming Turing machines is still very low-level, because we have to consider bare tapes. On layer L_3 — which can still access tools from the lower layers — we introduce the notion that a tape *contains* a value of an arbitrary encodable type, like natural numbers or lists over other encodable types. A tape may also be *void*, i.e. contain no value. We have primitive operators, for example to increase the value of a tape that contains a number, or to copy the value from one tape to another tape.

As case studies, we give a mechanisation of a translation from multi-tape to single-tape machines as well as a mechanisation of a universal Turing machine in Coq. Both machines have a polynomial time and constant factor space overhead. To the best of our knowledge, this is the first formalised universal machine verified w.r.t. time and space complexity for any model of computation in any proof assistant.

Definitions, lemmas, and theorems in the PDF-version of this document are hyperlinked with the accompanying Coq development[2]; those links are marked by a ❣-symbol.

Structure Section 2 introduces notation and type-theoretic preliminaries. Sections 3 to 6 give an overview over the layers L_0 - L_3. We report on the implementation of a translation from multi-tape to single-tape machines and a universal machine in Sections 7 and 8. We conclude by commenting on the mechanised proofs (Section 9), an overview of related work (Section 10), and a discussion of our results and future work (Section 11). The appendix [Forster et al. 2019] contains a full definition of Turing machines with all details.

2 Preliminaries

We work in constructive type theory with inductive types and an impredicative universe of propositions.

The basic inductive types we use are the Booleans $\mathbb{B} ::= \mathsf{true} \mid \mathsf{false}$, the unit type $\mathbb{1} ::= ()$, the natural numbers $\mathbb{N} ::= 0 \mid \mathsf{S}\, n$ for $n : \mathbb{N}$, product types $X \times Y$, sum types $X + Y$, and the type with exactly n elements $\mathbb{F}_n$. Given a type X, we further define options $O(X) ::= \emptyset \mid \lfloor x \rfloor$ and lists $\mathcal{L}(X) ::= [] \mid x :: A$ for $x : X$ and $A : \mathcal{L}(X)$. These notations are shared with vectors $\vec{x} : X^n$ of fixed length $n : \mathbb{N}$. We index elements of vectors $\vec{x} : X^n$ by indices $i : \mathbb{F}_n$ writing $\vec{x}[i]$. We write $f \mathbin{@} A$ to map a function over a list, vector or tape. To inline case distinctions over inductive values inside formulas, we write $\mathsf{match}\ s\ \mathsf{with}\ p_1 \Rightarrow r_1 | \ldots$ with patterns p_1 and results r_i for pattern matching.

We encode relations as functions returning a proposition. Thus $\lambda(nm : \mathbb{N}).\ n = m$ defines the identity relation on natural numbers. We write $R \subseteq A \times B$ as an abbreviation for $R : A \to B \to \mathbb{P}$. $R_1 \circ R_2$ denotes relational composition, i.e. the relation $\lambda ac.\exists b.\ R_1\ a\ b \wedge R_2\ b\ c$. We oftentimes identify $\mathbb{1} \times B$ with B and will thus identify relations $R \subseteq A \times (\mathbb{1} \times B)$ with relations $R \subseteq A \times B$.

We use a constructive notion of injections, namely retractions: A function $f : X \to Y$ is called a retraction, written $f : X \hookrightarrow Y$, if there is a function $f^{-1} : Y \to O(X)$ s.t. $f^{-1}(fx) = \lfloor x \rfloor$.

We say that a type X is discrete if there is a function $X \to X \to \mathbb{B}$ deciding equality. We say that a type X is finite if it is discrete and there is an exhaustive list of elements

[2] https://github.com/uds-psl/tm-verification-framework/tree/master/theories/TM

$[x_0, \ldots, x_n]$. Note that for finite types there is a canonical numbering of elements of x.

3 Multi-tape Turing Machines (L_0)

Our approach to make reasoning about Turing machines feasible is to focus on *what* can be computed instead of *how* exactly it is computed in terms of states and transition functions: We give a few basic relations that are computable and ways to derive new computable relations by combining existing ones. We formalise this by a computability predicate $M \vDash R$, stating that the machine M computes the relation R. Our framework contains combinators like the sequential composition $M_1; M_2$, and a proof that the relation computed by the sequential composition is the composition of the two relations computed by each M_i. This frees the user from the burden to verify Turing machines that are constructed in terms of state sets and transition functions.

The relations themselves relate the input to the result after executing a machine, i.e. a vector of tapes to a vector of tapes.

We represent tapes following Asperti and Ricciotti [2012]:

Definition 3.1. *Let Σ be a type. We define*

$$\mathsf{Tape}_\Sigma ::= \mathsf{niltape} \mid \mathsf{leftof}\ r\ rs \mid \mathsf{midtape}\ ls\ m\ rs \mid \mathsf{rightof}\ l\ ls$$

with $l, m, r : \Sigma$ and $rs, ls : \mathcal{L}(\Sigma)$.

We leave out Σ if the alphabet is clear from the context. We write Tape_Σ^n for a vector of n tapes. The representation does not specify an explicit blank-symbol. The constructors incorporate the position of the head and lead to a unique representation for any tape, not relying on blank symbols: midtape represents that the head currently is on some symbol m with remaining content further left and right in ls and rs, leftof (rightof) denotes that the head is on the left (right) end of the tape with remaining, nonempty content $r :: rs$, and niltape denotes an empty tape. The lists rs, ls are ordered such that symbols closer to the head come first. If blank symbols are needed, they have to be specified explicitly.

The definition of tapes and multi-tape Turing machines is taken from Asperti and Ricciotti [2015]:

Definition 3.2. *An n-tape Turing machine $M : \mathsf{TM}_\Sigma^n$ over a finite alphabet Σ is a tuple $M = (Q, \delta, start, halt)$, where Q is the finite type of states, $\delta : Q \times (O(\Sigma))^n \to Q \times (\mathsf{Act}_\Sigma)^n$ (with* actions $\mathsf{Act}_\Sigma := O(\Sigma) \times \mathsf{Move}$ *and* Move $::= \mathsf{L} \mid \mathsf{R} \mid \mathsf{N}$*) is the transition function, $start : Q$ is the initial state and $halt : Q \to \mathbb{B}$ represents the halting states.*

If the machine is not clear from the context, we may write e.g. Q_M for disambiguation.

4 Labelled Turing Machines, Correctness, and Termination (L_1)

In order to abstract away from the state space of Turing machines, we introduce labelled machines:

Definition 4.1. *A* labelled machine *over a type L, written $M : \mathsf{TM}_\Sigma^n(L)$, is a dependent pair (M', lab_M) of a machine $M' : \mathsf{TM}_\Sigma^n$ and a labelling function $\mathsf{lab}_M : Q_{M'} \to L$.*

We identify unlabelled machines $M : \mathsf{TM}_\Sigma^n$ with unit-labelled machines $M : \mathsf{TM}_\Sigma^n(\mathbb{1})$ and use the symbol M for both, relying on the context for disambiguation.

To reason about the semantics of Turing machines, we have two predicates, one for *correctness* and one for *termination*. Those notions use a predicate $M(q, t) \triangleright^k (q', t')$, denoting that M with the tapes t as input starting in state q will terminate in no more than k steps in the final configuration (q', t'). When we omit q, we mean the initial state start_M. We postpone the definition of $\triangleright$ to the Appendix [Forster et al. 2019] as our framework enables reasoning to start on the level of correctness and termination.

For correctness, the realisation predicate uses a relation $R \subseteq \mathsf{Tape}^n \times (L \times \mathsf{Tape}^n)$ to connect the input tape t to the label of the final state q and tape content t' reached after the execution of a machine:

Definition 4.2. *Let $M : \mathsf{TM}_\Sigma^n(L), R \subseteq \mathsf{Tape}_\Sigma^n \times (L \times \mathsf{Tape}_\Sigma^n)$.*

$$M \vDash R := \forall t\ k\ q\ t'.\ M(t) \triangleright^k (q, t') \to R\ t\ (lab_M\ q, t')$$

For clarity, we often write $M : \mathsf{TM}_\Sigma^n(L) \vDash R$ to specify the alphabet, number of tapes and label type simultaneously.

Since realisation states that *if* the machine halts, its result has certain properties, we need a second predicate for termination, incorporating the running time:

Definition 4.3. *Let $T \subseteq \mathsf{Tape}_\Sigma^n \times \mathbb{N}$.*

$$M \downarrow T := \forall t\ k.\ T\ t\ k \to \exists c.\ M(t) \triangleright^k c.$$

The introduced predicates are (anti-)monotone:

Fact 4.4. *The following hold:*

1. *If $M \vDash R'$ and $R' \subseteq R$, then $M \vDash R$.*
2. *If $M \downarrow T'$ and $T \subseteq T'$, then $M \downarrow T$.*

For simple machines, a realisation predicate that entails termination in a constant number k of steps is useful:

Definition 4.5.

$$M \vDash^k R := \forall t.\ \exists q\ t'.\ M(t) \triangleright^k (q, t') \wedge R\ t\ (\mathsf{lab}_M\ q,\ t')$$

Fact 4.6. $M \vDash^k R \leftrightarrow M \vDash R \wedge M \downarrow (\lambda\, t\, k'.\ k \le k')$

We will just write $\vDash^c$ instead of giving the numerical value of the constant.

Note that space complexity is a correctness property and can be included in relations R for realisation. Since we do not have explicit blank symbols in our representation of tapes, the number of used cells never decreases, which means that the size of the result of a computation bounds the space usage *during* the computation, i.e. we can bound the space usage by a function f proving $M \vDash (\lambda\, t\, t'.\ |t'| \le f(|t|))$, where $|t|$ denotes the number of symbols on a tape. Our

$$\mathsf{Read} : \mathsf{TM}^1_\Sigma(O(\Sigma)) \vDash^c \lambda\ t\ (\ell, t').\ \ell = \mathsf{current}\ (t[0]) \wedge t = t'$$

$$\mathsf{Write}\ s : \mathsf{TM}^1_\Sigma \vDash^c \lambda\ t\ t'.\ t'[0] = \mathsf{tape\_write}\ s\ t[0]$$

$$\mathsf{Move}\ d : \mathsf{TM}^1_\Sigma \vDash^c \lambda\ t\ t'.\ t'[0] = \mathsf{tape\_move}\ d\ t[0]$$

$$\mathsf{Nop} : \mathsf{TM}^n_\Sigma \vDash^c \lambda\ t\ t'.\ t' = t$$

Figure 1. Primitive Machines

framework does not commit to whether the maximal size of tapes or the sum of the sizes shall be considered as space measure; in our verified machines, we choose to bound each tape individually. In Section 6 we will introduce more explicit support for compositional space analysis.

5 Combining Machines (L_2)

In this section we define primitive machines, combinators to compose machines and lifting operations to change the number of tapes or the alphabet of a machine.

The idea is that every machine M defined using composition and lifting of primitive machines is equipped with canonical relations R_M and T_M and proofs $M \vDash R_M$ and $M \downarrow T_M$. To prove that $M \vDash R$ for a different relation R, a user only has to prove $R_M \subseteq R$, and for $M \downarrow T$, $T \subseteq T_M$ by fact 4.4.

In the end, we obtain a shallowly embedded, composable, imperative programming language operating on tapes, where a user does not have to mention states or transition functions at all.

To keep this section short, we omit the precise constructions of machines M and only give their correctness relation R_M and running time relation T_M.

5.1 Primitive Machines

The machines that perform the most basic operations a Turing machine can do are given in Figure 1, together with the most general relation they realise. Those relations use functions operating on tapes defined in the Appendix [Forster et al. 2019]. The fact that all those machines can run in a single step allows us to show termination and realisation in one step, by using $\vDash^c$.

Read is a 1-tape machine which returns in a final state that is labelled with the symbol under the head, if there is one, and does not change the tape. Write s, for any $s : \Sigma$, is a 1-tape machine that writes the constant symbol s at head position and does not return any result. Move d is a 1-tape machine that moves the head in the direction denoted by d. Finally, Nop is a no-op machine which is useful as a neutral element in our compositional framework.

5.2 Machine Combinators

We define machine combinators like sequential composition $M_1; M_2$, a case distinction operator Switch, and a 'do-while' operator While.

$$\frac{M : \mathsf{TM}^n_\Sigma(L) \vDash R \qquad \forall(\ell : L).\ f(\ell) : \mathsf{TM}^n_\Sigma(L') \vDash R'_\ell}{\mathsf{Switch}\ M\ f : \mathsf{TM}^n_\Sigma(L') \vDash \lambda t_0\ (\ell', t').\ \exists t\ (\ell : L).\ R\ t_0\ (\ell, t) \wedge R'_\ell\ t\ (\ell', t')}$$

$$\frac{M' \downarrow T' \qquad M' \vDash R' \qquad \forall(\ell : L).f(\ell) : \mathsf{TM}^n_\Sigma(L') \downarrow T_\ell}{\mathsf{Switch}\ M'\ f \downarrow \lambda\ t\ k.\ \exists\ k_1\ k_2.\ T'\ t\ k_1 \wedge 1 + k_1 + k_2 \leq k \wedge \forall\ \ell\ t'.\ R\ t\ (\ell, t') \rightarrow T_l\ t'\ k_2}$$

$$\frac{M : \mathsf{TM}^n_\Sigma(O(L)) \vDash R}{\mathsf{While}\ M : \mathsf{TM}^n_\Sigma(L) \vDash \mathsf{WhileRel}\ R}$$

$$\frac{M \downarrow T \qquad M \vDash R}{\mathsf{While}\ M \downarrow \mathsf{WhileT}\ R\ T}$$

$$\frac{M : \mathsf{TM}^n_\Sigma(L) \vDash R \qquad \ell' : L'}{\mathsf{Return}_{\ell'}\ M : \mathsf{TM}^n_\Sigma(L') \vDash \lambda\, t.(\ell, t').\ \ell = \ell' \wedge \exists \ell_0.R\, t\, (\ell_0, t')}$$

$$\frac{M \downarrow T}{\mathsf{Return}_\ell\ M \downarrow T}$$

$$M_1; M_2 := \mathsf{Switch}\ M_1\ (\lambda\ \_.\ M_2)$$

$$\mathsf{If}\ M_1\ \mathsf{Then}\ M_2\ \mathsf{Else}\ M_3 := \mathsf{Switch}\ M_1\ (\lambda\ b.\mathsf{if}\ b\ \mathsf{then}\ M_2\ \mathsf{else}\ M_3)$$

Figure 2. Combinators with Corresponding Relations

The semantics of e.g. sequential composition $M_1; M_2$ is easy to give informally: run M_1 and, on the resulting tape, run M_2. The correctness relation is then the composition of the correctness relations of the composed machines, e.g. if $M_1 : \mathsf{TM}^n_\Sigma(\mathbb{1}) \vDash R_1$ and $M_2 : \mathsf{TM}^n_\Sigma(\mathsf{L}_2) \vDash R_2$, then $M_1; M_2 : \mathsf{TM}^n_\Sigma(\mathsf{L}_2) \vDash R_1 \circ R_2$.

The running time relations can be composed by splitting the allowed running time between the two machines: If $M_1 \vDash R_1$, $M_1 \downarrow T_1$, and $M_2 \downarrow T_2$, then

$$M_1; M_2 \downarrow (\lambda t\ k.\ \exists k_1\ k_2.\ T_1\ t\ k_1\ \wedge\ 1 + k_1 + k_2 \leq k \wedge\ \forall t'\ \ell.\ R_1\ t\ (\ell, t') \rightarrow T_2\ t'\ k_2).$$

Here, as the second machine only runs on results of the first machine, the correctness of the first machine can be used to weaken the requirement on the running time relation of the second machine.

The case distinction operator Switch takes as arguments a machine $M : \mathsf{TM}^n_\Sigma(L)$ labelled over L and, for each possible label, another machine labelled over L', which is represented by a function $f : L \rightarrow \mathsf{TM}^n_\Sigma(L')$. It runs M first, and depending on the label ℓ returned by M, runs $f\ell$ on the resulting tape. We specify Switch in Figure 2. The finiteness of F (the domain of f) is crucial here to encode f into the finite state space of the resulting Turing machine.

In fact, sequential composition is a special case of Switch that does ignore the label of the first machine. The If combinator is also a special case of Switch (over Boolean labels), we thus leave it out here.

Loops can be implemented with the While M combinator. It takes a machine M labelled over $O(L)$ and runs M repeatedly as long as the resulting label is $\emptyset$. Once the label is $\lfloor \ell \rfloor$, While M returns ℓ. The correctness relation of While in Figure 2 uses the following inductively defined relation:

Definition 5.1 (Correctness of While). *Assuming* $R \subseteq \mathsf{Tape}_\Sigma^n \times O(L) \times \mathsf{Tape}_\Sigma^n$, *then* $\mathsf{WhileRel}\ R \subseteq \mathsf{Tape}_\Sigma^n \times L \times \mathsf{Tape}_\Sigma^n$ *is inductively defined by the following two rules:*

$$\frac{R\ t\ (\lfloor \ell \rfloor, t')}{\mathsf{WhileRel}\ R\ t\ (\ell, t')} \qquad \frac{R\ t\ (\emptyset, t') \qquad \mathsf{WhileRel}\ R\ t'\ (\ell, t'')}{\mathsf{WhileRel}\ R\ t\ (\ell, t'')}$$

The first rule corresponds to returning when some label other than $\emptyset$ was produced, and the second rule corresponds to looping once more in the other case. To verify that some machine M using While realises some relation R', we would usually first find a relation R realised by M. In a second step, one proves $R' \subset \mathsf{WhileRel}\ R$, usually by some sort of inductive reasoning, and uses fact 4.4 to deduce that While M realises R'.

Termination without running time could be obtained by requiring some sort of well-foundedness of the relation realised by M. For termination with time complexity, we use a co-inductively defined relation:

Definition 5.2 (Running Time for While). *Assuming* $M : \mathsf{TM}_\Sigma^n(O(L))$ *and realisation relations* R *and running time relation* T *(whose types fit for* M*), we define the co-inductive running time relation* $\mathsf{WhileT}\ R\ T$ *by*

$$\frac{T\ t\ k_1 \qquad \forall t'\ l.\ R\ t\ (\lfloor l \rfloor, t') \to k_1 \le k \qquad \forall t'.\ R\ t\ (\emptyset, t') \to \exists k_2.\ \mathsf{WhileT}\ R\ T\ t'\ k_2 \wedge 1 + k_1 + k_2 \le k}{\mathsf{WhileT}\ R\ T\ t\ k}$$

To now show that $\mathsf{While} M \downarrow T'$, it suffices (by fact 4.4) to show $T' \subseteq \mathsf{WhileT}\ R\ T$, which follows from

$$\begin{aligned}
&\forall t\ k.\ T\ t\ k \to \\
&\quad \exists k_1.\ T'\ t\ k_1 \wedge \\
&\qquad (\forall l\ t'.\ R\ t\ (\lfloor l \rfloor, t') \to k_1 \le k) \ \wedge \\
&\qquad (\forall t'.\ R\ t\ (\emptyset, t') \to \exists k_2.\ T\ t'\ k_2 \ \wedge\ 1 + k_1 + k_2 \le k)\,.
\end{aligned}$$

due to the co-inductive definition of WhileT.

Finally, altering a machine $M : \mathsf{TM}_\Sigma^n(L)$ to always return some constant label can be useful, for example to use it inside a Switch or While combinator. It can be defined by only adjusting the labels: $\mathsf{Return}_\ell\ M := (M, \lambda\_.\ \ell)$. This also means that termination is not altered at all, as termination is actually defined on the bare, unlabelled machine.

$$\frac{M : \mathsf{TM}_\Sigma^m(L) \vDash R \qquad I : (\mathbb{F}_n)^m \text{ duplicate free}}{\Uparrow_I M : \mathsf{TM}_\Sigma^n \vDash \Uparrow_I R}$$

$$\frac{M \downarrow T \qquad I : (\mathbb{F}_n)^m \text{ duplicate free}}{\Uparrow_I M \downarrow \Uparrow_I T}$$

$$\frac{M : \mathsf{TM}_\Sigma^n(L) \vDash R \qquad f : \Sigma \hookrightarrow \Gamma \text{ retraction} \qquad d \in \Gamma}{\Uparrow_{(f,d)} M : \mathsf{TM}_\Gamma^n \vDash \Uparrow_{(f,d)} R}$$

$$\frac{M \downarrow R \qquad f : \Sigma \hookrightarrow \Gamma \text{ retraction} \qquad d \in \Gamma}{\Uparrow_{(f,d)} M \downarrow \Uparrow_{(f,d)} T}$$

Figure 3. Machine Liftings

5.3 Lifting Machines

The combinators we have seen can only compose machines with the same alphabet and number of tapes. To allow the combination of other machines, we introduce lifting functions. The general idea is to extend the number of tapes (or the alphabet) of a machine or relation by renaming them consistently: The 'new' tapes/symbols are just ignored and never actively used in the lifted machines.

The *tape-lift* $\Uparrow_I(\cdot)$ is concerned with changing the number and/or order of tapes. Assuming that we have an m-tape machine M, we can lift this into an n-tape machine as long as $m \le n$. The information how to reorder the tapes can be encoded as a duplicate free vector I containing m elements of the finite type $\mathbb{F}_n$. The vector $[0, 3, 1] : (\mathbb{F}_4)^3$ can be used to convert a 3-tape machine into a 4-tape machine. Tape 0 is kept in the same place, tape 1 becomes tape 3 and tape 2 becomes tape 1. Tape 2 of the resulting machine is unused.

The transition function of $\Uparrow_I M$ uses I to permute all read and write operations on the tapes accordingly. All of the tapes not mentioned in I are never read or written in $\Uparrow_I M$.

To add the corresponding rules in Figure 3, we define tape-lifts for realisation and running time relations in the Appendix [Forster et al. 2019]: We write $\Uparrow_I R$ and $\Uparrow_I T$ for them.

The *alphabet-lift* $\Uparrow_{(f,d)}(\cdot)$ changes a machine $M : \mathsf{TM}_\Sigma^n(L)$ over an alphabet Σ to a machine $\Uparrow_{(f,d)} M : \mathsf{TM}_\Gamma^n(L)$ for any retraction $f : \Sigma \hookrightarrow \Gamma$ and default symbol $d : \Sigma$.

The idea of the alphabet-lift of a machine is to translate every read symbol using f^{-1} and every written symbol using f; thus the transition relation of M, which originally operated on Σ, can now operate on Γ. Every read symbol which is not hit by f, e.g. for which f^{-1} returns $\emptyset$, is translated to d. In practice, when constructing machines, one thus always wants that M does not normally read d.

To complete Figure 3, we again overload the alphabet-lift for relations, which we write as $\Uparrow_{(f,d)} R$ and $\Uparrow_{(f,d)} T$. Their definitions are in the Appendix [Forster et al. 2019].

During proofs, tape-lifts result in renaming some tapes and equalities of unmentioned tapes. To deal with many

equations, we develop automation in Coq, for more details see Section 9. Alphabet-lifts that occur during the use of the abstractions presented in Section 6 are handled by our automation as well.

5.4 Verifying Simple Machines

As an example, we construct and verify the two-tape machine CopyLeft which copies the symbol on the left of the head of tape 0 to the current position on tape 1.

Definition 5.3. *We define* $\mathsf{CopyLeft} : \mathsf{TM}^2_\Sigma$ *as*

$$
\begin{aligned}
&\mathsf{CopyLeft} :=\\
&\quad \Uparrow_{[0]} \mathsf{Move\ L}\,;\\
&\quad \mathsf{Switch}\ (\Uparrow_{[0]} \mathsf{Read})\\
&\qquad\qquad (\lambda(\ell : O(\Sigma)).\ \mathsf{match}\ \ell\ \mathsf{with}\\
&\qquad\qquad\quad \lfloor s \rfloor \Rightarrow \Uparrow_{[1]} \mathsf{Write}\ s\\
&\qquad\qquad\quad \mid \emptyset \Rightarrow \mathsf{Nop})
\end{aligned}
$$

To shorten the example, we will verify a weak relation that omits that all other cells on tape 1 remain unchanged and does not fully specify what happens if there is no symbol to read.

Lemma 5.4. $\mathsf{CopyLeft}\ d_0\ d_1 \vDash \mathsf{CopyLeftRel}$ *with*

$$
\begin{aligned}
\mathsf{CopyLeftRel} := \lambda t\ t'.\ & t[0] = t'[0]\\
& \wedge \forall (s : \Sigma).\ \mathsf{current}\ (\mathsf{move\ L}\ t[0]) = \lfloor s \rfloor\\
& \qquad \rightarrow \mathsf{current}\ t'[1] = \lfloor s \rfloor
\end{aligned}
$$

Proof. The rules from Figures 1 to 3 directly yield a correctness relation for CopyLeft, which can be used in combination with fact 4.4 to reduce our proof obligation to a relational inclusion:

$$
\begin{aligned}
&\Uparrow_{[0]} (\lambda\ t_0\ t_1.\ t_1[0] = \mathsf{tape\_move\ L}\ t_0[0])\\
&\circ \big(\lambda\ t_1\ ((\ell' : \mathbb{1}), t_3).\\
&\ \exists t_2\ (\ell : O(\Sigma)).\ (\Uparrow_{[0]} (\lambda\ t_1\ (\ell, t_2).\ \ell = \mathsf{current}(t[0]) \wedge t_2 = t_1))\\
&\qquad\qquad t_1\ (\ell, t_2)\\
&\qquad \wedge (\mathsf{match}\ \ell\ \mathsf{with}\\
&\qquad\quad \lfloor s \rfloor \Rightarrow \Uparrow_{[1]} (\lambda\ t_2\ t_3.\ t_3[0] = \mathsf{tape\_write}\ s\ t_2[0])\ t_2\ t_3\\
&\qquad\quad \mid \emptyset \Rightarrow t_3 = t_2)\big)\\
&\subseteq \mathsf{CopyLeftRel}
\end{aligned}
$$

This inclusion follows by unfolding all definitions and basic logical transformations. □

We could prove the running time for CopyLeft in a similar fashion. Note that these kinds of proofs are very mechanical as long as no While loop is involved. We only need to unfold the relations and rewrite tapes, which can be automated in Coq. We will not detail more proofs of this kind on paper.

6 Generalised Register Machines (L_3)

We now come to the highest layer of abstraction of our framework, where we treat Turing machines as abstract register machines. Each tape of such an abstract register machine contains a value of an encodable inductive type. We first define the notion of encodable types and tape containment, and then show how to specify machines constructing values. Afterwards, we show how case analysis can be implemented and give an exemplary implementation of addition and multiplication of natural numbers as abstract register machines.

6.1 Tapes as Registers

Definition 6.1. *A type X is encodable on Σ if there is an injective function $\varepsilon : X \to \mathcal{L}(\Sigma)$, where Σ is a finite type.*

For encodable types X we can define the size of elements $x : X$ as the number $|\varepsilon(x)|$. We can also define what it means for a tape t to contain a value $x : X$. Note that if the type X is encodable, the proof of encodability gives a canonical alphabet that can encode X, which we denote by Σ_X. In addition, X is by definition encodable on Γ given any $f : \Sigma_X \hookrightarrow \Gamma$. We make this distinction between canonical alphabets and extension via retractions explicit and define $t \simeq_{f,k} x$. Here, $x : X$, $f : \Sigma_X \hookrightarrow \Gamma$, $k : \mathbb{N}$ is denoting the number of symbols on the tape that are irrelevant to the encoding and t is a tape over the alphabet Γ^+, which extends Γ by symbols denoting the start and end of the encoding:

Definition 6.2. *Let Γ be a type. We define*

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

Definition 6.3. *Let X be encodable on Σ, $f : \Sigma \hookrightarrow \Gamma$ and $t : \mathsf{Tape}_{\Gamma^+}$. We define*

$$
\begin{aligned}
t \simeq_{f,k} x :=& \exists\ ls.\ |ls| \leq k \wedge\\
& t = \mathsf{midtape}\ ls\ \mathsf{START}\ (f@(\varepsilon(x)) +\!\!+ [\mathsf{STOP}]).
\end{aligned}
$$

Note that the length of the tape (and thus the space consumption of the machine up to this point) is exactly $k+|\varepsilon x|+2$. If we write $t \simeq_k x$, we mean that f is the identity retraction and thus $t : \Sigma^+_X$. Explicitly mentioning the additional retract f is useful when combining machines over different alphabets: The alphabet-lift that are then required can be composed and stored into the retract, e.g. the relation $\Uparrow_{(g,d)} (\lambda t\ t'.t'[0] \simeq_{f,k} x)$ is the same as $(\lambda t\ t'.t'[0] \simeq_{g^{-1} \circ f,k} x)$. This enables our framework to hide alphabet lifts to the user.

The special symbol UNKNOWN is used globally as the default-symbol for the alphabet-lift. It is never read/written from/to a tape.

Lemma 6.4. *The types $\mathbb{1}$, $\mathbb{B}$, and $\mathbb{N}$ are all encodable. If X and Y are encodable, then $O(X)$, $\mathcal{L}(X)$, $X + Y$, and $X \times Y$ are encodable.*

Proof. The type $\mathbb{N}$ is encodable on the alphabet $\Sigma_{\mathbb{N}} ::= \mathsf{S} \mid \mathsf{O}$:

$$
\begin{aligned}
\varepsilon\ (0) &:= [\mathsf{O}]\\
\varepsilon\ (S\ n) &:= \mathsf{S} :: \varepsilon(n)
\end{aligned}
$$

The encodings for the other types work similarly. □

Using this definitions, we have

$$\text{midtape } ls \text{ START } [\text{S}, \text{S}, \text{S}, \text{O}, \text{STOP}] \simeq_{|ls|} 3$$

for all contents ls to the left — that is, we ignore garbage on the tape (apart from the length) and only require that to the right of the START symbol we have exactly an encoding and the STOP symbol. As an invariant, we will never consider empty tapes on the level of abstract register machines. This makes realisation proofs easier, because it restricts the shape of tapes. Thus, we define $\text{isVoid}_k(t)$ to mean that t is non-empty, the head points to the right-most symbol and there are at most k other symbols.

Definition 6.5. $\text{isVoid}_k(t) := \exists m\, ls.\, t = \text{midtape } ls\, m\, \text{nil} \wedge |ls| \leq k$.

We can now define our first abstract register machine, a machine that statically writes a value into a register given its encoding:

Lemma 6.6. *Let X be encodable over Σ and $str : \mathcal{L}(\Sigma)$. There is a machine* $\text{WriteValue}(str) : \text{TM}^1_\Sigma$ *s.t.:*

$$\text{WriteValue}(str) \vDash^{3+2\cdot|str|} \lambda t\, t'.\ \forall x\, k.\, \varepsilon(x) = str \to \text{isVoid}_k\, t[0] \to t'[0] \simeq_{k-|\varepsilon(x)|-1} x.$$

Note that the machine takes a string instead of a value as static input to avoid parametrising the machine over encoding functions, which then is only necessary for the correctness statement.

Similarly, we can reset a tape which contains a value and make it void:

Lemma 6.7. *Let X be encodable over Σ_X and $f : \Sigma_X \hookrightarrow \Gamma$. There is a machine* $\text{Reset} : \text{TM}^1_\Gamma$ *s.t.:*

$$\text{Reset} \vDash \lambda t\, t'. \forall x k. t[0] \simeq_{f,k} x \to \text{isVoid}_{1+k+|\varepsilon(x)|}\, t'[0]$$

Lastly, we can change the retraction explicitly by inserting a translation machine which does not actually change tapes:

Lemma 6.8. *Let X be encodable over Σ_X and $f, g : \Sigma_X \hookrightarrow \Gamma$. There is a machine* $\text{Translate}\, f\, g : \text{TM}^1_\Gamma$ *s.t.:*

$$\text{Translate}\, f\, g \vDash \lambda t\, t'. \forall x k. t[0] \simeq_{f,k} x \to t'[0] \simeq_{g,k} x$$

6.2 Constructor and Destructor Machines

Building on the previous definitions and machines we implement the two constructors of natural numbers:

Lemma 6.9. *There is a machine* $\text{ConstrO} : \text{TM}^1_{\Sigma_\mathbb{N}}$ *s.t.*

$$\text{ConstrO} \vDash^5 \lambda t\, t'.\ \text{isVoid}_k\, t[0] \to t'[0] \simeq_{k-2} 0.$$

Proof. Define $\text{ConstrO} := \text{WriteValue}(\varepsilon 0)$. □

The S constructor overwrites the current symbol (which is the start symbol) with S and writes a new start symbol one step left.

Lemma 6.10. *There is a machine* $\text{ConstrS} : \text{TM}^1_{\Sigma_\mathbb{N}}$ *s.t.*

$$\text{ConstrS} \vDash^3 \lambda t\, t'.\ \forall n k.\ t[0] \simeq_k n \to t'[0] \simeq_{k-1} S\, n.$$

We now define the destructor machine for $\mathbb{N}$. As all destructor machines will, it analyses the head constructor on the tape, remove it and signal which constructor was found via labels. For the case of natural numbers, the destructor machine CaseNat reads a number from tape $t[0]$. If the number is 0, CaseNat leaves the number unchanged and terminates in the label false. Else, it decreases the number and terminates in true.

Lemma 6.11. *There is a machine* $\text{CaseNat} : \text{TM}^1_{\Sigma_\mathbb{N}}(\mathbb{B})$ *s.t.*

$$\begin{aligned}&\text{CaseNat} \vDash^5\\ &\quad \lambda t\, (\ell, t').\ \forall (n : \mathbb{N}).\ t[0] \simeq_k n \to \text{match } \ell, n \text{ with}\\ &\qquad \text{false}, 0 \Rightarrow t'[0] \simeq_k 0\\ &\qquad \mid \text{true}, S\, n' \Rightarrow t'[0] \simeq_{k+1} n'\\ &\qquad \mid \_, \_ \Rightarrow \bot\end{aligned}$$

Proof. We implement the machine using Switch and Read:

```
CaseNat :=
  Move R ;
  Switch Read
      (λ(s : O(Σ_ℕ^+)). match s with
          ⌊S⌋ ⇒ Return_true (Write START)
        | ⌊O⌋ ⇒ Return_false (Move L)
        | _ ⇒ _)
```

□

We implement and use similar machines for Booleans (ConstrTrue, ConstrFalse, CaseBool), options (ConstrNone, ConstrSome, CaseOption), and lists (ConstrNil, ConstrCons, CaseOption), but leave out their definitions in the paper.

6.3 Case Study: Addition and Multiplication

We now explain how to actually implement recursive abstract register machines. We will not require any direct operations on tapes and only use combinators from L_2 and constructors and destructors from L_3.

Recall that the While M operator corresponds to 'do-while' in imperative languages, i.e. the machine M has to decide at the end of its execution whether to continue or break out of the loop. Tail-recursive functions can easily be transformed into 'do-while' loops. Thus, when we translate functions to Turing machines, we first have to implement the function as a tail-recursive function.

As an example, we implement addition Add and multiplication Mult for natural numbers.

The algorithm for addition can be described with the following pseudocode:

```
a ← n
b ← m
While (b−−) {
  a++
}
Reset b
```

Tape of Add	Variable	Tape in AddStep
0	m	–
1	n	–
2	a	0
3	b	1

Figure 4. Tape assignment for Add and AddStep

The output tape is the tape that is represented by the variable a. First, we copy the input n to this tape, and the number m to an internal tape. In the loop, as long as we can decrease the copy of m, we increment a. After the loop, we reset the copy b and the machine terminates. The first step in the design of the machine is to specify, which tape contains which variable. This is visualised in Figure 4.

Because the machine only operates on natural numbers, we choose $\Sigma^+_{\mathbb{N}}$ as the alphabet of Add and all its sub-machines.

The next step is to implement the step machine. AddStep has only access to the variables a and b, stored on tape 0 and 1 (fig. 4). The decrement operation and test whether b was 0 is implemented using the deconstructor machine CaseNat. In the case that b is 0, the step machine terminates in $\lfloor() \rfloor$, so that the loop breaks. In case b is greater than 0, CaseNat decreases b and the step machine increases a. Then it terminates in $\emptyset$, so that the loop continues.

Definition 6.12. AddStep :=
If ($\Uparrow_{[1]}$ CaseNat) Then (Return$_{\emptyset}$ ($\Uparrow_{[0]}$ ConstrS)) Else (Return$_{\lfloor() \rfloor}$ Nop)

Because all parts of AddStep terminate in constant time, we get the constant running time part of the semantics of AddStep for free.

Lemma 6.13. AddStep $\vDash^9$ AddStepRel :=

$$\begin{aligned}
&\lambda\, t\ (\ell, t').\ \forall a\ b\ k_a\ k_b.\\
&\quad t[0] \simeq_{k_a} a \longrightarrow t[1] \simeq_{k_b} b \longrightarrow\\
&\quad \text{match } \ell, b \text{ with}\\
&\qquad \lfloor() \rfloor, O \Rightarrow t'[0] \simeq_{k_a} a \ \wedge\ t'[1] \simeq_{k_b} b\\
&\qquad \mid \emptyset, S\ b' \Rightarrow t'[0] \simeq_{k_a - 1} S\ a \ \wedge\ t'[1] \simeq_{1+k_b} b'\\
&\qquad \mid _, _ \Rightarrow \bot
\end{aligned}$$

According to the general design plan, we define AddLoop := While AddStep. The correctness relation of AddLoop now says, that after the execution of the loop, $t'[0]$ contains $a+b$ and $t'[1]$ contains 0:

Lemma 6.14. AddLoop $\vDash \lambda t\ t'.\ \forall a b k_a k_b.$

$$t[0] \simeq_{k_a} a \longrightarrow t[1] \simeq_{k_b} b \longrightarrow t'[0] \simeq_{k_a - b} a+b \wedge t'[1] \simeq_{k_b+b} 0.$$

Proof. By fact 4.4 and induction on the structure of WhileRel. In the case the loop terminates, b is 0 and $t'[0] \simeq a$, therefore $t'[0]$ contains $a = a + 0 = a + b$. In the induction/loop case, we know that $b = S\ b'$ and that $t'[0] \simeq S\ a$ and $t'[1] \simeq b'$. By the inductive hypothesis, we know that $t''[0]$ contains $b' + S\ a = a + b$ and $t''[1] \simeq 0$. □

Input	CopyValue	CopyValue	AddLoop	Reset
$0 : m$		$0 : m$		
$1 : n$	$0 : n$			
$2 : \dashv$	$1 : n$		$0 : m + n$	
$3 : \dashv$		$1 : m$	$1 : 0$	$0 : \dashv$

Figure 5. Execution protocol of Add

The running time of AddLoop must be shown separately. We know that the loop is executed $b + 1$ times, and each iteration takes 9 steps. We have to add 1 step for each re-iteration of the loop. Thus, the total step number is $9 + 10 \cdot b$.

Lemma 6.15. AddLoop $\downarrow \lambda t\ k.$

$$\exists (a\ b : \mathbb{N}).\ t[0] \simeq a \wedge t[1] \simeq b \wedge 9 + 10 \cdot b \leq k$$

Now we can define the full machine Add:

Definition 6.16. Add := $\Uparrow_{[1,2]}$ CopyValue; $\Uparrow_{[0,3]}$ CopyValue; $\Uparrow_{[2,3]}$ AddLoop; $\Uparrow_{[3]}$ Reset.

At this point, we introduce a graphical notation of *execution protocols*, that show the value of each tape after the execution of each sub-machine. In the left column, we have the input values for each tape, or $\dashv$ if the tape is initially void. Each further column denotes the (lifted) sub-machines. We write entries $j : x$ in each cell that is in the index-vector of the sub-machine, where j is the tape-index of the lifted machine and x is the value after the execution of the sub-machine on this tape. We write $j : \dashv$ when the tape j is void after the execution of the sub-machine. If a cell of the table is empty, then the tape has not changed, thus the current value is found further left in the same row. In Figure 5, we have an example of an execution protocol for Add.

From the execution protocol in Figure 5, we see that after the execution of all four sub-machines, the tapes 0 and 1 still contain the values m and n, tape 2 contains $m + n$, and tape 3 is void. Execution protocols serve as outlines of the formal correctness proofs. We conclude the correctness of Add.

Lemma 6.17.

$$\begin{aligned}
&\text{Add} \vDash \lambda t t'.\ \forall m\ n\ k_0\ k_1\ k_2\ k_3.\\
&\quad t[0] \simeq_{k_0} m \longrightarrow t[1] \simeq_{k_1} n \longrightarrow\\
&\quad \text{isVoid}_{k_2}(t[2]) \longrightarrow \text{isVoid}_{k_3}(t[3]) \longrightarrow\\
&\quad t'[0] \simeq_{k_0} m \wedge t'[1] \simeq_{k_1} n \wedge t'[2] \simeq_{k_2+m-n-2} (m+n) \wedge\\
&\quad \text{isVoid}_{2+(k_3-(2+m)+m)}
\end{aligned}$$

For the running time function of Add, we have to add linear components for copying m and n, a constant for Reset, and 1 step for each sequential composition operator.

Lemma 6.18. Add $\downarrow \lambda t\ k.\ \exists m\ n.\ t[0] \simeq m \wedge t[1] \simeq n \wedge$ isVoid$(t[2]) \wedge$ isVoid$(t[3]) \wedge 98 + 22 \cdot m + 12 \cdot n \leq k$.

When we prove the running time of sequences of multiple machines, we have to give running time functions for all suffixes of the sequence in terms of the sequence operator.

We use the machine Add to implement a machine Mult that computes the multiplication function $mult : \mathbb{N} \to \mathbb{N} \to \mathbb{N}$. The machine Mult 'calls' the machine Add m-times to add n to a counter c that is initialised with 0. The following pseudo code presents the algorithm that we implement:

```
c  ← 0
m' ← m
While (m'--) {
  c' ← Add(n, c)
  Reset c
  c ← c'
  Reset c'
}
Reset m'
```

Note that we do not have to copy n, since we do not write on n and the machine Add does not change n. Also note that we cannot simply add n to c, since input and output tapes are distinct. Therefore, we have to store the result of the addition to an intermediate variable (i.e. internal tape), which has to be reset afterwards.

MultStep, has only access to the copy of m, which is mapped to tape 0 of MultStep. In the following, we also use the name m for its copy in the context of MultStep.

Definition 6.19. MultStep :=

If $\Uparrow_{[0]}$ CaseNat

Then Return$_{\emptyset}$ $\left(\Uparrow_{[1,2,3,4]} \text{Add}; \Uparrow_{[2]} \text{Reset}; \Uparrow_{[3,2]} \text{CopyValue}; \Uparrow_{[3]} \text{Reset}\right)$

Else Return$_{\lfloor () \rfloor}$ Nop.

The correctness relation of MultStep is analogous to AddStep.

Lemma 6.20. MultStep ⊨ MultStepRel *with*

$$
\begin{aligned}
&\text{MultStepRel} := \\
&\quad \lambda\, t\ (\ell, t').\ \forall (c\ m\ n : \mathbb{N}) k_0\ k_1\ k_2\ k_3\ k_4. \\
&\qquad t[0] \simeq_{k_0} m \to t[1] \simeq_{k_1} n \to t[2] \simeq_{k_2} c \to \\
&\qquad \text{isVoid}_{k_3}\ t[3] \to \text{isVoid}_{k_4}\ t[4] \to \\
&\qquad \text{match } \ell, m \text{ with} \\
&\qquad [\lfloor () \rfloor, O \Rightarrow t'[0] \simeq_{k_0} m \ \wedge\ t'[1] \simeq_{k_1} n \ \wedge\ t'[2] \simeq_{k_2} c \\
&\qquad\qquad \wedge\ \text{isVoid}_{k_3}\ t'[3] \ \wedge\ \text{isVoid}_{k_4}\ t'[4] \\
&\qquad | \emptyset, S\ m' \Rightarrow t'[0] \simeq_{k_0+1} m' \ \wedge\ t'[1] \simeq_{k_1} n \ \wedge\ t'[2] \simeq_{k_2-n} n + c \\
&\qquad\qquad \wedge\ \text{isVoid}_{(2+n+c+((k_3+n)-c-2))}\ t'[3] \\
&\qquad\qquad \wedge\ \text{isVoid}_{(2+(k_4-(2+n)+n))}\ t'[4] \\
&\qquad | \_, \_ \Rightarrow \bot
\end{aligned}
$$

Note that for AddStep, we also prove termination in constant running time in lemma 6.13. This is not true for MultStep, because it calls Add, which has non-constant running time.

As usual, we define MultLoop := While MultStep. The correctness statement of MultLoop says that if the first three tapes contain m, n, c, and all other tapes are void, then after the execution of the loop, the tapes contain 0, n, and $m \cdot n + c$ and the other tapes are void. Thus, when we instantiate the c-tape with the value 0, the output tape contains $m \cdot n$.

Lemma 6.21. MultLoop ⊨ MultLoopRel *with*

$$
\begin{aligned}
&\text{MultLoopRel} := \lambda t\ t'.\ \forall (c\ m\ n : \mathbb{N})\ k_0\ k_1\ k_2\ k_3\ k_4. \\
&\quad t[0] \simeq_{k_0} m \to t[1] \simeq_{k_1} \to t[2] \simeq_{k_2} c \\
&\quad \to \text{isVoid}_{k_3}\ t[3] \to \text{isVoid}_{k_4}\ t[4] \to \\
&\quad t'[0] \simeq_{k_0+m} 0 \wedge t'[1] \simeq_{k_1} n \wedge t'[2] \simeq_{k_2-m\cdot n} m \cdot n + c \\
&\quad \wedge\ \text{isVoid}_P\ t'[3] \wedge \text{isVoid}_{P'}\ t'[4]
\end{aligned}
$$

Where P and P' are polynomials linear in m, n, c, k_3, and k_4.

It is now easy to define the rest of Mult.

Definition 6.22. Mult := $\Uparrow_{[0,5]}$ CopyValue; $\Uparrow_{[2]}$ ConstrO; $\Uparrow_{[5,1,2,3,4]}$ MultLoop; $\Uparrow_{[5]}$ Reset

It follows that Mult computes multiplication with time complexity in $O(m \cdot n^2)$ and linear space overhead. We omit the detailed theorems for brevity.

7 Multi-tape to Single-tape Compiler

All our constructions that use tapes as registers, as described in Section 6, are quite generous with the number of used tapes. It is a well-known fact that any n-tape Turing machine can be converted into an equivalent single-tape machine. In this section, we formalise such a compiler.

Given a labelled n-tape machine $M_n : \text{TM}^n_\Sigma(L)$ we want to define a single-tape machine $M_1 : \text{TM}^1_\Gamma(L)$ that simulates M_n. This means that if M_1 terminates, so does M_n, and vice versa. Additionally, the runtime overhead of M_1 is polynomial, and the space overhead is linear.

As in the literature (e.g. [Sipser 2006, p. 149]), we encode the vector of tapes on the single tape of M_1. The machine state of M_n is encoded in the machine state of M_1.

7.1 Encoding of Tapes

We first have to address how M_1 encodes a vector of tapes on its single tape. We follow Sipser [2006] and choose to encode tapes as a delimited list of (encodings of) tapes. The alphabet of M_1 includes markers for the current symbols of each tape and delimiters for the encoded tapes.

We define functions ε which encode vectors of tapes and tapes as strings. To encode vectors, every tape is delimited by '#' and the encoding of the entire vector of tapes is terminated by the symbol '$':

$$
\begin{aligned}
\varepsilon\ \text{nil} &:= [\$] \\
\varepsilon\ (t :: ts) &:= \# :: \varepsilon(t) +\!\!+ \varepsilon(ts)
\end{aligned}
$$

We still have to define how to encode a single tape as a string of symbols. Like Sipser, we have to mark the current symbol. However, there is a complication that is due to our tape model: On a multi-tape machine, moving the head one position after the rightmost symbol and back to the rightmost symbol on a tape does not change this tape at all. The encoding as a string has to reflect this property, in particular the length of the encoding must be invariant during head

movement. The length of the encoding may only increase when we allocate a new symbol.

These constraints give rise to the following alphabet for the tape encoding:

$$\Sigma_{tp} ::= \underline{B} \mid \overleftarrow{\underline{B}} \mid \overrightarrow{\underline{B}} \mid \overleftarrow{B} \mid \overrightarrow{B} \mid x \mid \underline{x} \qquad (x : \Sigma)$$

Underlined symbols denote that this is the current symbol of a tape. All encodings are delimited by the symbol B (for *boundary*) symbol, where the arrows indicate whether it is the left or right boundary. We then define the encoding of tapes as strings as follows:

$$\begin{aligned}
\varepsilon\ (\text{niltape}) &:= [\underline{B}] \\
\varepsilon\ (\text{leftof } r\ rs) &:= \overleftarrow{\underline{B}} :: r +\!\!+ rs +\!\!+ [\overrightarrow{B}] \\
\varepsilon\ (\text{midtape } l\ m\ rs) &:= \overleftarrow{B} :: \text{rev } ls +\!\!+ \underline{m} :: rs +\!\!+ [\overrightarrow{B}] \\
\varepsilon\ (\text{rightof } l\ ls) &:= \overleftarrow{B} :: \text{rev } ls +\!\!+ [l; \overrightarrow{\underline{B}}]
\end{aligned}$$

For example, the tape encodings in the above mentioned manoeuvre look like this:

$$[\overleftarrow{B}, a, b, \underline{c}, \overrightarrow{B}] \rightsquigarrow [\overleftarrow{B}, a, b, c, \overrightarrow{\underline{B}}] \rightsquigarrow [\overleftarrow{B}, a, b, \underline{c}, \overrightarrow{B}]$$

We add the delimiter symbols '#' and '$' to define the alphabet Γ of M_1. Additionally, we use the symbol START and STOP to delimit the entire encoding of the vector of tapes:

$$\Gamma ::= \sigma \mid \# \mid \$ \mid \text{START} \mid \text{STOP} \qquad (\sigma : \Sigma_{tp})$$

Note that START and STOP are in theory not neccessary, but it simplifies the implementation.

When M_1 starts, the head position is assumed to be on the START symbol. We then write $t \simeq tps$:

Definition 7.1 (Containment of tapes). *Let* $t : \text{Tape}_\Gamma$ *and* $tps : \text{Tape}_\Sigma^n$. *Then:*

$$t \simeq tps \ := \ t = \text{midtape } [] \text{ START } (\varepsilon(tps) +\!\!+ [\text{STOP}])$$

7.2 Implementation

The concrete implementation and verification is very technical; we only outline the definition of M_1 in this paper.

As noted above, we want to encode the machine states of M_n in the state of M_1. For that, we use the MemWhile operator, a combination of While and Switch described in the Appendix [Forster et al. 2019]. We have to give a function $\text{Step} : Q_M \to \text{TM}^1_\Gamma(Q_M + L)$. After we defined the function $\text{Step}(q)$, we define $M_1 := \text{MemWhile Step } (start_{M_n})$.

$\text{Step } q : \text{TM}^1_\Gamma(Q_M{+}L)$ is defined to be $\text{Return}_{\text{inl}(\text{lab}_{M_n} q)}$ Nop if q is a halting state of M_n. That way, Step q will terminate immediately with the right label. If q is not a halting state, Step q has to simulate one machine step of M_n. For that, it first scans the single tape to construct the vector of all current symbols. Since the set of all possible vectors $cs : O(\Sigma)^n$ of current symbols is finite, we can store it in the state space of Step q. Similarly, the computation of the tuple $(q', acts) = \delta_{M_n}(q, cs)$ can be encoded in the state space, where $q' : Q_{M_n}$ is the state in which M_n transitioned and $acts : \text{Act}^n$ is the vector of actions that have to be executed on each tape. After executing these actions, Step q terminates in the label $\text{inl } q'$, indicating that M_1 will continue to execute Step q' in the next loop iteration.

We use many auxiliary machines, like GoToCurrent, which starts at the # symbol of an encoded tape and halts at the current (i.e. marked) symbol of the selected tape. The machine $\text{GoToNext} : \text{TM}^1_\Gamma(\mathbb{B})$ starts at the current symbol and either goes to the # symbol of the next tape, or stops at $ and terminates in a state labelled with false, in the case that the head was already on the last tape.

For the auxiliary machines, the position of the tape head is an important part of their specifications. For example, we have a predicate atCons $t\ tps_1\ tps_2\ tp$ which states that the tape $t : \text{Tape}_\Gamma$ of M_1 is under a # symbol and to the left there is the encoding of the tapes tps_1; to the right, there is the encoding of the tape $tp : \text{Tape}_\Sigma$, and after this the encoding of the list of tapes tps_2. We also have to maintain invariants like $|tps_1| + 1 + |tps_2| = n$.

7.2.1 Reading all Current Symbols

In order to know which actions Step q has to emulate, it first has to scan all current (i.e. marked) symbols on the tapes. The machine $\text{ReadCurrentSymbols} : \text{TM}^1_\Gamma O(\Sigma)^n$ scans the list of tapes from left to right and stores the read symbols in a vector, and counts how many symbols it has already read. Note that an exponential blow-up of the states of Step q (w.r.t. n) seems to be inevitable because ReadCurrentSymbols has to account for all permutations of read symbols.

In order to store all read symbols, ReadCurrentSymbols has to do some 'book keeping'. It uses the MemWhile operator, with $O(\Sigma)^n \times \mathbb{F}_n$ as the internal state type. The vector contains the already read symbols, and the index i is the number of tapes that have already been scanned. After executing GoToCurrent and GoToNext, it increases the counter and repeats the loop if GoToNext terminated on a # symbol. Otherwise, if the $ symbol was reached, the counter i must have been $n - 1$; ReadCurrentSymbols will then terminate in the state that is labelled with the complete vector of read symbols $cs : O(\Sigma)^n$.

7.2.2 Executing Actions

After ReadCurrentSymbols has read all symbols, we can use the state q and the vector cs on the meta level to access the vector $acts : \text{Act}^n$. The machine $\text{DoActions } acts : \text{TM}^1_\Gamma$ scans the tape again and executes each action. Similar to ReadCurrentsymbols, it keeps track of the index i of the current tape.

The interesting part is the machine DoAction a, which executes the action $a : O(\Sigma) \times \text{Move}$.

Executing single actions consists of two steps: first, the current symbol is overwritten (if the $O(\Sigma)$ is not $\emptyset$), and then the move is performed. Moving the current symbol just

amounts to toggling the 'underline' marking of two adjacent symbols, as we have shown in the above example. Writing is simple if we are not at a boundary symbol; we just have to overwrite the current symbol. However, in the case that we write on an underlined boundary symbol, we have to shift the *entire* tape content on one of the sides in order to append a new boundary symbol to the left or right. Here, we need the arrows over the boundary symbols, in order to know in which direction we have to shift the tape.

7.2.3 Complete Machine

Using the outlined machines ReadCurrentSymbols and DoActions, we can, for each step q, define a machine Step q executing it, found in the appendix [Forster et al. 2019]. The loop function $\mathsf{Loop} : Q_{M_n} \to \mathsf{TM}^1_\Gamma(L)$ and the machine $M_1 : \mathsf{TM}^1_\Gamma$ can now be finally defined:

$$\mathsf{Loop} := \mathsf{MemWhile}\ \mathsf{Step}$$
$$M_1 := \mathsf{Loop}\ init_{M_n}$$

The correctness statement says that whenever M_1 terminates, M_n also terminates given the encoded tapes with the same label as M_1:

Theorem 7.2 (Correctness of M_1)**.**

$$M_1 \vDash \big(\lambda t\ (\ell, t').\ \forall (tps : \mathsf{Tape}^n_\Sigma).\ t[0] \simeq tps \to$$
$$\exists q\ tps'.\ M_n(tps) \triangleright (q, tps') \ \wedge\ t'[0] \simeq tps' \wedge \ell = \mathsf{lab}_{M_n} q\big)$$

Note that, similar to our universal machine, the constant-factor space overhead follows as our tape encoding guarantees that, except for boundary symbols, every symbol on the tapes tps corresponds to symbols on t.

The termination statement says that if M_n terminates, then M_1 will also terminate when given the encoding of the tapes of M_n; and the running time of M_1 can be bounded by a concrete function. We have also shown that this running time function can be bounded by a polynomial.

Theorem 7.3 (Running time of M_1)**.** *There is a constant c s.t.*

$$M_1 \downarrow \big(\lambda t\ k. \exists tps\ k'\ c'.\ M_n(tps) \triangleright^{k'} c' \wedge t \simeq tps$$
$$\wedge\ c \cdot (|tps| + n \cdot k')^2 \cdot k' \le k\big)$$

8 Universal Turing Machine

In this section, we outline a universal Turing machine Univ. Univ expects as input the encoding of a single-tape Turing machine M and a tape t and computes the result of M on t. Tape 0 is treated as a 'raw' tape, containing the tape of the simulated machine, while all other tapes are registers. For instance, tape 1 will contain an encoding of δ_M while tape 2 will contain the encoding of the current state of M. Instead of restricting to binary machines M, we parametrise Univ over the tape alphabet of the machines that can be simulated.

The high-level outline of Univ is as follows: The machine Step first checks whether the current state is a final state and halts if this is the case. Otherwise, it reads the current symbol and looks up the action to be executed from the encoding of δ. After this, it executes the action on the object state and updates the tape for the current state. Ultimately, we will define Univ := While Step.

Encoding of transition functions We encode states as tuples $\mathbb{B} \times \mathbb{N}$. The Boolean encodes whether the state is a halting state. The number is a numerical representation for the machine state.

Transition functions δ are encoded as an 'association list', i.e. a list $\mathcal{L}(A \times B)$, specifically with:

$$A := O(\Sigma_M) \times (\mathbb{B} \times \mathbb{N}) \quad B := (O(\Sigma_M) \times \mathsf{Move}) \times (\mathbb{B} \times \mathbb{N})$$

Σ_δ is defined as the canonical alphabet that is used to encode this list.

We have a family of machines $\mathsf{Lookup} : \mathsf{TM}^5_{\Sigma_{\mathcal{L}(X \times Y)}}$ (parametrised over alphabets Σ_X and Σ_Y), that looks up a value x in an associative list of type $\mathcal{L}(X \times Y)$. This machine is used in order to lookup the pair of the required actions and the successor state according to the transition function δ.

Alphabet For the definition of Univ, we follow a general practice for machines involving complex alphabets, which we demonstrate as an example here. The class of machines $\mathsf{Univ} : \mathsf{TM}^6_{\Sigma^+_{\mathsf{Univ}}}$ is parametrised over an arbitrary alphabet Σ_{Univ}. This alphabet has to 'include' some smaller alphabets that we need for the definition of Univ. In this case, we need an alphabet Σ_δ (for the encoding of δ), and the alphabet Σ_M (the object tape). The 'inclusion' of these alphabets is specified by the user of Univ using retractions into Σ_{Univ}, i.e. $f_\delta : \Sigma_\delta \hookrightarrow \Sigma_{\mathsf{Univ}}$ and $f_M : \Sigma_M \hookrightarrow \Sigma_{\mathsf{Univ}}$.

Outline We have some auxiliary machines that work directly on the object tape. We say that the object tape $t : \mathsf{Tape}_{\Sigma_{\mathsf{Univ}}}$ contains the simulated tape $t_M : \mathsf{Tape}_{\Sigma_M}$ if we have $t = (\lambda s.\ f_M\ s)@t_M$, and write $t \simeq t_M$.[3]

The auxiliary machine $\mathsf{ReadCurrent} : \mathsf{TM}^1_{\Sigma^+_{\mathsf{Univ}}}(O(\Sigma_M))$ reads the current symbol from the object tape by reading *its* current symbol and mapping it to the alphabet of Σ_M using f_M^{-1}. Similarly, the machine DoAction a executes an action $a : O(\Sigma_M) \times \mathsf{Move}$ on the object tape.

We also have auxiliary machines SetFinal b and $\mathsf{IsFinal} : \mathsf{TM}(\mathbb{B})$ that set or get the Boolean in the encoding of a state.

The step machine $\mathsf{Step} : \mathsf{TM}^6_{\Sigma^+_{\mathsf{Univ}}}(O(\mathbb{1}))$ works as follows: It first uses IsFinal to check whether the machine is in a final state. If so, it will immediately terminate in the label $\lfloor() \rfloor$. Otherwise, it will copy the read symbol from the object tape to an internal tape. It will then construct the pair $(cs, \varepsilon(q))$ and uses the machine Lookup to look up the pair $(\varepsilon(q'), a)$ for this combination of read symbols and current state. It destructs the pair, executes the action, and stores the state q' as the current state. Finally, we define Univ := While Step.

[3] Note that we implicitly map $f_M\ s$ to the extended alphabet Σ^+_{Univ}.

Note that if we want an actual machine (for a given alphabet Σ_M), we have to instantiate the retractions f_δ and f_M. For example, we can choose $\Sigma_{\mathsf{Univ}} := \Sigma_\delta$ and $f_\delta := id$; there are multiple options to define f_M (because Σ_M occurred multiple times in Σ_δ).

The correctness theorem states that if Univ contained a configuration c of some machine $M : \mathsf{TM}^1_{\Sigma_M}$, and if Univ terminated, then M also terminates in some number of steps, and the tapes of Univ encode the final configuration of M.

Theorem 8.1 (Correctness of Univ).

$$\mathsf{Univ} \vDash \lambda t\ t'.$$
$$\forall (M : \mathsf{TM}^1_{\Sigma_M})\ (t_M : \mathsf{Tape}_{\Sigma_M})\ (q : Q_M)\ (s_1\ s_2\ s_3\ s_4\ s_5 : \mathbb{N}).$$
$$t[0] \simeq t_M \to t[1] \simeq_{s_1} \delta_M \to t[2] \simeq_{s_2} q \to$$
$$\mathsf{isVoid}_{s_3}(t[3]) \to \mathsf{isVoid}_{s_4}(t[4]) \to \mathsf{isVoid}_{s_5}(t[5]) \to$$
$$\exists (t'_M : \mathsf{Tape}_{\Sigma_M})\ (q' : Q_M).\ M(q, t_M) \triangleright (q', t'_M) \wedge$$
$$t'[0] \simeq t'_M \wedge t'[1] \simeq_{s_1} \delta_M \wedge t[2] \simeq_{(2+|Q_M|+\max c_M\ s_2)} q' \wedge$$
$$\mathsf{isVoid}_{\max c_M\ s_3}(t'[3]) \wedge \mathsf{isVoid}_{\max c_M\ s_4}(t'[4]) \wedge$$
$$\mathsf{isVoid}_{\max c_M\ s_5}(t'[5]), \quad \textit{where } c_M := |\varepsilon(\delta_M)| + 1$$

Here, note that regarding space, we have that every tape (except for the object tape) can be bounded by a constant depending on M. The object tape t_M itself uses exactly the same number of symbols as in the simulated machine M, as it contains the same content.

The running time theorem of Univ states that if M terminates in k steps, then Univ will terminate in a certain number of steps if Univ's input tapes contain the encoding of M's initial configuration. Furthermore, we polynomially bounded the number of steps.

Theorem 8.2 (Running time of Univ). *There is a constant c such that for every alphabet Σ_M:*

$$\mathsf{Univ} \downarrow \lambda t\ k.$$
$$\exists (M : \mathsf{TM}^1_{\Sigma_M})(t_M\ t'_M : \mathsf{Tape}_{\Sigma_M})(q\ q' : Q_M)(k' : \mathbb{N}).$$
$$t[0] \simeq t_M \wedge t[1] \simeq \delta_M \wedge t[2] \simeq q_M \wedge$$
$$\mathsf{isVoid}(t[3]) \wedge \mathsf{isVoid}(t[4]) \wedge \mathsf{isVoid}(t[5]) \wedge$$
$$M(q, t_M) \triangleright^{k'} (q', t'_M) \wedge c \cdot (1 + k') \cdot |\varepsilon(\delta)| \cdot |Q_M| \leq k$$

We summarise the above theorems:

Theorem 8.3. Univ *is a universal Turing machine that simulates single-tape machines with constant space overhead and linear running time overhead.*

Using the compiler from section 7 we can compile Univ to a class of single-tape universal Turing machines U (parametrised over an alphabet Σ_M) simulating other single-tape machines. Furthermore, if we see the compiler as an encoding for n-tape machines, we can define a class of single-tape universal Turing machines MU (again parametrised over an alphabet Σ_M) simulating multi-tape Turing machines.

Table 1. Line counts (grouped, counted with coqwc)

Component		Spec	Proof
Libraries (finite types, retractions, etc.)		2157	2716
L_0&L_1	Semantics of (labelled) TMs	644	288
	Primitive Machines	178	48
L_2	Lifts	367	195
	Combinators	561	541
	Simple machines	598	432
L_3	Encodable types	282	179
	Tape-containment	171	43
	(De-) constructors	501	511
	CopyValue, Reset, Compare	538	548
	Refinement on alphabet-lift	139	93
Ltac automation (for L_2 and L_3)		156	15
Complexity	Time (infrastructure and machines)	268	279
	Space (infrastructure and machines)	259	193
Add and Mult		222	276
M_1	Encoding	76	65
	Implementation	783	1186
	Time bounds	236	371
Univ	Lookup in association list	169	120
	Implementation	324	333
	Time bounds	146	212
	Space bounds	240	300
MU (single-tape Univ for multi-tape machines)		408	637
Total (excl. libraries)		7327	6957

9 Mechanisation in Coq

The development covered in this paper consists of ca. 19100 lines of code. Of that, ca. 4800 lines are various extensions of the standard library. The development of L_0 and L_1 take together 1158 lines, L_2 2694 lines, and L_3 3005 lines. The implementation of addition and multiplication takes 496 lines, whereas the multi-tape to single-tape compiler takes 2717 lines, and the universal machine takes 1844 lines. In total, we have a ratio of 51% to 49% specification vs. proofs (excluding libraries). We developed ca. 200 lines of new Ltac tactics. The development takes about 5 minutes to compile on an Intel(R) Core(TM) i7-6600U CPU @ 2.60GHz machine.

Our project started in October 2016 and needed several re-implementations to serve as a basis for feasible implementations covering both time- and space-complexity. We had an almost finished implementation of levels L_0-L_3 not covering termination or any complexity analyses early on. This implementation however did parameterise every machine w.r.t. both tapes and an alphabet, which lead to very high compile-times and finally to unsolvable problems with deconstructor machines on level L_3.

We tried to keep the paper presentation faithful to the mechanisation. There are however several minor differences. For instance, we do not use the implicit coercion from machines to unit-labelled machines in Coq and always write

$\mathsf{TM}^n_\Sigma(\mathbb{1})$ explicitly. In some cases, it made sense to split type classes to decrease the size of proof terms. For instance, we do not capture injectivity in the type class for encodable types, because it is never technically needed, but only prove it as a lemma. More notably, we define tapes over arbitrary types X instead of finite types Σ for all definitions not making use of the finiteness. The definition of Turing machines then instantiates the alphabet with a finite type. This change had noticeable impact on compilation times: In dependent type theory, equalities take the type over which they are defined as argument. Finiteness proofs can be big, and rewriting with bigger proof terms can take longer time, making our change result in a significant speed-up since we rewrite with tape-equalities a lot.

Finite types in general proved to be a hurdle. We relied on the implementation by Menz [2016] in Coq. However, using a more mature implementation like that of `mathcomp` would have been more helpful, since we found ourselves going back to changing the implementation of finite types more than once. At the start of our project, the Coq ecosystem was not as mature as it is today, and the installation of third-party libraries was sometimes a hassle, which prevented us from using `mathcomp`.[4] The ease of installing new libraries using `opam` would thus have spared us lots of work.

For realisation proofs, we developed a considerable amount of automation. Crucial for this were three tools: First, Sigurd Schneider's `smpl` plugin,[5] which allows automated forward reasoning and simplification and is extendable by hints. Second, Coq's tactic language `Ltac`, with the ability to introduce existential variables (so called `evars`). And third, Coq's setoid rewriting facilities [Sozeau 2010]. In total, we are sure that implementing a project like ours in a proof assistant without tactics is infeasible.

The most advanced simplification tactic we use is called `TMSimp`. It destructs complex assumptions and simplifies the goal by exhaustively rewriting with all available equations between tapes. The tape equations can be considered the main antagonist in terms of usability and compile time in our framework, using them by hand is almost impossible, and the introduction of `TMSimp` eased our work on all levels.

10 Related Work

To the best of our knowledge, there are three mature mechanisations of Turing machines in the literature.

Ciaffaglione [2016] formalises the undecidability of the halting problem for Turing machines in Coq. They use a coinductive encoding of tapes. Since only a finite part of tapes is used in any computation, we were happy with our inductive characterisation.

Xu, Zhang, and Urban [2013] formalise Turing machines, Abacus machines and μ-recursive functions. They obtain a universal Turing machine by formalising a universal μ-recursive function and verified compilers from μ-recursive functions to Abacus machines and on to Turing machines. Abacus machines are a simple form of register machines which allow incrementation, decrementation and an unconditional jump operation. Xu et al. also verify that the halting problem of Turing machines is undecidable.

Asperti and Ricciotti [2012, 2015] formalise single- and multi-tape Turing machines over arbitrary finite alphabets in Matita [Asperti et al. 2011], an interactive theorem prover based on the same constructive type theory as Coq. They define a universal Turing machine and show the functional correctness and termination of their machine.

We build upon their work: We use almost the same definition of Turing machines in constructive type theory, and especially their representation of tapes as an inductive type is very useful, as it is canonical and abstracts away from the notion of blank cells used commonly on paper. We build upon their notion of realisability that specifies the semantics of a machine in terms of the relation that holds between the input and the output tape, and upon their idea of composing machines by using shallowly-embedded combinators.

We see four main improvements in our work:

First, we introduce the possibility to analyse time and space complexity of machines. Thus, both our universal machine and the compiler from multi-tape to single-tape machines have stronger specifications. In principle, since space complexity is a correctness property (for the definition of tapes used in their and our work), analysing space would have already been possible in Asperti and Ricciotti's approach, but is not considered.

Secondly, our notions of encoding and containment allow constructions of Turing machines reading like programs in a while-language that supports (a finite number of) registers containing values of inductive types. This simplified the construction of our universal Turing machine considerably.

Thirdly, Asperti and Ricciotti use *concrete* final states of the *implicitly* constructed machines to handle the control flow in the 'if' and 'while' combinator. This breaks the abstractions their realisability framework provides, thus they define a second version of realisability that mentions a concrete 'accepting' state. Because we want to abstract away from the state space of machines, we introduce labelled Turing machines, with a labelling function mapping (final) states to the elements of an arbitrary type. Compositions of machines can now be specified w.r.t. their labels, and we never have to mention states.

Fourthly, Asperti and Ricciotti usually construct machines parametrically in the alphabet used. Similarly, the number of tapes of the constructed machines is a parameter. This means that when composing machines, each composed machine has to work with the same alphabet and the same number of tapes as the resulting machine. Therefore, a lot of basic machines are parametric in the number of tapes and the index

[4] In retrospect, we should have gone through the hassle.
[5] https://github.com/sigurdschneider/smpl

of the different used tapes, which need to be given explicitly. We use the notion of liftings to extend tapes and the alphabet. The tape-lift significantly simplifies both the definition and verification of machines, whereas for the alphabet we think that the two approaches don't differ considerably.

To our knowledge, we presented the first completely axiom-free mechanisation of a multi-tape-to-single-tape compiler. There is a conceptually different, but incomplete, mechanisation of such a compiler in Matita's standard library.

Concerning other sequential models of computations, we are aware of the formalisation of register machines by Forster and Larchey-Wendling [2019] in Coq, which are proven equivalent to Diophantine equations [Larchey-Wendling and Forster 2019]. We believe that showing the equivalence of their register machines and our Turing machines would be relatively straightforward. Bayer et al. [2019] formalise a variant of register machines for their work-in-progress proof of the undecidability of Hilbert's tenth problem in Isabelle.

There are various other formalisation of other models of computation. Norrish [2011] formalises the full λ-calculus in HOL light and proves the undecidability of the halting problem, besides various other computability-theoretic results. In a similar spirit, Forster and Smolka [2017] formalise the weak call-by-value λ-calculus in Coq and also formalise basic computability theory. Carneiro [2018] formalises μ-recursive functions and similar basic computability results in Lean. Finally, Ramos et al. [2018] formalise the undecidability of the halting problem of a PVS-like functional language in PVS.

11 Discussion and Future Work

Asperti and Ricciotti [2015] set out with the goal to formalise complexity theory in a proof assistant, and accomplished to verify the correctness of a universal machine. We have gotten some steps closer to formalising actual complexity theoretic results by extending their results with time and space complexity, but are still nowhere near making formalisations of complexity theory feasible. With the current state of the art, neither simple examples nor interesting complexity theoretic theorems are in reach, much less results requiring oracles or nondeterminism. We are convinced that this is not due to the lack of powerful verification tools for Turing machinesbut rather because Turing machines as model of computation are inherently infeasible for the formalisation of any computability or complexity theoretic result.

The most promising approach to formalising complexity theory in our eyes is to carry out formalised complexity theory in the λ-calculus, following formalisations of computability theory (e.g. by Forster and Smolka [2017]). For the most satisfying results, a complexity-theoretic equivalence between Turing machines and some form of λ-calculus should then be formalised, allowing results proved in the λ-calculus to be formally translated back to Turing machines. Forster and Kunze [2019] show in Coq that the weak call-by-value λ-calculus L can simulate Turing machines with a linear time overhead, building on our definition of machines. Their translation also is linear in space (although that fact is not made formal in Coq). Similarly, Forster et al. [2020a] show that Turing machines can simulate L with polynomial time and constant factor space overhead, using a rather involved simulation technique. In the current state of our framework, formalising this technique would take very long and would be at least daunting, but probably entirely infeasible.

What we have accomplished is to use the combinators from layer L_2 and the register operations from L_3 to obtain a relatively easy and explainable verification of a universal Turing machine w.r.t. to the notions of correctness and termination defined on layer L_1. The multi-tape to single-tape compiler explained in Section 7 did not make use of L_3.

For these large programs, proofs oftentimes became hard to navigate: The relations our framework derives follow the structure of the program. This is helpful in getting an intuition for proving these relations equivalent to shorter, problem-focused relations, but the equivalence proofs can become huge and involve a lot of tedious reasoning.

Coq's kernel takes long to elaborate the verification of large machines like Univ. One problem is that the tape-lifting operator introduces equations between tapes, and the automation that tries to rewrite with them everywhere has bad asymptotic complexity. The equations cannot be simply removed, because they may still be needed in some later part of the proof. Furthermore, termination proofs have a similar structure as realisation proofs, but are too different to make generalisations or copy-and-paste approaches impossible.

Xu et al. [2013] employ a very low-level Hoare logic to verify their machines. For low-level verification, we actually found our realisation framework to be quite well-suited. In future work, we want to investigate a Hoare-logic with native support for realisation, termination, asymptotic time- and space complexity as well as value-containment. We have hope that this would make a simulation of the λ-calculus in Turing machines feasible.

We contribute our formalisation to the library of undecidable problems in Coq [Forster et al. 2020b]. Note that Section 7 contains a synthetic many-one reduction reduction from the halting problem of multi-tape Turing machines to the halting problem of single-tape machines. To connect the halting problem of single-tape Turing machines to other problems in the library, we translated the halting problem for machines $M : \mathsf{TM}^1_\Sigma$ to the formalisation of single-tape machines used by Forster, Heiter, and Smolka [2018], thereby obtaining a reduction from halting to the Post correspondence problem.

Acknowledgments

We would like to thank Dominik Kirst and Gert Smolka for very helpful feedback on the presentation and the anonymous reviewers for their useful hints.

References

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

Andrea Asperti and Wilmer Ricciotti. 2015. A formalization of multi-tape Turing machines. *Theoretical Computer Science* 603 (2015), 23–42.

Andrea Asperti, Wilmer Ricciotti, Claudio Sacerdoti Coen, and Enrico Tassi. 2011. The Matita Interactive Theorem Prover. In *International Conference on Automated Deduction*. Springer, 64–69.

Jonas Bayer, Marco David, Abhik Pal, Benedikt Stock, and Dierk Schleicher. 2019. The DPRM Theorem in Isabelle (Short Paper). In *10th International Conference on Interactive Theorem Proving (ITP 2019)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

Mario Carneiro. 2018. Formalizing computability theory via partial recursive functions. *arXiv preprint arXiv:1810.08380* (2018).

Alberto Ciaffaglione. 2016. Towards Turing computability via coinduction. *Science of Computer Programming* 126 (2016), 31–51.

Yannick Forster, Edith Heiter, and Gert Smolka. 2018. Verification of PCP-related computational reductions in Coq. In *International Conference on Interactive Theorem Proving*. Springer, 253–269.

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

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

Yannick Forster, Fabian Kunze, and Maximilian Wuttke. 2019. Appendix for Verified Programming of Turing Machines in Coq. https://www.ps.uni-saarland.de/extras/tm-verification-framework/cpp20-tms-appendix.pdf

Yannick Forster and Dominique Larchey-Wendling. 2019. 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*. ACM, 104–117.

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

Yannick Forster and Gert Smolka. 2017. Weak Call-by-Value Lambda Calculus as a Model of Computation in Coq. In *Interactive Theorem Proving - 8th International Conference, ITP 2017, Brasilia, Brazil, September 26-29, 2017*.

Dominique Larchey-Wendling and Yannick Forster. 2019. Hilbert's Tenth Problem in Coq. In *4th International Conference on Formal Structures for Computation and Deduction, FSCD 2019, Dortmund, Germany*. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 27:1–27:20.

Jan Christian Menz. 2016. *A Coq Library for Finite Types*. Bachelor's thesis. Saarland University. https://www.ps.uni-saarland.de/~menz/bachelor.php

Michael Norrish. 2011. Mechanised Computability Theory. In *Interactive Theorem Proving*, Marko van Eekelen, Herman Geuvers, Julien Schmaltz, and Freek Wiedijk (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 297–311.

Thiago Mendonça Ferreira Ramos, César Muñoz, Mauricio Ayala-Rincón, Mariano Moscato, Aaron Dutle, and Anthony Narkawicz. 2018. Formalization of the Undecidability of the Halting Problem for a Functional Language. In *International Workshop on Logic, Language, Information, and Computation*. Springer, 196–209.

Michael Sipser. 2006. *Introduction to the Theory of Computation*. Vol. 2. Thomson Course Technology Boston.

Matthieu Sozeau. 2010. A new look at generalized rewriting in type theory. *Journal of formalized reasoning* 2, 1 (2010), 41–62.

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