SAARLAND UNIVERSITY

FACULTY OF MATHEMATICS AND COMPUTER SCIENCE

BACHELOR'S THESIS

VERIFIED PROGRAMMING OF TURING MACHINES IN COQ

Author
Maximilian Wuttke

Advisor
Yannick Forster

Supervisor
Prof. Dr. Gert Smolka

Reviewers
Prof. Dr. Gert Smolka
Prof. Dr. Holger Hermanns

Submitted: 01th September 2018

Eidesstattliche Erklärung

Ich erkläre hiermit an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Statement in Lieu of an Oath

I hereby confirm that I have written this thesis on my own and that I have not used any other media or materials than the ones referred to in this thesis.

Einverständniserklärung

Ich bin damit einverstanden, dass meine (bestandene) Arbeit in beiden Versionen in die Bibliothek der Informatik aufgenommen und damit veröffentlicht wird.

Declaration of Consent

I agree to make both versions of my thesis (with a passing grade) accessible to the public by having them added to the library of the Computer Science Department.

Saarbrücken, 01th September, 2018

Abstract

Turing machines build the traditional foundation of the theory of computation and complexity. However, concrete Turing machines are often only sketched out. Even if authors define the complete machine, they leave the invariants to figure out for the reader. Moreover, it is common to employ the *Church-Turing thesis*, to (informally) conclude that a function is Turing-computable. Reasons for that are manifold. Turing machines are very low-level, because the operations on tapes are primitive. They are also non-compositional; control-flow operators like sequential composition and loops are not available. Reasoning about invariants and concrete machine states is tedious, because the execution of the machine could proceed from one state of the machine to any other state; the set of states may also be huge for complex machines.

In this thesis, we fill these gaps. We present a framework developed in the theorem prover Coq, in that we can define, specify, and formally verify multi-tape Turing machines. The framework eases programming and verification of Turing machines, because it provides abstractions like values and control-flow operators. We showcase the power of this framework by programming and verifying a multi-tape Turing machine that simulates a two-stack machine for the call-by-value λ-calculus.

Acknowledgements

I feel obliged to say special words of thank to Yannick Forster, who advised this thesis and the development of the framework presented in this thesis. In many weekly meetings, over the course of one year, we discussed how the framework could look like. I am thankful that he provided a LaTeX template for this thesis and that he proof read my texts many times.

I want to thank my supervisor Prof. Smolka, who introduced me to the field of functional programming and computational logic. I always appreciated his passion in his lectures. Without him, I would not have found the field of computational logic and formal verification. I also recognise the help of Kathrin Stark, who helped me to get into this topic, in particular because I missed the first three weeks of *Introduction to Computational Logic*. I want to thank Prof. Smolka for giving me the opportunity to write my thesis at his chair. I really appreciate the atmosphere together with motivated PhD students, especially with Kathrin Stark, Fabian Kunze, and Yannick Forster.

I express gratitude to Prof. Hermanns, who mentored me in the computer science undergraduate Honours Program of Saarland University. I am glad that he is the second reviewer of this thesis.

Last but not least, I want to thank my parents for supporting me in every imaginable way. Without them, this thesis would not exist, me neither.

Contents

Chapter 1

Introduction

Although Turing machines are a simple (but not quite simplistic) model of computation, there are not many *rigorous* proofs about Turing machines in the literature. We think that the following points are reasons for that. First, their semantics is *unstructured*: from each state of the machine, the execution can proceed in every other state, similar to the infamous *goto* statement [7]. But even worse, Turing machines are not *compositional*. Sequential composition or loops of Turing machines are not *per se* available. Even the formal specification of machines is a burden, because complex machines may have a huge number of internal states. Last but not least, they are *low-level*, because the operation on tapes are primitive: read a symbol from the tape, write a symbol to the tape, or move the (read/write) head in a direction.

For the above reasons, textbooks like Boolos et al. [4] leave out detailed proofs of correctness. They also often only give an informal description of machines, which obviously makes formal reasoning impossible. Even if they define the whole machine, they leave out formal specifications of invariants to figure out for the reader. To establish that a function is Turing computable, authors often give an informal description of the algorithm and conclude, using the *Church-Turing thesis*, that the function is Turing-computable. Or they switch to another abstract machine model, but define the compilation function between the models of computation only informally.

In this thesis, we aim to define, specify, and formally verify Turing machines in a framework built in the theorem prover Coq [16]. First of all, we address the problems above. Instead of defining machines in terms of transition tables, we compose machines using functions of Coq's dependent type theory – the *Calculus of (Co)Inductive Constructions* (also known as CIC). For example, we define a function that builds the sequential composition of two machines. To eliminate the need to reason about concrete machine states, we give all states a label (e.g. true or false) and only have to reason about these labels. The number of labels is always reasonable small, compared to the potential huge amount of states. We address the prob-

lem that machines are low-level, by introducing abstractions, so that we can define Turing machines that directly manipulate values of arbitrary encodable types. This gives the advantages of register machines, but we are not restricted to natural numbers.

There are many variants of Turing machines. All variants can be shown to be computationally equivalent. In this thesis, we choose multi-tape Turing machines with arbitrary finite alphabets. Our plan is that each tape should contain a value. We choose a finite model of tapes. This means that each tape has only finitely (but arbitrarily) many symbols.

1.1 Contributions

We formalise a variant of deterministic multi-tape Turing machines in the interactive theorem prover Coq. We build a framework for programming and formally verifying correctness and time complexity of Turing machines. Our framework extends the framework by Asperti and Ricciotti [2] in the interactive theorem prover Matita [3]. Compared to their framework, we eliminate the need to reason about concrete machine states and introduce more general control-flow operators. We increase the level of programming abstraction and make it possible that Turing machines can directly manipulate values of arbitrary encodable types. We show that our framework is strong enough to implement and verify a Turing machine that simulates a two-stack machine for a variant of the λ-calculus. We formally prove that the halting problem of this abstract machine reduces to the halting problem of multi-tape Turing machines. Thereby, this work is the last step to formally prove that multi-tape Turing machines can simulate the λ-calculus.

1.2 Related Work

Asperti and Ricciotti [1] formalise single-tape Turing machines over arbitrary finite alphabets in the interactive theorem prover Matita. Matita uses the same constructive type theoretic foundation as Coq. In [2], they formalise multi-tape Turing machines in Matita. They introduce the notion of *realisation* for specifying the semantics of concrete Turing machines. They define and verify a universal Turing machine and also formalise the reduction from multi-tape Turing machines to single-tape Turing machines. Furthermore, they propose the formalisation of Turing machines as a benchmark for comparing proof assistants.

Xu, Zhang, and Urban [17] formalise single-tape Turing machines over a binary alphabet in Isabelle/HOL. They follow the textbook of Boolos et al. [4] and use Hoare-logic to specify the semantics of concrete Turing machines. They implement formally verified translation functions from *abacus programs* and *partial recursive*

functions to Turing machines and prove the undecidability of the halting problem of Turing machines.

Ciaffaglione et al. [6] define tapes of Turing machines as infinite streams. They verify Turing machines using induction and co-induction and also show the undecidability of the halting problem in Coq.

Forster, Heiter, and Smolka [12] formally reduce the halting problem of single-tape Turing machines to the *Post correspondence problem* (PCP) in Coq. They use the same definition of Turing machines as we use, but restricted to one tape. This definition of single-tape Turing machines was originally presented in [1].

There are other mechanisations of abstract machine models. For example, Forster and Smolka [10] formalise the theory of computation in Coq, based on the language L, which is also known as the (weak) call-by-value λ-calculus. Norrish [15] formalises computability theory in HOL4. He considers a variant of the λ-calculus and recursive functions, and show that both models of computation are computationally equivalent. Kunze et al. [13] formalise reductions from the programming language L to several stack-machines. The stack-machine for that we build a simulator is a variant of a machine of this paper.

1.3 Outline

In Chapter 2, we define the notion of multi-tape Turing machines. We also introduce means to specify the semantics of concrete machines. In Chapter 3, we define primitive machines, on which all our machines are based. In Chapter 4, we define control-flow operators. In Chapter 5, we show how to combine machines with different alphabets and numbers of tapes. We build simple machines in Chapter 6. In Chapter 7, we introduce abstractions that enable the programmer to directly manipulate values, and we show complex case-studies. In Chapter 8, we develop our final case-study where we implement and verify a Turing machine that simulates a two-stack machine for L and show that the halting problem of this machine reduces to the halting problem of multi-tape Turing machines. We conclude and discuss possible future work in Chapter 9. In the appendix, we present pearls of the Coq development of this thesis.

Throughout the thesis, we use mathematical notation, the reader is not required to be expert in type theory. In the PDF version of this thesis, all definitions and lemmas are hyperlinked to the documented online source code of the Coq implementation. The source code is tested to compile with Coq versions 8.7 and 8.8. The home page of this thesis contains the PDF version, the source code, and online documentation:

`https://www.ps.uni-saarland.de/~wuttke/bachelor/`

Chapter 2

Definitions

In this chapter, we formally define multi-tape Turing machines. We take the definition of multi-tape Turing machines and their tapes from Asperti and Ricciotti [2]. We introduce notions for specifying correctness and time complexity of machines, where the former is also based on [2].

2.1 Preliminary Definitions

2.1.1 Notational Conventions

The symbols 1, $\mathbb{B}$, $\mathbb{N}$, $X \times Y$, $X + Y$, $\mathcal{O}(X)$, and $\mathcal{L}(X)$ stand for the well-known standard types. $\mathbb{T}$ stands for the type of types and $\mathbb{P}$ for the type of propositions. The unit element $()$ is the only element of 1. $\sum_{a:A} B(a)$ denotes sigma types, i.e. dependent pairs, with the projections π_1 and π_2. We write (a, b) for elements of sigma types. For (named) tuples $A = (a : X,\ b : Y,\ c : Z)$, we use subscripts, i.e. a_A, for the projections. We use the symbols $\emptyset$ and $\lfloor x \rfloor$ as elements of the type $\mathcal{O}(X)$. $\mathbb{F}_n := \{0, \ldots, n-1\}$ is the type with n elements. We use indices $i : \mathbb{F}_n$ for vector-access $x[i]$ with $x : X^n$, where X^n denotes the type of vectors over X of size n. We usually leave subscripts out if they are clear from the context.

2.1.2 Relations

We define the semantics of concrete Turing machines in terms of relations. We write $R \subseteq A \times B$ as a notation for $R : A \to B \to \mathbb{P}$. We call relations of the form $R \subseteq A \times (B \times A)$ **labelled relations** (labelled over B) and write $R \subseteq A \times B \times A$. We identify unit-labelled relations $R \subseteq A \times 1 \times A$ with binary relations $R \subseteq A \times A$. We use λ-notation to define relations.

We use the following standard relational operators:

Definition 2.1 (Relational operators) *Let* $R, S \subseteq A \times B$ *and* $T \subseteq B \times C$.

$$R \cap S := \lambda x\ y.\ R\ x\ y \wedge S\ x\ y$$
$$R \cup S := \lambda x\ y.\ R\ x\ y \vee S\ x\ y$$
$$R \circ T := \lambda x\ z.\ \exists y.\ R\ x\ y \wedge T\ y\ z$$

Note that if we compose a binary relation $R \subseteq A \times A$ with a labelled relation $S \subseteq A \times B \times A$, we get a labelled relation $R \circ S \subseteq A \times B \times A$.

We use $\bigcup_{c:C} R(c)$ as a notation for $\lambda a\ b.\ \exists c.\ R(c)\ a\ b$. We also define the reflexive transitive closure of binary relations, also known as relational Kleene star:

Definition 2.2 (Kleene star) *Let* $R \subseteq A \times A$. *The relation* R^* *is defined inductively:*

$$\frac{}{R^*\ x\ x} \qquad \frac{R\ x\ y \qquad R^*\ y\ z}{R^*\ x\ z}$$

The relational power operator composes a relation k times.

Definition 2.3 (Relational power) *Let* $R \subseteq A \times A$. *The relation* R^k *is defined inductively:*

$$\frac{}{R^0\ x\ x} \qquad \frac{R\ x\ y \qquad R^k\ y\ z}{R^{(S\ k)}\ x\ z}$$

We have an operator that restricts the label B of a labelled relation and yields a binary relation:

Definition 2.4 (Relational restriction) *Let* $R \subseteq A \times B \times A$ *and* $y : B$.

$$R|_y := \lambda x\ z.\ R\ x\ (y, z)$$

Similarly, we can define an operator that takes a binary relation and yields a labelled relation where we fix the label.

Definition 2.5 (Relational fix) *Let* $R \subseteq A \times A$ *and* $y : B$.

$$R\|_y := \lambda x\ (y', z).\ R\ x\ z \wedge y' = y$$

Definition 2.6 (Relational inclusion and equivalence) *Let* $R, S \subseteq A \times A$.

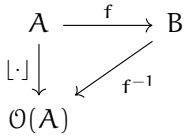
$$R \subseteq S := \forall x\ y.\ R\ x\ y \to S\ x\ y$$
$$R \equiv S := R \subseteq S\ \wedge\ S \subseteq R$$

2.1.3 Retractions

Retractions are a natural way to formalise injections f together with their partial inversion function f^{-1}.

Definition 2.7 (Retraction) *Let* $A, B : \mathbb{T}$. *A pair of functions* $f : A \to B$, $f^{-1} : B \to \mathcal{O}(A)$ *is called a retraction from* A *to* B, *if* $\forall x\ y.\ f^{-1}(y) = \lfloor x \rfloor \leftrightarrow y = f(x)$.

The direction from right to left of Definition 2.7 means that f^{-1} inverses f. It is equivalent to the following commutative diagram:



The direction from left to right of Definition 2.7 means that f^{-1} only maps values back that are in the image of f. This property is called *tightness*.

We write $f : A \hookrightarrow B$ when we assume that the pair (f, f^{-1}) is a retraction. Note that this notation introduces two names for functions ($f : A \to B$ and $f^{-1} : B \to \mathcal{O}(A)$), but mostly we use the name f for the *pair* of both functions.

Lemma 2.8 (Basic properties of retractions) *Let* $f : A \hookrightarrow B$.

1. $\forall (x : A).\ f^{-1}(f(x)) = \lfloor x \rfloor$
2. $\forall (y : B).\ f^{-1}(y) = \emptyset \to \forall (x : A).\ f(x) \neq y$
3. $f : A \to B$ *is injective, i.e.* $\forall x\ y.\ f(x) = f(y) \to x = y$
4. $\forall (x\ y : A).\ f^{-1}(f(x)) = \lfloor y \rfloor \to x = y$

Proof Claim 1 and 2 are direct consequences of Definition 2.7. Claim 4 follows by Claim 3.

Proof of Claim 3. Let $x, y : A$ and $f(x) = f(y)$. We have to show $x = y$. It is enough to show $\lfloor x \rfloor = \lfloor y \rfloor$. By Claim 1, we know $\lfloor x \rfloor = f^{-1}(f(x))$ and $\lfloor y \rfloor = f^{-1}(f(y))$. Therefore, it is enough to show that $f^{-1}(f(x)) = f^{-1}(f(y))$. This is trivial because we assumed $f(x) = f(y)$. □

Definition 2.9 (Basic retractions) *Let* A *and* B *be types. We define the retractions*

$\mathtt{RetrId} : A \hookrightarrow A$, $\mathtt{RetrLft} : A \hookrightarrow A + B$, *and* $\mathtt{RetrRgt} : B \hookrightarrow A + B$*:*

$$\mathtt{RetrId}(x) := x \qquad \mathtt{RetrId}^{-1}(x) := \lfloor x \rfloor$$

$$\mathtt{RetrLft}(x) := \mathsf{inl}\, x \qquad \mathtt{RetrLft}^{-1}(z) := \begin{cases} \lfloor x \rfloor & z = \mathsf{inl}\, x \\ \emptyset & z = \mathsf{inr}\, y \end{cases}$$

$$\mathtt{RetrRgt}(y) := \mathsf{inr}\, y \qquad \mathtt{RetrRgt}^{-1}(z) := \begin{cases} \emptyset & z = \mathsf{inl}\, x \\ \lfloor y \rfloor & z = \mathsf{inr}\, y \end{cases}$$

Definition 2.10 (Composition of retractions) *Let* $f : A \hookrightarrow B$ *and* $g : B \hookrightarrow C$*. Then* $g \circ f : A \hookrightarrow C$ *is defined as the following retraction:*

$$(g \circ f) \quad (a) := g(f(a))$$

$$(g \circ f)^{-1}(c) := \begin{cases} f^{-1}(b) & g^{-1}(c) = \lfloor b \rfloor \\ \emptyset & g^{-1}(c) = \emptyset \end{cases}$$

2.2 Machines and Tapes

We use the definition of multi-tape Turing machines, their tapes, and semantics from Asperti and Ricciotti [2].[1]

Definition 2.11 (Multi-tape Turing machine) *An* n*-tape Turing machine over a finite alphabet* Σ *is a tuple* $M = (Q, \delta, \mathtt{start}, \mathtt{halt})$ *where*

- Q *is a finite type*
- $\delta : Q \times (\mathcal{O}(\Sigma))^n \to Q \times (\mathcal{O}(\Sigma) \times \mathsf{Move})^n$
- $\mathtt{start} : Q$
- $\mathtt{halt} : Q \to \mathbb{B}$

There are three possible movements: $\mathsf{Move} ::= \mathsf{L} \mid \mathsf{R} \mid \mathsf{N}$.

We write TM^n_Σ for the type of n-tape Turing machines over the alphabet Σ.

While we parametrise Definition 2.11 over the alphabet Σ and the number of tapes n, we abstract the finite type Q of states inside the type of Turing machines. The transition function δ yields for every state and vector of n read symbols the new state and for every tape an optional symbols to write and a direction to move. The read symbols are also optional, since it can be the case that there is no symbol under the head of a tape. $\mathtt{start}$ is the start state of the machine and $\mathtt{halt}$ represents the

[1] Asperti and Ricciotti [2] restrict machines to have $n > 0$ tapes. We do not have this restriction. We actually define a 0-tape machine Null, see Section 3.1.

subset of halting states. Tuples of the type $\mathcal{O}(\Sigma) \times \mathsf{Move}$ are called *actions*. They are referred to with the symbol Act_Σ or Act if Σ is clear. Our machines behave deterministically, because δ is a function.

When we want to verify complex machines, we do not want to reason about *concrete machine states*, because the number of states of a machine could be huge. Reasoning about *all* states of a machine is thus unfeasible. We still need to reason about states, but we do not need to distinct most of the states. We introduce the notion of **labelled machines**, where we assign a *label* to every state. A label type could be, for example, the type $\mathbb{B}$. Then we only distinct between two kinds of states. If L is a finite type, then $M = (M', \mathtt{lab})$ is a labelled machine, where M' is an unlabelled machine and $\mathtt{lab} : Q_{M'} \to L$ is called the *labelling function* of M. The value of $\mathtt{lab}$ is irrelevant for non-terminating states. The function $\mathtt{lab}$ partitions the terminating states. This means that all terminating states with the same label can be seen semantically equivalent. We write $\mathsf{TM}^n_\Sigma(L)$ for the type of labelled machines over L.[2] We identify unit-labelled machines $\mathsf{TM}^n_\Sigma(1)$ with unlabelled machines TM^n_Σ. We use the symbol M for both labelled and unlabelled machines TM^n_Σ. It should however be always clear from the context, whether M is a labelled or unlabelled machine.

On a **tape**, arbitrarily much memory can be allocated. However, every tape has only finitely many symbols, i.e. there is a left-most and a right-most symbol. A tape essentially is a triple $(\mathtt{ls}, \mathtt{m}, \mathtt{rs})$, where the symbol $\mathtt{m}$ is the symbol under which the (read/write) head of tape is. It is essential that the symbol lists ($\mathtt{ls}$ and $\mathtt{rs}$) are ordered such that the head of the list is the symbol next to the symbol $\mathtt{m}$. When we think of tapes as a finite sequence of symbols from left to right, this means that $\mathtt{ls}$ is stored in reversed order.

There are three cases where there is no current symbol: the tape can be completely empty, or the head can be to the left (or right) outermost of a non-empty tape. Formally, tapes are defined inductively:

Definition 2.12 (Tape) *Let* $\Sigma : \mathbb{T}$. *Then* Tape_Σ *is defined as the inductive type:*

$$\begin{aligned}\mathsf{Tape}_\Sigma ::= {} & \mathsf{niltape} \\ & \mid \mathsf{leftof}\ (\mathtt{r} : \Sigma)\ (\mathtt{rs} : \mathcal{L}(\Sigma)) \\ & \mid \mathsf{midtape}\ (\mathtt{ls} : \mathcal{L}(\Sigma))\ (\mathtt{m} : \Sigma)\ (\mathtt{rs} : \mathcal{L}(\Sigma)) \\ & \mid \mathsf{rightof}\ (\mathtt{l} : \Sigma)\ (\mathtt{ls} : \mathcal{L}(\Sigma))\end{aligned}$$

Recall that we leave the subscript Σ out, if it is clear from the context.

[2] Formally, the type is defined as a sigma type: $\mathsf{TM}^n_\Sigma(L) := \sum_{M' : \mathsf{TM}^n_\Sigma} (Q_{M'} \to L)$. We use the projection π_1 implicitly and write $\mathtt{lab}_M$ for $\pi_2(M)$.

Now we can define the **configuration** of a multi-tape Turing machine. It is captured by the current state and the vector of the n tapes:

Definition 2.13 (Configuration) *A configuration of* $M : \mathsf{TM}^n_\Sigma$ *is a tuple* $c = (q, t)$*, where* $q : Q_M$ *and* $t : \mathsf{Tape}^n_\Sigma$*. We write* $\mathsf{Conf}_M := Q_M \times \mathsf{Tape}^n_\Sigma$ *for the type of configurations of* M*.*

The function $\mathsf{mv} : \mathsf{Move} \to \mathsf{Tape}_\Sigma \to \mathsf{Tape}_\Sigma$ moves a tape in a direction.

Definition 2.14 (Tape movement)

$$
\begin{array}{llll}
\mathsf{mv\ L}\ (\mathsf{leftof}\ r\ rs) & := \mathsf{leftof}\ r\ rs & \mathsf{mv\ R}\ (\mathsf{leftof}\ r\ rs) & := \mathsf{midtape}\ \mathsf{nil}\ r\ rs \\
\mathsf{mv\ L}\ (\mathsf{midtape}\ \mathsf{nil}\ m\ rs) & := \mathsf{leftof}\ m\ rs & \mathsf{mv\ R}\ (\mathsf{midtape}\ ls\ m\ \mathsf{nil}) & := \mathsf{rightof}\ m\ ls \\
\mathsf{mv\ L}\ (\mathsf{midtape}\ (l :: ls)\ m\ rs) & := \mathsf{midtape}\ ls\ l\ (m :: rs) & \mathsf{mv\ R}\ (\mathsf{midtape}\ ls\ m\ (r :: rs)) & := \mathsf{midtape}\ (m :: ls)\ r\ rs \\
\mathsf{mv\ L}\ (\mathsf{rightof}\ l\ ls) & := \mathsf{midtape}\ ls\ l\ \mathsf{nil} & \mathsf{mv\ R}\ (\mathsf{rightof}\ l\ ls) & := \mathsf{rightof}\ l\ ls \\
\mathsf{mv}\ \_\ (\mathsf{niltape}) & := \mathsf{niltape} & \mathsf{mv\ N}\ t & := t
\end{array}
$$

Note that moving further right (or left) when that tape already is to the right (or left) of the symbols, does not change the tape.

The functions $\mathsf{left}, \mathsf{right} : \mathsf{Tape} \to \mathcal{L}(\Sigma)$ return the symbols to the left (or right) side of the head:

Definition 2.15 (left **and** right**)**

$$
\begin{array}{llll}
\mathsf{left}\ (\mathsf{niltape}) & := \mathsf{nil} & \mathsf{right}\ (\mathsf{niltape}) & := \mathsf{nil} \\
\mathsf{left}\ (\mathsf{leftof}\ r\ rs) & := \mathsf{nil} & \mathsf{right}\ (\mathsf{leftof}\ r\ rs) & := r :: rs \\
\mathsf{left}\ (\mathsf{midtape}\ ls\ m\ rs) & := ls & \mathsf{right}\ (\mathsf{midtape}\ ls\ m\ rs) & := rs \\
\mathsf{left}\ (\mathsf{rightof}\ l\ ls) & := l :: ls & \mathsf{right}\ (\mathsf{rightof}\ l\ ls) & := \mathsf{nil}
\end{array}
$$

Now we can define the function $\mathsf{wr} : \mathsf{Tape}_\Sigma \to \mathcal{O}(\Sigma) \to \mathsf{Tape}_\Sigma$, that writes an optional symbol to a tape. When we write $\lfloor a \rfloor$, we get a midtape, where the left and right symbols remain unchanged and a is now in the middle. For $\emptyset$, the tape remains unchanged. Note that there is no way to decrease the number of symbols on a tape or to write "blank" symbols.

Definition 2.16 (wr**)**

$$
\begin{array}{ll}
\mathsf{wr}\ t\ \emptyset & := t \\
\mathsf{wr}\ t\ \lfloor a \rfloor & := \mathsf{midtape}\ (\mathsf{left}\ t)\ a\ (\mathsf{right}\ t)
\end{array}
$$

To define the function $\mathsf{step} : \mathsf{Conf} \to \mathsf{Conf}$, we need to know the symbols on the tapes. Therefore, we define a function $\mathsf{current} : \mathsf{Tape}_\Sigma \to \mathcal{O}(\Sigma)$. It returns $\emptyset$ if the head is not under a symbol, and $\lfloor m \rfloor$ if the head is under the symbol m.

Definition 2.17 (current**)** *The function* $\mathsf{current} : \mathsf{Tape}_\Sigma \to \mathcal{O}(\Sigma)$ *is defined by*

$$\begin{aligned} \mathsf{current}\ (\mathsf{midtape}\ \mathtt{ls}\ \mathtt{m}\ \mathtt{rs}) &:= \lfloor \mathtt{m} \rfloor \\ \mathsf{current}\ \_ &:= \emptyset \end{aligned}$$

We can state a correctness lemma of the function wr:

Fact 2.18 (Correctness of wr**)** *For all tapes* $\mathtt{t}$ *and symbols* $\mathtt{a} : \Sigma$*:*

1. $\mathsf{right}(\mathsf{wr}\ \mathtt{t}\ \lfloor \mathtt{a} \rfloor) = \mathsf{right}(\mathtt{t})$
2. $\mathsf{left}(\mathsf{wr}\ \mathtt{t}\ \lfloor \mathtt{a} \rfloor) = \mathsf{left}(\mathtt{t})$
3. $\mathsf{current}(\mathsf{wr}\ \mathtt{t}\ \lfloor \mathtt{a} \rfloor) = \lfloor \mathtt{a} \rfloor$

We can now define the function $\mathsf{step} : \mathsf{Conf} \to \mathsf{Conf}$. First, the machine reads all the current symbols from the tapes. We apply this vector and the machine state to the transition function δ. Then, each tape writes the symbol and moves its head into the direction that δ yielded for it. The machine ends up in a new state $\mathtt{q}'$.

Definition 2.19 (step**)**

$$\begin{aligned} \mathsf{doAct}\ \mathtt{t}\ (\mathtt{s}, \mathtt{d}) &:= \mathsf{mv}\ \mathtt{d}\ (\mathsf{wr}\ \mathtt{t}\ \mathtt{s}) \\ \mathsf{step}\ (\mathtt{q}, \mathtt{t}) &:= \mathsf{let}\ (\mathtt{q}', \mathtt{act}) := \delta(\mathtt{q}, \mathsf{map}\ \mathsf{current}\ \mathtt{t})\ \mathsf{in} \\ &\qquad (\mathtt{q}', \mathsf{map}_2\ \mathsf{doAct}\ \mathtt{t}\ \mathtt{act}) \end{aligned}$$

To define the execution of a machine, we first define an abstract recursive function $\mathsf{loop} : (A \to A) \to (A \to \mathbb{B}) \to A \to \mathbb{N} \to \mathcal{O}(A)$ (for every $A : \mathbb{T}$):

Definition 2.20 (loop**)**

$$\mathsf{loop}\ \mathtt{f}\ \mathtt{h}\ \mathtt{a}\ \mathtt{k} := \begin{cases} \lfloor \mathtt{a} \rfloor & \mathtt{h}(\mathtt{a}) \\ \emptyset & \neg \mathtt{h}(\mathtt{a}) \wedge \mathtt{k} = 0 \\ \mathsf{loop}\ \mathtt{f}\ \mathtt{h}\ (\mathtt{f}\ \mathtt{a})\ (\mathtt{k} - 1) & \neg \mathtt{h}(\mathtt{a}) \wedge \mathtt{k} > 0 \end{cases}$$

We can show basic facts about loop.

Lemma 2.21 (Basic facts about loop**)** *Let* $\mathtt{k}, \mathtt{l} : \mathbb{N}$ *and* $\mathtt{a}, \mathtt{b}, \mathtt{c} : A$*.*

1. *If* $\mathtt{k} \leqslant \mathtt{l}$ *and* $\mathsf{loop}\ \mathtt{f}\ \mathtt{h}\ \mathtt{a}\ \mathtt{k} = \lfloor \mathtt{b} \rfloor$*, then* $\mathsf{loop}\ \mathtt{f}\ \mathtt{h}\ \mathtt{a}\ \mathtt{l} = \lfloor \mathtt{b} \rfloor$*.*
2. *If* $\mathsf{loop}\ \mathtt{f}\ \mathtt{h}\ \mathtt{a}\ \mathtt{k} = \lfloor \mathtt{b} \rfloor$ *and* $\mathsf{loop}\ \mathtt{f}\ \mathtt{h}\ \mathtt{a}\ \mathtt{l} = \lfloor \mathtt{c} \rfloor$*, then* $\mathtt{b} = \mathtt{c}$*.*
3. *If* $\mathsf{loop}\ \mathtt{f}\ \mathtt{h}\ \mathtt{a}\ \mathtt{k} = \lfloor \mathtt{b} \rfloor$*, then* $\mathtt{h}(\mathtt{b}) = \mathsf{true}$*.*

4. *If* $h\ a = \mathsf{true}$, *then* $\mathsf{loop}\ f\ h\ a\ k = \lfloor a \rfloor$.
5. *If* $h\ a = \mathsf{true}$ *and* $\mathsf{loop}\ f\ h\ a\ k = \lfloor b \rfloor$, *then* $a = b$.

Proof Claims 1, 2, and 3 follow by induction on $k : \mathbb{N}$. Claim 4 follows by Definition. Claim 5 is a direct consequence of claim 4. □

We instantiate the abstract loop function and get a function $\mathsf{loopM} : \mathsf{Conf} \to \mathbb{N} \to \mathcal{O}(\mathsf{Conf})$ that executes k steps of the machine:

Definition 2.22 (Machine execution)

$$\begin{aligned} \mathsf{initConf}\ t &:= (t, start) \\ \mathsf{haltConf}\ (t, q) &:= halt(q) \\ \mathsf{loopM}\ c\ k &:= \mathsf{loop\ step\ haltConf}\ c\ k \end{aligned}$$

We write $M(c) \triangleright^k c'$ for $\mathsf{loopM}\ c\ k = \lfloor c' \rfloor$ and $M(t) \triangleright^k c$ for $M(\mathsf{initConf}\ t) \triangleright^k c$.

All definitions, except labelled machines, are from Asperti and Ricciotti [2], with similar names. However, the loop function was slightly changed for convenience, so that it needs zero steps when the (abstract) starting state is a halting state.

2.3 Specification of Semantics

We have defined semantics for multi-tape Turing machines. Now we want to define predicates to specify the semantics of a concrete machine $M : \mathsf{TM}^n_\Sigma(L)$. There are two parts of the semantics: **correctness** and **time complexity**.

The correctness part is captured by realisation of a (labelled) relation R:

Definition 2.23 (Realisation) *Let* $M : \mathsf{TM}^n_\Sigma(L)$ *and* $R \subseteq \mathsf{Tape}^n_\Sigma \times L \times \mathsf{Tape}^n_\Sigma$.

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

Where $lab_M : Q_M \to L$ *is the labelling function of* M.

If $M \vDash R$, we say that M realises the relation R. Informally, this means that the output of the machine is correct w.r.t. the relation, if the machine terminates.

To show realisation, e.g. $M \vDash R$, it suffices to find a (smaller) relation R' and show that it implies the (target) relation R:

Lemma 2.24 (Monotonicity of $\vDash R$) *If* $M \vDash R'$ *and* $R' \subseteq R$, *then* $M \vDash R$.

The running time part of the semantics implies termination of the machine on certain inputs. It relates the input $t : \mathsf{Tape}_\Sigma^n$ to the number of steps that the machine needs for the computation.

Definition 2.25 (Termination in a running time relation) *Let* $T \subseteq \mathsf{Tape}_\Sigma^n \times \mathbb{N}$.

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

Termination is anti-monotone. This means that it suffices for showing $M \downarrow T$ to find a (bigger) relation T' and show $T \subseteq T'$.

Lemma 2.26 (Anti-monotonicity of $M \downarrow T$) *If* $M \downarrow T'$ *and* $T \subseteq T'$, *then* $M \downarrow T$.

For machines that always terminate in a constant number of steps, it is useful to combine both predicates:

Definition 2.27 (Realisation in constant time)

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

Lemma 2.28 $M \vDash^k R \leftrightarrow M \vDash R \wedge M \downarrow (\lambda\_\ k'.\ k \leqslant k')$

Lemma 2.29 (Monotonicity of $M \vDash^k R$) *If* $M \vDash^{k'} R'$, $k' \leqslant k$, *and* $R' \subseteq R$, *then* $M \vDash^k R$.

Asperti and Ricciotti [2] make a distinction between weak and strong realisation, where the strong version implies termination for every input, however, in an uncertain number of steps. We use a variant of their weak realisation. They have no notion of time complexity. Because we do not want to reason about concrete machine states, we introduced the notion of labelled machines. The idea to label the states of machines with elements of a finite type is due to Y. Forster and F. Kunze. They also invented the realisation of labelled relations and the notion for time complexity.

Chapter 3

Primitive Machines

In this chapter, we define several classes of *primitive* machines.[1] These machines are defined on an arbitrary finite alphabet Σ, but they have at most one tape, and they terminate after at most one transition. Primitive machines are the only concrete machines for that we give transition functions δ explicitly. They are also the only concrete machines, for that we fully specify the type of machine states Q. In the following two chapters, we will show how to combine these machines and build (and verify) complex machines without mentioning Q or δ again. All states and transition functions will be derived from these machines.

3.1 Null

The machine Null_Σ is parametrised over the alphabet Σ. Because Σ is always clear from the context, we leave the index out. We fix an alphabet Σ. Null has zero tapes and terminates immediately, i.e. after 0 steps.

Definition 3.1 (Null) $\mathsf{Null} : \mathsf{TM}^0_\Sigma$ *is defined as follows:*

$$\begin{aligned} Q &:= 1 \\ \mathtt{start} &:= () \\ \delta\ \_ &:= ((), \mathsf{nil}) \\ \mathtt{halt}\ \_ &:= \mathsf{true} \end{aligned}$$

Note that if we have an unlabelled machine $M : \mathsf{TM}^n_\Sigma$, we implicitly label their states over 1 with the labelling function $\mathtt{lab}_M(q) := ()$.

The correctness relation is the universal relation, because empty vectors do not have information. However, the correctness lemma also states that the machine terminates in 0 steps.

[1] Asperti and Ricciotti [2] call these machines *basic* machines.

Lemma 3.2 (Correctness of Null**)** $\mathsf{Null} \vDash^0 \mathtt{NullRel}$ *with* $\mathtt{NullRel} := \lambda\, t\, t'.\ \top$.

Proof By execution. The machine terminates after zero steps in the state $()$. □

3.2 DoAct

Machines of the next class, $\mathsf{DoAct}\ a : \mathsf{TM}^1_\Sigma(L)$, do only one action $a : \mathsf{Act}$ (i.e. they optionally write a symbol and move the head of the tape) and terminate.

Definition 3.3 (DoAct **a)** *Let* $a : \mathsf{Act}_\Sigma$. *Then* $\mathsf{DoAct}\ a : \mathsf{TM}^1_\Sigma$ *is defined as follows:*

$$\begin{aligned} Q &:= \mathbb{B} \\ \mathtt{start} &:= \mathsf{false} \\ \delta\ \_ &:= (\mathsf{true}, [a]) \\ \mathtt{halt}\ b &:= b \end{aligned}$$

The semantics of DoAct is easily expressed using the function doAct (see Definition 2.19). The machine terminates after one transition:

Lemma 3.4 (Correctness of DoAct**)** $\mathsf{DoAct}\ a \vDash^1 \mathtt{DoActRel}\ a$ *with*

$$\mathtt{DoActRel}\ a := \lambda t\, t'.\ t'[0] = \mathsf{doAct}\ t[0]\ a.$$

We define some abbreviations:

Definition 3.5 (Machine classes derived from DoAct**)**

$$\begin{aligned} \mathsf{Move}\ d &:= \mathsf{DoAct}(\emptyset, d) \\ \mathsf{Write}\ s &:= \mathsf{DoAct}(\lfloor s \rfloor, N) \\ \mathsf{WriteMove}\ s\ d &:= \mathsf{DoAct}(\lfloor s \rfloor, d) \end{aligned}$$

3.3 Read

$\mathsf{Read} : \mathsf{TM}^1_\Sigma(\mathcal{O}(\Sigma))$ is an interesting class of labelled one-tape machines. The machines of this class have one terminating state for each symbol of the alphabet Σ. They read the current symbol from the tape and terminate in the state that corresponds to that symbol. For the case that there is no current symbol, they also have a distinct terminating state. The labelling function maps the terminating state for the symbol s to the label $\lfloor s \rfloor$.

Definition 3.6 (Read**)** *The machine* $\mathsf{Read} : \mathsf{TM}^1_\Sigma(\mathcal{O}(\Sigma))$ *is defined as follows:*

$$
\begin{aligned}
Q &:= \mathbb{B} + \Sigma \\
\mathtt{start} &:= \mathsf{inl}\,\mathsf{false} \\
\delta(\_, s) &:= \begin{cases} (\mathsf{inl}\,\mathsf{true}, [(\emptyset, N)]) & s[0] = \emptyset \\ (\mathsf{inr}\,c, [(\emptyset, N)]) & s[0] = \lfloor c \rfloor \end{cases} \\
\mathtt{halt}\ (\mathsf{inl}\,b) &:= b \\
\mathtt{halt}\ (\mathsf{inr}\,\_) &:= \mathsf{true} \\
\mathtt{lab}\ (\mathsf{inl}\,\_) &:= \emptyset \\
\mathtt{lab}\ (\mathsf{inr}\,s) &:= \lfloor s \rfloor
\end{aligned}
$$

The correctness lemma of Read states that the machine terminates after one step, leaves the tape unchanged, and that the label of the terminating state corresponds to the current symbol on the tape.

Lemma 3.7 (Correctness of Read**)** $\mathsf{Read} \vDash^1 \mathtt{ReadRel}$ *with*

$$\mathtt{ReadRel} := \lambda\, t\ (l, t').\ l = \mathsf{current}\ t[0] \wedge t' = t$$

Proof Case distinction over current $t[0]$. Both cases follow by executing the machine one step. □

Chapter 4

Combining Turing Machines

Recall that Turing machines are unstructured – the execution of a machine could continue from one state to any other state of the machine. There is also no trivial way how to sequentially compose two machines. We fix these problems and introduce *control-flow* operators, like "if then else", "sequential composition", and "while". Thus, we do not need to define transition functions δ or even states Q for complex machines, because we only combine machines using these operators. The transition function of the sequential composition of M_1 and M_2 is derived from the transition functions of M_1 and M_2. But maybe more importantly, this also makes correctness of machines compositional: From the correctness of M_1 and M_2, we can conclude the correctness of the sequential composition of M_1 and M_2. Also note that the number of machine states can be *huge* for complex machines, so it is important to not refer to any concrete machine state, either in the definition or in the verification of a machine.[1]

We define control-flow operators as *first-class citizen* in Coq's type theory. As a result, we get a *shallow-embedded* language for programming multi-tape Turing machines in an imperative way. The primitive machines defined in Chapter 3 serve as the "primitive instructions" of our language. Another possible approach, called *deep embedding*, is to define a fixed syntax of a language as an inductive data type. We would need a compiler from syntax trees of Turing machine programs to actual Turing machines. This compiler would have to be extended, whenever we add a new feature to our language. In our approach, we can just "extend" our language by defining a function, and we use the same verification techniques (i.e. realisation and termination) on each stage of the development.

[1] Using this framework, we define and verify a machine with 11537 states.

4.1 Switch

Asperti and Ricciotti [2] define the control-flow operators sequential composition, conditional, and while. Their conditional operator takes a concrete machine state as a parameter. We noticed that this is not acceptable. We now introduce a generalised operator on labelled machines, called Switch, and derive sequential composition and conditional from this operator.

Definition 4.1 (Switch M f) *Let* $M : \mathsf{TM}^n_\Sigma(L)$ *and* $f : L \to \mathsf{TM}^n_\Sigma(L')$. *We define the machine* $\mathsf{Switch}\ M\ f : \mathsf{TM}^n_\Sigma(L')$ *with the following components:*

$$
\begin{aligned}
Q &:= Q_M + \sum_{l:L} Q_{f(l)} \\
start &:= \mathsf{inl}\, start_M \\
\delta\ (\mathsf{inl}\, q, s) &:= \begin{cases} \big(\mathsf{inr}(lab_M\ q, start_{f(lab_M\ q)}), (\emptyset, N)^n\big) & halt_M(q) \\ \mathsf{let}\ (q', act) := \delta_M(q, s)\ \mathsf{in}\ (\mathsf{inl}\, q', act) & \neg halt_M(q) \end{cases} \\
\delta\ (\mathsf{inr}\, q, s) &:= \mathsf{let}\ (q', act) := \delta_{f(\pi_1 q)}(\pi_2 q, s)\ \mathsf{in}\ \big(\mathsf{inr}(\pi_1 q, q'), act\big) \\
halt\ (\mathsf{inl}\, q) &:= \mathsf{false} \\
halt\ (\mathsf{inr}\, q) &:= halt_{f(\pi_1\ q)}(\pi_2\ q) \\
lab\ (\mathsf{inl}\, q) &:= \_ \\
lab\ (\mathsf{inr}\, q) &:= lab_{f(\pi_1\ q)}(\pi_2\ q)
\end{aligned}
$$

In Definition 4.1, the lab value for inl is unimportant, because the lifted states of M are not terminating states for $\mathsf{Switch}\ M\ f$. We just use a canonical value.

$\mathsf{Switch}\ M\ f$ first executes a copy of M. When it reaches a final state q of M, it does a "nop" action (i.e. $(\emptyset, N)^n$) and changes to the injection of the start state of the machine $f(lab_M\ q)$. When $\mathsf{Switch}\ M\ f$ reaches a state that is the injection of a final state of a machine $f\ l$, it terminates. The correctness part of the semantics can be expressed using the following lemma:

Lemma 4.2 (Correctness of $\mathsf{Switch}\ M\ f$) *Let* $R \subseteq \mathsf{Tape}^n_\Sigma \times L \times \mathsf{Tape}^n_\Sigma$ *and* $R'\ l \subseteq \mathsf{Tape}^n_\Sigma \times L' \times \mathsf{Tape}^n_\Sigma$ *for all* $l : L$. *If* $M \vDash R$ *and* $f\ l \vDash R'\ l$ *for all* $l : L$, *then*

$$\mathsf{Switch}\ M\ f \vDash SwitchRel\ R\ R'$$

with

$$SwitchRel\ R\ R' := \bigcup_{l:L} \big(R|_l \circ R'\ l\big)$$

Note that in the correctness relation, we compose the unlabelled relation $R|_l \subseteq \mathsf{Tape}^n_\Sigma \times \mathsf{Tape}^n_\Sigma$ with the labelled relation $R'\ l \subseteq \mathsf{Tape}^n_\Sigma \times L' \times \mathsf{Tape}^n_\Sigma$. This means that $\mathsf{Switch}\ M\ f$ terminates in a state with a label of $f\ l$, which has type L'.

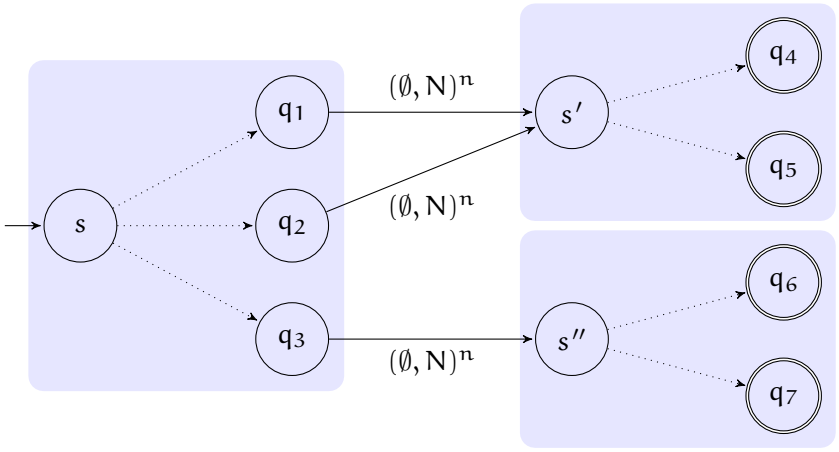


Figure 4.1: Example of a Switch. The left box stands for the first machine $M_1 : \mathsf{TM}^n_\Sigma(\mathbb{B})$. The final states q_1 and q_2 are mapped to true and q_3 to false. After Switch reaches one of the injections of the terminal states q_1, q_2, q_3 of M_1, it continues its execution either in the top case-machine M_2 or in the bottom case-machine M_3. The halting states of Switch are exactly the injections of the halting states of the case-machines.

To specify the running time of Switch M f, we need to know the running time relation in that M terminates, and for each $l : L$ the running time relation of $f\ l$. We also need to know the correctness relation of M, because the running time of $f\ l$ depends on the output of M, which is the input of $f\ l$. Also, the choice of the case-machine depends on the label of the terminating state of M.

Lemma 4.3 (Running Time of Switch M f) *Let* $R \subseteq \mathsf{Tape}^n_\Sigma \times L \times \mathsf{Tape}^n_\Sigma$, $T \subseteq \mathsf{Tape}^n_\Sigma \times \mathbb{N}$, *and* $T'\ l \subseteq \mathsf{Tape}^n_\Sigma \times \mathbb{N}$ *for all* $l : L$. *If* $M \vDash R$, $M \downarrow T$, *and* $f\ l \downarrow T'\ l$ *for all* $l : L$, *then* Switch M f $\downarrow$ *SwitchT R T T′, where*

$$\mathit{SwitchT}\ R\ T\ T' := \lambda\, t\ k.\ \exists\, k_1\ k_2.\ T\ t\ k_1 \wedge 1 + k_1 + k_2 \leqslant k \wedge \forall\, l\ t'.\ R\ t\ (l, t') \to T'(l)\ t'\ k_2$$

We can combine the correctness and running time lemma, in case M and every $f\ l$ terminates in constant time.

Lemma 4.4 (Correctness of Switch M f in constant time) *Let* $k_1, k_2 : \mathbb{N}$. *If* $M \vDash^{k_1} R$ *and* $f\ l \vDash^{k_2} R'\ l$ *for every* $l : L$, *then*

$$\mathsf{Switch}\ M\ f \vDash^{1+k_1+k_2} \mathit{SwitchRel}\ R\ R'$$

Proof Follows with the Lemmas 2.28, 4.2, and 4.3. □

4.1.1 Derived Operators

As mentioned above, conditional and sequential composition can be defined as instances of the Switch operator. For the sequential composition $M_1;\ M_2$ with $M_1 : \mathsf{TM}^n_\Sigma(L)$ and $M_2 : \mathsf{TM}^n_\Sigma(L')$, the function $f : L \to \mathsf{TM}^n_\Sigma(L')$, maps all labels of M_1 to the machine M_2.

Definition 4.5 (Sequential composition) *Let* $M_1 : \mathsf{TM}^n_\Sigma(L)$ *and* $M_2 : \mathsf{TM}^n_\Sigma(L')$.

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

This means that regardless in which state the machine M_1 terminates, the sequential composition $M_1;\ M_2$ continues its execution in the start state of M_2. The following lemma is the correctness lemma of sequential composition for constant-time termination. The version without constant-time termination, i.e. the lemma we get notationally by removing the step numbers, holds as well.

Lemma 4.6 (Correctness of sequential composition) *If* $M_1 \vDash^{k_1} R_1$ *and* $M_2 \vDash^{k_2} R_2$, *then* $M_1;\ M_2 \vDash^{1+k_1+k_2} \bigcup_{l:L} (R_1|_l \circ R_2)$.

We often have the case that the first machine M_1 is labelled over the unit-type $L = 1$. In this case, the relation is $(\bigcup_{l:L} R_1|_l) \circ R_2 \equiv R_1|_{()} \circ R_2 = R_1 \circ R_2$ by the convention that we identify unit-labelled relations with unlabelled relations. This means that sequential composition of two machines amounts to composing their correctness relations.

In case either M_1 or M_2 do not have constant running time, we need the running time lemma, which can be derived from Lemma 4.3.

Lemma 4.7 (Running Time of sequential composition) *If* $M_1 \vDash R_1$, $M_1 \downarrow T_1$, *and* $M_2 \downarrow T_2$, *then*

$$M_1;\ M_2 \downarrow \left(\lambda t\ k.\ \exists k_1\ k_2.\ T_1\ t\ k_1\ \wedge\ 1 + k_1 + k_2 \leqslant k\ \wedge\ \forall t'\ l.\ R_1\ t\ (l, t') \to T_2\ t'\ k_2\right)$$

For the conditional $\mathsf{If}\ M_1\ \mathsf{Then}\ M_2\ \mathsf{Else}\ M_3$ with $M_1 : \mathsf{TM}^n_\Sigma(\mathbb{B})$, $M_2, M_3 : \mathsf{TM}^n_\Sigma(L)$, the function $f : \mathbb{B} \to \mathsf{TM}^n_\Sigma(L)$ simply maps true to M_2 and false to M_3. The conditional, as defined below, first executes M_1. If M_1 terminates in a state with label true, it continues the execution in M_2, else in M_3.

Definition 4.8 (Conditional) *Let* $M_1 : \mathsf{TM}^n_\Sigma(\mathbb{B})$, $M_2, M_3 : \mathsf{TM}^n_\Sigma(L)$.

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

Lemma 4.9 (Correctness of conditional) *If* $M_1 \vDash^{k_1} R_1$, $M_2 \vDash^{k_2} R_2$, *and* $M_3 \vDash^{k_3} R_3$, *then* $\mathsf{If}\ M_1\ \mathsf{Then}\ M_2\ \mathsf{Else}\ M_3 \vDash^{1+k_1+\max(k_2+k_3)} (R_1|_{\mathsf{true}} \circ R_2) \cup (R_1|_{\mathsf{false}} \circ R_3)$.

The correctness relation $(R_1|_{\mathsf{true}} \circ R_2) \cup (R_1|_{\mathsf{false}} \circ R_3)$ captures the idea of the following case-distinction: If the conditional machine terminates, than the copy of M_1 either terminates in a state with label true or false. In the first case, the conditional proceeds in M_2, else in M_3.

The running time lemma of the conditional can also be derived from Lemma 4.3.

Lemma 4.10 (Running Time of the conditional) *If* $M_1 \vDash R_1$, $M_1 \downarrow T_1$, $M_2 \downarrow T_2$, *and* $M_3 \downarrow T_3$, *then*

$$\begin{aligned}&\mathsf{If}\ M_1\ \mathsf{Then}\ M_2\ \mathsf{Else}\ M_3 \downarrow \\ &\quad \big(\lambda t\ k.\ \exists k_1\ k_2.\ T_1\ t\ k_1\ \wedge\ 1 + k_1 + k_2 \leqslant k \wedge \\ &\qquad\qquad \forall b\ t'.\ R_1\ t\ (b, t') \rightarrow \mathsf{if}\ b\ \mathsf{then}\ T_2\ t'\ k_2\ \mathsf{else}\ T_3\ t'\ k_2\big)\end{aligned}$$

Asperti and Ricciotti [2] also define sequential composition and a conditional operator. However, they have two fundamental differences. First, they define and verify each operator separately. Secondly, they do not consider state labelling. Their conditional operator takes one concrete "negative" state, i.e. the machine M_3 is only executed if M_1 terminates in this particular state. This makes programming and reasoning about concrete machines tedious, because complex machines also have many states. They also have to introduce a separate notion for correctness, because they do not have states in their original one. By introducing state-labelled machines and by implementing the more general Switch operator, we solve both problems. We no longer have to specify concrete machine states in the definition and verification of machines. Moreover, we often exploit the convenient generality of the Switch operator. For example, it can be used to implement a machine that copies a symbol from one tape to another tape, as demonstrated in Chapter 4. Also note, that Asperti and Ricciotti [2] have no notion of time complexity; their strong notion of realisation only implies termination in an uncertain number of steps. The idea of state-labelling and Switch is due to Y. Forster and F. Kunze.

4.1.2 Proof of Switch

The idea of the proofs of Lemma 4.2 and Lemma 4.3 is to abstract two features of the machine: lifting of configurations from one abstract machine to another abstract machines, and sequencing of two executions. We formalise these two concepts for abstract machines, i.e. we argue on the abstract loop function.

For the first feature, *lifting*, we assume two types A, B for abstract configurations, a function $\mathtt{lift} : A \to B$, two step functions $f : A \to A$, $f' : B \to B$, and two halting functions $h : A \to \mathbb{B}$, $h' : B \to \mathbb{B}$. We assume that the step functions f and f' are compatible with $\mathtt{lift}$ in non-halting states of A. Formally, this means:

$$\forall a : A.\ h(x) = \mathsf{false} \rightarrow f'(\mathtt{lift}\ x) = \mathtt{lift}(f\ x) \tag{4.1}$$

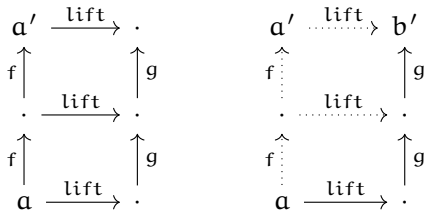


Figure 4.2: Instances of Lemma 4.11 (left) and Lemma 4.12 (right) for $k = 2$ as commuting diagrams. Dotted lines denote existentials. Note that the rectangles correspond to condition (4.1). Only the top "states" are terminating states.

The second assumption is that h' and h are compatible w.r.t. lift; formally:

$$\forall a : A.\ h'(\mathsf{lift}\ x) = h(x) \tag{4.2}$$

Under these two assumptions we can show two lemmas that essentially say that the second abstract machine B simulates the first machine A.

Lemma 4.11 (Loop lifting) *Under the assumptions (4.1) and (4.2):*

$$\begin{aligned}&\forall(k : \mathbb{N})\ (a\ a' : A).\\ &\quad \mathsf{loop}\ f\ h\ a\ k = \lfloor a' \rfloor \to\\ &\quad \mathsf{loop}\ f'\ h'\ (\mathsf{lift}\ a)\ k = \lfloor \mathsf{lift}\ a' \rfloor\end{aligned}$$

Proof By induction on $k : \mathbb{N}$. □

Lemma 4.12 (Loop unlifting) *Under the assumptions (4.1) and (4.2):*

$$\begin{aligned}&\forall(k : \mathbb{N})\ (a : A)\ (b' : B).\\ &\quad \mathsf{loop}\ f'\ h'\ (\mathsf{lift}\ a)\ k = \lfloor b' \rfloor \to\\ &\quad \exists(a' : A).\ \mathsf{loop}\ f\ h\ a\ k = \lfloor a' \rfloor \wedge b' = \mathsf{lift}\ a'\end{aligned}$$

Proof By induction on $k : \mathbb{N}$. □

The Lemmas 4.11 and 4.12 are visualised in Figure 4.2.

For the second feature, *sequential execution*, we assume another type A with a step function $f : A \to A$ and two halting functions $h, h' : A \to \mathbb{B}$. We assume, that if a is a non-halting state w.r.t. h, then a also is a non-halting state w.r.t. h':

$$\forall(a : A).\ h\ a = \mathsf{false} \to h'\ a = \mathsf{false} \tag{4.3}$$

Lemma 4.13 (Loop merging) *Under assumption (4.3):*

$$\begin{aligned}&\forall(k_1\ k_2 : \mathbb{N})\ (a_1\ a_2\ a_3 : A).\\ &\quad \mathsf{loop}\ f\ h\ a_1\ k_1 = \lfloor a_2 \rfloor \to\\ &\quad \mathsf{loop}\ f\ h'\ a_2\ k_2 = \lfloor a_3 \rfloor \to\\ &\quad \mathsf{loop}\ f\ h'\ a_1\ (k_1 + k_2) = \lfloor a_3 \rfloor\end{aligned}$$

Proof By induction on $k_1 : \mathbb{N}$, using Lemma 2.21 (1). □

Lemma 4.14 (Loop splitting) *Under assumption (4.3):*

$$\begin{aligned}&\forall(k : \mathbb{N})\ (a_1\ a_3 : A).\\ &\quad \mathsf{loop}\ f\ h'\ a_1\ k = \lfloor a_3 \rfloor \to\\ &\quad \exists(k_1\ k_2 : \mathbb{N})\ (a_2 : A).\\ &\qquad \mathsf{loop}\ f\ h\ a_1\ k_1 = \lfloor a_2 \rfloor \wedge\\ &\qquad \mathsf{loop}\ f\ h'\ a_2\ k_2 = \lfloor a_3 \rfloor \wedge\\ &\qquad k_1 + k_2 \leqslant k\end{aligned}$$

Proof By complete induction on $k : \mathbb{N}$. □

Back to the verification of Switch $M\ f$. In the following, we simply write Switch. For a configuration c_k, we write q_k and t_k for the state and tapes component of c_k.

The execution steps of Switch are essentially a sequence of first, the lifted steps of an execution of M, second, a "nop" transition, and third, the lifted steps of the execution of $f\ l$. We give the concrete lifting functions from M to Switch and from $f\ l$ to Switch:

Definition 4.15 (Liftings of Switch**)** *We define the functions* $\mathtt{liftL} : \mathsf{Conf}_M \to \mathsf{Conf}_{\mathsf{Switch}}$ *and* $\mathtt{liftR}_l : \mathsf{Conf}_{f(l)} \to \mathsf{Conf}_{\mathsf{Switch}}$ *for all* $l : L$*:*

$$\begin{aligned}\mathtt{liftL}\ (q, t) &:= (\mathsf{inl}\ q, t)\\ \mathtt{liftR}_l(q, t) &:= (\mathsf{inr}(l, q), t)\end{aligned}$$

For the sequential lab, we also have to define the lifted halting function of $\mathtt{haltConfL} : \mathsf{Conf}_{\mathsf{Switch}} \to \mathbb{B}$:

Definition 4.16 (Lifted halting function)

$$\begin{aligned}\mathtt{haltConfL}(\mathsf{inl}\ q, t) &:= \mathtt{halt}_M(q)\\ \mathtt{haltConfL}(\mathsf{inr}\ q, t) &:= \mathsf{true}\end{aligned}$$

Using Lemma 4.13 and 4.11, we can show the lemma we need for running time:

Lemma 4.17 (Merging parts of executions of Switch**)** *Let* $t : \mathsf{Tape}^n_\Sigma$, $k_1, k_2 : \mathbb{N}$, $c_1 : \mathsf{Conf}_M$ *and* $c_2 : \mathsf{Conf}_{f(\mathit{lab}_M\ q_1)}$.

$$\begin{aligned}&M(t) \triangleright^{k_1} c_1 \to\\ &\big(f\ (\mathit{lab}_M\ q_1)\big)(t_1) \triangleright^{k_2} c_2 \to\\ &\mathsf{Switch}(t) \triangleright^{1+k_1+k_2} \mathit{liftR}(c_2)\end{aligned}$$

Proof We apply Lemma 4.13 and have to show:

1. $\forall a : \mathsf{Conf}_{\mathsf{Switch}}.\ \mathit{haltConfL}\ a = \mathsf{false} \to \mathsf{haltConf}\ a = \mathsf{false}$. This holds trivially by case-analysis over a.
2. $\mathsf{loop}\ \mathsf{step}_{\mathsf{Switch}}\ \mathit{haltConfL}\ (\mathsf{initConf}_{\mathsf{Switch}}\ t)\ k_1 = \lfloor \mathit{liftL}\ c_1 \rfloor$. By definition, we have $\mathsf{initConf}_{\mathsf{Switch}}\ t = \mathit{liftL}(\mathsf{initConf}_M\ t)$. The claim follows with Lemma 4.11.
3. $\mathsf{loop}\ \mathsf{step}_{\mathsf{Switch}}\ \mathit{haltConf}_{\mathsf{Switch}}\ (\mathit{liftL}\ c1)\ (1 + k_2) = \lfloor \mathit{liftR}(c_2) \rfloor$. By definition, we know that the first step must be a "nop" transition from $\mathit{liftL}\ c_1$ to $\mathit{liftR}\ \big(\mathsf{initConf}_{f(\mathit{lab}_M\ q_1)}\ t_1\big)$. It remains to show that:

   $$\mathsf{loop}\ \mathsf{step}_{\mathsf{Switch}}\ \mathit{haltConf}_{\mathsf{Switch}}\ \mathit{liftR}\big(\mathsf{initConf}_{f(\mathit{lab}_M\ q_1)}\ t_1\big)\ k_2 = \lfloor \mathit{liftR}(c_2) \rfloor$$

   This follows with Lemma 4.13. □

The running time Lemma 4.3 follows directly from Lemma 4.17.

Lemma 4.18 (Splitting execution of Switch**)** *Let* $t : \mathsf{Tape}^n_\Sigma$, $k : \mathbb{N}$, $c : \mathsf{Conf}_{\mathsf{Switch}}$.

$$\begin{aligned}&\mathsf{Switch}(t) \triangleright^{k} c \to\\ &\exists (k_1\ k_2 : \mathbb{N})\ (c_1 : \mathsf{Conf}_M)\ (c_2 : \mathsf{Conf}_{f(\mathit{lab}_M\ q_1)}).\\ &\quad M(t) \triangleright^{k_1} c_1 \wedge\\ &\quad \big(f(\mathit{lab}_M\ q_1)\big)(t_1) \triangleright^{k_2} c_2 \wedge\\ &\quad c = \mathit{liftR}(c_2)\end{aligned}$$

Proof Analogous to the proof of Lemma 4.17, using Lemmas 4.14 and 4.12. □

The correctness Lemma 4.2 follows directly from Lemma 4.18.

Asperti and Ricciotti [2] have a version of Lemma 4.11 and Lemma 4.13.

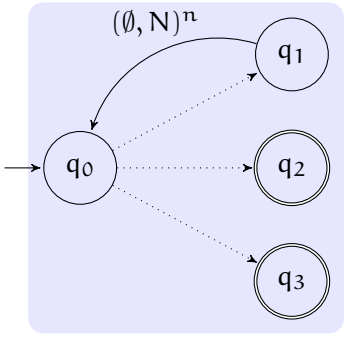


Figure 4.3: Example for While M with $M : \mathsf{TM}^n_\Sigma(\mathcal{O}(\mathbb{B}))$. When M reaches the state q_1, the loop is continued, because q_1 is in assumed to have label $\emptyset$. The other halting states q_2 and q_3 have the labels $\lfloor \mathsf{true} \rfloor$ and $\lfloor \mathsf{false} \rfloor$. Therefore While M terminates in a state with label true or false, when M terminates in q_2 or q_3, respectively.

4.2 While

The machine While M essentially behaves like a "do-while" loop in imperative languages like C. At the end of the execution of the loop body M, M decides either to continue or break out of the loop. If M terminates in a state with label $\emptyset$, then the loop is continued, and if M terminates in $\lfloor l \rfloor$, the loop breaks and While M terminates in a state with label l.

Definition 4.19 (While M) *Let* $M : \mathsf{TM}^n_\Sigma(\mathcal{O}(L))$ *and* $\mathtt{def} : L$. *We define* While $M : \mathsf{TM}^n_\Sigma(L)$ *with the following components.*

$$
\begin{aligned}
Q &:= Q_M \\
\mathtt{start} &:= \mathtt{start}_M \\
\delta\,(q,s) &:= \begin{cases} (\mathtt{start}_M, (\emptyset, N)^n) & \mathtt{halt}_M(q) \\ \delta_M(q,s) & \neg\mathtt{halt}_M(q) \end{cases} \\
\mathtt{halt}\ q &:= \mathtt{halt}_M(q) \wedge \mathtt{lab}_M(q) \neq \emptyset \\
\mathtt{lab}\ q &:= \begin{cases} l & \mathtt{lab}_M(q) = \lfloor l \rfloor \\ \mathtt{def} & \mathtt{lab}_M(q) = \emptyset \end{cases}
\end{aligned}
$$

In Definition 4.19, we have to assume that L is inhabited. However, the choice of $\mathtt{def} : L$ is semantically irrelevant, because While M only halts in states where $\mathtt{lab}_M(q) \neq \emptyset$.

The correctness of While can be expressed using the following lemma:

Lemma 4.20 (Correctness of While**)** *Let* $R \subseteq \mathsf{Tape}_\Sigma^n \times \mathcal{O}(L) \times \mathsf{Tape}_\Sigma^n$. *If* $M \vDash R$, *then* While $M \vDash \mathit{WhileRel}\ R$, *where* $\mathit{WhileRel}\ R \subseteq \mathsf{Tape}_\Sigma^n \times L \times \mathsf{Tape}_\Sigma^n$ *is inductively defined by the following two rules:*

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

We can also express the correctness relation of While using the Kleene star:

Fact 4.21 (Alternative correctness relation for While M**)**

$$\mathit{WhileRel}\ R \equiv (R|_\emptyset)^* \circ \Big(\bigcup_{l:L} (R|_{\lfloor l \rfloor})\Vdash_l\Big)$$

Both definitions of $\mathit{WhileRel}$ should make clear what While M does: It repeats the execution of M as long as it terminates in $\emptyset$, and after M terminates in $\lfloor l \rfloor$, While M terminates in a state with label l. This is visualised in Figure 4.3.

When we want to prove While $M \vDash R$ for some machine $M : \mathsf{TM}_\Sigma^n(\mathcal{O}(L))$ and relation $R \subseteq \mathsf{Tape}_\Sigma^n \times L \times \mathsf{Tape}_\Sigma^n$, we must of course have already proven that $M \vDash R'$ for some relation $R' \subseteq \mathsf{Tape}_\Sigma^n \times \mathcal{O}(L) \times \mathsf{Tape}_\Sigma^n$. Then we apply the monotonicity Lemma 2.24 and the above correctness Lemma 4.20, and have to show $\mathit{WhileRel}\ R' \subseteq R$. We can prove this by induction on the inductive predicate, which is equivalent to applying the following lemma:

Lemma 4.22 (Induction for $\mathit{WhileRel}$**)**

$$\begin{aligned}&\big(\forall t\ t'\ l.\ R'\ t\ (\lfloor l \rfloor, t') \to R\ t\ (l, t')\big) \to\\ &\big(\forall t\ t'\ t''\ l.\ R'\ t\ (\emptyset, t') \to R\ t'\ (l, t'') \to R\ t\ (l, t'')\big) \to\\ &\mathit{WhileRel}\ R' \subseteq R\end{aligned}$$

The running time lemma of While is dual. We define a co-inductive termination relation $\mathit{WhileT}\ R\ T$, where R is the relation that M realises and T is the running time relation in that M terminates.

Lemma 4.23 (Running Time of While M**)** *If* $M \vDash R$ *and* $M \downarrow T$. *Then* While $M \downarrow \mathit{WhileT}\ R\ T$, *where* $\mathit{WhileT}\ R\ T \subseteq \mathsf{Tape}_\Sigma^n \times \mathbb{N}$ *is defined as the following co-inductive running time relation:*

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

When we want to show $\mathsf{While}\, M \downarrow T$, this is dual to showing $\mathsf{While}\, M \vDash R$. We apply the anti-monotonicity Lemma 2.26 and the above running time lemma (where T' is the running time relation in that M terminates), and have to show $T \subseteq \mathit{WhileT}\ R\ T'$. For that, we use the co-induction lemma of WhileT:

Lemma 4.24 (WhileT co-induction) *To show* $T \subseteq \mathit{WhileT}\ R\ T'$*, it suffices to show:*

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

The number k_1 is the number of steps needed for the first iteration of the loop. We have to consider all possible outputs of the first loop. If M terminates in label $\lfloor l \rfloor$ in k_1 steps, then $\mathsf{While}\, M$ also only needs k_1 steps. However, if M terminates in $\emptyset$, and $\mathsf{While}\, M$ needs k_2 steps for all other loops, then $\mathsf{While}\, M$ needs $1 + k_1 + k_2$ steps in total. The one additional step comes from the "nop"-transition back to the starting state.

4.2.1 Proof of While

The running time and correctness proofs are similar to the proofs of Switch, as explained in Section 4.1.2. The configurations of While are exactly the configurations of M, so the lifting function is the identity function. However, the fundamental difference between Switch and While is that While can execute M arbitrarily often; it could also diverge. As a consequence, we also need complete induction on step-numbers, in addition to the loop-splitting and loop-merging lemmas. We present the key lemmas here and the most important parts of the proofs.

We simply write While instead of $\mathsf{While}\, M$ in this section. Since we have $\mathsf{Conf}_M = \mathsf{Conf}_{\mathsf{While}}$, we also only write Conf.

The first lemma says that an execution of While consists of an execution of M and a (possibly empty) continuation of While:

Lemma 4.25 (Splitting the execution of While) *Let* $c_1, c_3 : \mathsf{Conf}$ *and* $k : \mathbb{N}$*. Then*

$$\begin{aligned}
&\mathsf{While}(c_1) \triangleright^k c_3 \to \\
&\exists (k_1\ k_2 : \mathbb{N})\ c_2. \\
&\quad M(c_1) \triangleright^{k_1} c_2 \wedge \\
&\quad \mathsf{While}(c_2) \triangleright^{k_2} c_3 \wedge \\
&\quad k_1 + k_2 \leqslant k
\end{aligned}$$

Proof Follows with Lemma 4.14 and Lemma 4.11. □

We have two more splitting lemmas, one for the case that While terminates immediately, and one for the case that While continues the loop.

Lemma 4.26 (Splitting, break case) *Let* $c_1, c_2 : \mathsf{Conf}$, $k : \mathbb{N}$, *and* $l : L$.

$$\begin{aligned}&\mathsf{While}(c_1) \triangleright^k c_2 \to\\ &\mathtt{haltConf}_M\ c_1 \to\\ &\mathtt{lab}_M(q_1) = \lfloor l \rfloor \to\\ &c_1 = c_2\end{aligned}$$

Proof By Lemma 2.21 (5), because c_1 is a halting state of While. □

Lemma 4.27 (Splitting, continue case) *Let* $c_1, c_2 : \mathsf{Conf}$, $k : \mathbb{N}$.

$$\begin{aligned}&\mathsf{While}(c_1) \triangleright^k c_2 \to\\ &\mathtt{haltConf}_M\ c_1 \to\\ &\mathtt{lab}_M(q_1) = \emptyset \to\\ &\exists k'.\, k = 1 + k' \wedge \mathsf{While}(t_1) \triangleright^{k'} c_2\end{aligned}$$

Proof While must have taken the "nop"-transition from c_1 to $\mathtt{initConf}\ t_1$, because c_1 is a halting configuration of M but not of While. □

We now can prove the correctness Lemma 4.20 of While.

Proof We assume $\mathsf{While}(t_1) \triangleright^k c_3$ and have to show $\mathit{WhileRel}\ t_1\ (\mathtt{lab}_{\mathsf{While}}\ q_3, t_3)$. We use complete induction on $k : \mathbb{N}$. By Lemma 4.25, we have $M(t_1) \triangleright^{k_1} c_2$ and $\mathsf{While}(c_2) \triangleright^{k_2} c_3$, for $k_1 + k_2 \leqslant k$. Case analysis.

1. $\mathtt{lab}_M(q_2) = \lfloor l \rfloor$. Then we know by Lemma 4.26, that $c_2 = c_3$. It remains to show $\mathit{WhileRel}\ t_1\ (l, t_2)$. By applying the first constructor, it is enough to show $R\ t_1\ (\lfloor l \rfloor, t_2)$. This follows from the realisation of M.
2. $\mathtt{lab}_M(q_2) = \emptyset$. By Lemma 4.27, we know that $\mathsf{While}(t_2) \triangleright^{k_2'} c_3$ for $k_2 = 1 + k_2'$. The inductive hypothesis gives $\mathit{WhileRel}\ t_2\ (\mathtt{lab}_{\mathsf{While}}\ q_3, t_3)$. The goal follows by applying the second constructor and the realisation of M. □

For the running time proofs, we again have lemmas that "merge" executions together.

Lemma 4.28 (Merging, break case) *Let* $k : \mathbb{N}$, $c_1, c_2 : \mathsf{Conf}$, *and* $l : L$.

$$\begin{aligned} &M(c_1) \triangleright^k c_2 \to \\ &\mathit{lab}_M(q_2) = \lfloor l \rfloor \to \\ &\mathsf{While}(c_1) \triangleright^k c_2 \end{aligned}$$

Lemma 4.29 (Merging, continue case) *Let* $k_1, k_2 : \mathbb{N}$, $c_1, c_2, c_3 : \mathsf{Conf}$.

$$\begin{aligned} &M(c_1) \triangleright^{k_1} c_2 \to \\ &\mathit{lab}_M(q_2) = \emptyset \to \\ &\mathsf{While}(t_2) \triangleright^{k_2} c_3 \to \\ &\mathsf{While}(c_1) \triangleright^{1+k_1+k_2} c_3 \end{aligned}$$

The running time Lemma 4.23 follows similarly, by complete induction on $k : \mathbb{N}$.

4.3 Mirror

We can define a machine operator that "mirrors" a machine M. Whenever M makes a transition with a move to L, Mirror M moves the head to the right instead. For example, we define a machine MoveToSymbol below, that moves the head of the tape right, until it reads a certain symbol. Using this operator, we get a machine "for free" that moves the head to the left instead. However, we still have to copy or parametrise the correctness and running time relations.

Using the proof techniques developed in the previous sections, verifying the Mirror operator is very easy. The function $\mathsf{mirror} : \mathsf{Tape}_\Sigma \to \mathsf{Tape}_\Sigma$ swaps the left and right part of a tape. Furthermore, we define an injective and involutive function $\mathsf{swap} : \mathsf{Move} \to \mathsf{Move}$ that simply swaps the movements L and R.

Definition 4.30 (Mirror M) *Let* $M : \mathsf{TM}^n_\Sigma(L)$. *The machine* Mirror $M : \mathsf{TM}^n_\Sigma(L)$ *has the same components as* M, *except:*

$$\delta(q, s) := \mathsf{let}\ (q', a) := \delta_M(q, s)\ \mathsf{in}\ \big(q', \mathsf{map}\ (\lambda(w, m).\ (w, \mathsf{swap}\ m))\ a\big)$$

The correctness and termination proofs are similar to the proofs of Switch and While. The "lifting" between configurations of M and Mirror is the injective and involutive function $\mathit{mirrorConf} : \mathsf{Conf} \to \mathsf{Conf}$ that simply mirrors the tapes.

Definition 4.31 (Mirror configuration) $\mathit{mirrorConf}(q, t) := (q, \mathsf{map}\ \mathsf{mirror}\ t)$.

Lemma 4.32 (Mirroring steps) *Let* $c_1 : \mathsf{Conf}$. *Then*

$$\mathit{step}_M\ (\mathit{mirrorConf}\ c_1) = \mathit{mirrorConf}\ (\mathit{step}_{\mathsf{Mirror}}\ c_1)$$

Lemma 4.33 (Mirroring executions) *Let* $c_1, c_2 : \mathsf{Conf}$.

1. $\mathsf{Mirror}(c_1) \triangleright^k c_2 \to M(\mathit{mirrorConf}\ c_1) \triangleright^k (\mathit{mirrorConf}\ c_2)$
2. $M(\mathit{mirrorConf}\ c_1) \triangleright^k (\mathit{mirrorConf}\ c_2) \to \mathsf{Mirror}(c_1) \triangleright^k c_2$

Proof Claim 1 follows with Lemma 4.11 and Lemma 4.32. Claim 2 follows with Lemma 4.12 and Lemma 4.32. □

Lemma 4.34 (Correctness of Mirror**)** *Let* $M \vDash R$. *Then* $\mathsf{Mirror}\ M \vDash \mathit{MirrorRel}\ R$ *with*

$$\mathit{MirrorRel}\ R := \lambda t\ (l, t').\ R\ (\mathsf{map\ mirror}\ t)\ (l, \mathsf{map\ mirror}\ t')$$

Proof Follows from Lemma 4.33 (1). □

Lemma 4.35 (Running Time of Mirror**)** *Let* $M \downarrow T$. *Then* $\mathsf{Mirror}\ M \downarrow \mathit{MirrorT}\ T$ *with*

$$\mathit{MirrorT}\ T := \lambda t\ k.\ T\ (\mathsf{map\ mirror}\ t)\ k$$

Proof Follows from Lemma 4.33 (2). □

4.4 Relabelling Operators

The operators of the above sections of this Chapter modify the behaviour of the machine. We can also define simple operators that modify the labelling function $lab : Q_M \to L$. This is for example useful if we want a machine to terminate in one particular label.

Definition 4.36 (Relabel**)** *Let* $M : \mathsf{TM}^n_\Sigma(L)$ *and* $g : L \to L'$.

$$\mathsf{Relabel}\ M\ g := (M, g \circ lab_M)$$

Definition 4.37 (Return**)** *Let* $M : \mathsf{TM}^n_\Sigma(L)$ *and* $l' : L'$.

$$\mathsf{Return}\ M\ l' := \mathsf{Relabel}\ M\ (\lambda\_.\ l')$$

The correctness for these simple operators is obvious. Note that we do not need lemmas for running time, because Definition 2.25 of running time is defined over the bare machine without labelling function. So the running time lemmas for M also apply for Relabel and Return.

Lemma 4.38 (Correctness of Relabel **and** Return**)** *If* $M \vDash R$, *then*

$$\mathsf{Relabel}\ M\ g \vDash \bigcup_{l:L} \left((R|_l) \Vert_{g(l)} \right)$$

$$\mathsf{Return}\ M\ l' \vDash \left(\bigcup_{l:L} R|_l \right) \Vert_{l'}$$

Chapter 5

Lifting Machines

We observe that whenever we want to combine machines, the number of tapes and alphabets of all sub-machines have to agree. For example, we have the following typing rule for the sequential composition of labelled Turing machines:

$$\frac{M_1 : \mathsf{TM}^n_\Sigma(L_1) \qquad M_2 : \mathsf{TM}^n_\Sigma(L_2)}{M_1;\ M_2 : \mathsf{TM}^n_\Sigma(L_2)}$$

Assume that we have a one-tape machine $M_{aux} : \mathsf{TM}^1_\Sigma$ that moves the head to the right of the tape. If we need a two-tape machine M that moves both tapes to the right, we would like to use sequential composition and move one tape after the other tape to the right. But according to the typing rule above, we would need two auxiliary two-tape machines M_1 and M_2, where M_1 moves the first tape to the right and M_2 the second.

There are multiple ways to solve this problem. Maybe the most obvious solution is to define a class of n-tape machines $M_i : \mathsf{TM}^n_\Sigma$ parametrised over the number of tapes and the tape-index i of the tape to move. The machine M_i moves the ith tape to the right. All other tapes are "inactive" and remain unchanged. This approach of parametrising machines over the tape-indices of "active" tapes, however, becomes unhandy for machines with a lot of active tapes.

We choose another approach, because it is in general easier to define and verify machines with a fixed number of tapes. We lift the one-tape machine M_{aux} to n different n-tape machines, for every $n \geqslant 1$. Asperti and Ricciotti [2] implement such an operator that translates a one-tape machine into n-tape machines. We implement a generalised operator that takes an m-tape machine and a mapping from $\mathbb{F}_m$ to $\mathbb{F}_n$, and yields an n-tape machine, for $m \leqslant n$.

The second part of this problem is how to combine machines with different alphabets. For example, if we have a machine Add that adds (encodings of) natural numbers, we could want to build a machine Sum that computes the sum of a list

of numbers. Consider an alphabet $\Sigma_{\mathbb{N}}$ where we could encode natural numbers on and an alphabet $\Sigma_{\mathcal{L}(\mathbb{N})}$ to encode lists of natural numbers. If the alphabet $\Sigma_{\mathbb{N}}$ is included in $\Sigma_{\mathcal{L}(\mathbb{N})}$, we would like to lift Add to the alphabet $\Sigma_{\mathcal{L}(\mathbb{N})}$, to define Sum.

Asperti and Ricciotti [2] avoid the problem of agreement of alphabets. They consider a fixed alphabet to encode all needed data on and implement a universal Turing machine. However, this approach does not scale when we need to encode many different data types. Whenever the alphabet has to be changed, they also must change the definitions of all auxiliary machines. That is why we introduce another operator that lifts a machine to a bigger alphabet.

Both lifting operators are easy to define and verify using the lemmas of the previous chapter (see Section 4.1.2).

5.1 Tape-Lift

The tape-lift takes a machine $M : \mathsf{TM}_{\Sigma}^{m}(L)$ and a duplicate-free vector $I : \mathbb{F}_n^m$, and yields a machine $\Uparrow_I M : \mathsf{TM}_{\Sigma}^{n}(L)$. The tape of $\Uparrow_I M$ with the index $i = I[j]$ (with $i : \mathbb{F}_n$, $j : \mathbb{F}_m$) behaves exactly as the tape j of M. All other tapes of $\Uparrow_I M$ that are not in I are inactive and do not change.

The transition function of $\Uparrow_I M$ gets the n read symbols and selects the m relevant symbols. Then it applies the transition function δ_M with the selected symbols and the current state q. δ_M yields an m-vector $act : \mathsf{Act}^m$ and the continuation state q'. It fills "nop"-actions into act, to get an action vector $act' : \mathsf{Act}^n$.

Definition 5.1 (Vector selecting) *Let* $X : \mathbb{T}$, $m, n : \mathbb{N}$, $I : \mathbb{F}_n^m$, *and* $V : X^m$. *Then* $\mathtt{select}\ I\ V : X^n$ *is defined by* $\mathtt{select}\ I\ V := \mathsf{map}\ \big(\lambda(j : \mathbb{F}_m).\ V[j]\big)\ I$

Lemma 5.2 (Correctness of `select`) *By definition, for* $j : \mathbb{F}_m$, *we have*

$$(\mathtt{select}\ I\ V)[j] = V\big[I[j]\big]$$

Definition 5.3 (Vector filling) *Let* $X : \mathbb{T}$, $m, n : \mathbb{N}$, $I : \mathbb{F}_n^m$, $\mathtt{init} : X^n$, *and* $V : X^m$. *Then* $\mathtt{fill}\ I\ \mathtt{init}\ V : X^n$ *is defined per recursion:*

$$\begin{aligned}&\mathtt{fill}\ (\mathsf{nil})\quad \mathtt{init}\ V := \mathtt{init}\\ &\mathtt{fill}\ (i :: I')\ \mathtt{init}\ V := \mathtt{replace}\ (\mathtt{fill}\ I'\ (\mathsf{tl}\ V))\ i\ (\mathsf{hd}\ V)\end{aligned}$$

Where $\mathtt{replace} : X^n \to \mathbb{F}_n \to X \to X^n$ *replaces the* i*th element of a vector.*

Lemma 5.4 (Correctness of `fill`) *If* I *is duplicate-free, then:*

1. *If* $I[j] = i$, *then* $(\mathtt{fill}\ I\ \mathtt{init}\ V)[i] = V[j]$.
2. *If* $i \notin I$, *then* $(\mathtt{fill}\ I\ \mathtt{init}\ V)[i] = \mathtt{init}[i]$.

Proof By induction on $I : \mathbb{F}_n^m$. □

Definition 5.5 ($\Uparrow_I M$) *Let* $M : \mathsf{TM}_\Sigma^m(L)$ *and* $I : \mathbb{F}_n^m$. *We define* $\Uparrow_I M : \mathsf{TM}_\Sigma^n(L)$. *All components are the same as in* M*, except:*

$$\begin{aligned}\delta(q, s) := \mathsf{let}\ (q', act) := \delta_M(q, \mathtt{select}\ I\ s)\ \mathsf{in}\\ (q', \mathtt{fill}\ I\ (\emptyset, N)^n\ act)\end{aligned}$$

Lemma 5.6 (Correctness of $\Uparrow_I M$) *Let* $M : \mathsf{TM}_\Sigma^m(L)$ *and* $I : \mathbb{F}_n^m$ *duplicate-free. If* $M \vDash R$*, then* $\Uparrow_I M \vDash \Uparrow_I R$ *with*

$$\Uparrow_I R := \lambda t\ (l, t').\ R\ (\mathtt{select}\ I\ t)\ (l, \mathtt{select}\ I\ t') \wedge \forall i : \mathbb{F}_n.\ i \notin I \to t'[i] = t[i]$$

Lemma 5.7 (Running time of $\Uparrow_I M$) *Let* $M : \mathsf{TM}_\Sigma^m(L)$ *and* $I : \mathbb{F}_n^m$ *duplicate-free. If* $M \downarrow T$*, then* $\Uparrow_R M \downarrow \Uparrow_I T$ *with* $\Uparrow_I T := \lambda t\ k.\ T\ (\mathtt{select}\ I\ t)\ k$.

The proofs are similar to the former proofs, i.e. using Lemma 4.11 and Lemma 4.12 with the following configuration lifting function:

$$\mathtt{selectConf}(q, t) := (q, \mathtt{select}\ I\ t)$$

However, for the second part of the correctness, i.e. tapes that are not in I do not change, we need another lemma about loop:

Lemma 5.8 (Mapping loops) *Let* $A : \mathbb{T}$, $f : A \to A$, $h : A \to \mathbb{B}$. *Let* $B : \mathbb{T}$ *and* $g : A \to B$. *If* $g(f\ a) = g(a)$ *for all* $a : A$*, then*

$$\begin{aligned}&\forall (k : \mathbb{N})\ (a_1\ a_2 : A).\\ &\quad \mathsf{loop}\ f\ h\ k\ a_1 = \lfloor a_2 \rfloor \to\\ &\quad g(a_1) = g(a_2)\end{aligned}$$

Proof By induction on $k : \mathbb{N}$. □

We apply this lemma in the proof of Lemma 5.6 with $g := \lambda((q, t) : \mathsf{Conf}).\ t[i]$.

5.2 Alphabet-Lift

Let $M : \mathsf{TM}_\Sigma^n(L)$ be a machine over the alphabet Σ, and $f : \Sigma \hookrightarrow \Gamma$ a retraction on another alphabet Γ. Note that then Γ has at least as many symbols as Σ. We need a default symbol $\mathtt{def} : \Sigma$. In contrast to the default label we needed in the definition of While, the choice of $\mathtt{def}$ is semantically *relevant*. This means that $\mathtt{def}$ should be a symbol that M does not expect to read. The alphabet-lift $\Uparrow_{(f,\mathtt{def})} M : \mathsf{TM}_\Gamma^n(L)$ is a machine over the bigger alphabet Γ.

The transition function of the lifted machine $\Uparrow_{(f,def)} M$ reads the n optional symbols $s : \mathcal{O}(\Gamma)^n$ and tries to translate them to $s' : \mathcal{O}(\Sigma)^n$ using the partial inversion function $f^{-1} : \Gamma \to \mathcal{O}(\Sigma)$. If the symbol $\tau : \Gamma$ has no corresponding symbol in Σ, it must be translated to def. The transition function δ_M of M yields the successor state q' and a vector of actions $act : (\mathcal{O}(\Sigma) \times \mathsf{Move})^n$, which is translated using $f : \Sigma \to \Gamma$ to $act' : (\mathcal{O}(\Gamma) \times \mathsf{Move})^n$.

Definition 5.9 ($\Uparrow_{(f,def)} M$) *Let $f : \Sigma \hookrightarrow \Gamma$, $def : \Sigma$, and $M : \mathsf{TM}^n_\Sigma(L)$. We define the machine $\Uparrow_{(f,def)} M : \mathsf{TM}^n_\Gamma(L)$ with the same components as M, except:*

$$\begin{aligned}\delta(q, s) := \mathsf{let}\ (q', act) := \delta_M(q, \mathsf{map}\ (mapOpt\ (surject\ f\ def))\ s)\ \mathsf{in}\\ (q', \mathsf{map}\ (mapAct\ f)\ act)\end{aligned}$$

With $surject\ f\ def : \Gamma \to \Sigma$*:*

$$surject\ f\ def\ \tau := \begin{cases}\sigma & f^{-1}(\tau) = \lfloor\sigma\rfloor\\ def & f^{-1}(\tau) = \emptyset\end{cases}$$

and with the canonical functions $mapOpt : \forall(X\ Y : \mathbb{T}).\ (X \to Y) \to \mathcal{O}(X) \to \mathcal{O}(Y)$ *and* $mapAct : \forall(\Sigma\ \Gamma : \mathbb{T}).\ (\Sigma \to \Gamma) \to \mathsf{Act}_\Sigma \to \mathsf{Act}_\Gamma$.

For the correctness and running time lemmas, we also need the canonical function

$$mapTape : \forall(\Gamma\ \Sigma : \mathbb{T}).\ (\Gamma \to \Sigma) \to \mathsf{Tape}_\Gamma \to \mathsf{Tape}_\Sigma$$

that maps every symbol on a tape. We write $mapTapes$ for the respective tape-vector function.

Lemma 5.10 (Correctness of $\Uparrow_{(f,def)} M$) *If $M \vDash R$, then $\Uparrow_{(f,def)} M \vDash \Uparrow_{(f,def)} R$ with*

$$\Uparrow_{(f,def)} R := \lambda t\ (l, t').\ R\ (mapTapes\ (surject\ f\ def)\ t)\ (l, mapTapes\ (surject\ f\ def)\ t')$$

Lemma 5.11 (Running time of $\Uparrow_{(f,def)} M$) *If $M \downarrow T$, then $\Uparrow_{(f,def)} M \downarrow \Uparrow_{(f,def)} T$ with*

$$\Uparrow_{(f,def)} T := \lambda t\ k.\ T\ (mapTapes\ (surject\ f\ def)\ t)\ k$$

The proofs are analogous to the former proofs. The configuration lifting is:

$$surjectConf(q, t) := (q, mapTapes\ (surject\ f\ def)\ t)$$

Chapter 6

Simple Machines

In this chapter, we build and verify simple machines using the primitive machines from Chapter 3, the operators developed in Chapter 4, and the tapes-lift from Chapter 5. The machines in this chapter will be useful in the next chapter. We do not use the alphabet-lift yet. First we show how we prove correctness and termination of machines from now on.

When we prove $M \vDash R$ for a machine M and its correctness relation R, we first find a relation R' that M realises. We derive this relation by applying the correctness lemmas of the control-flow operators, lifts, and concrete machines. This process is mechanical and does in general not depend on arbitrary choices (with a few exceptions). The derived relation respects the "structure" of the machine. For example, the relation of a sequential composition is the relational composition of two relations. Using the monotonicity Lemma 2.24, it remains to show $R' \subseteq R$. Because the structure of R' respects the structure of M, the proof of the inclusion also follows its structure. For example, the relation for a conditional is $(R_1|_{\mathsf{true}} \circ R_2) \cup (R_1|_{\mathsf{false}} \circ R_3)$. From that it follows that we do a case-distinction for both branches in the proof. Note that we do not have to reason about machine states at all, because the correctness relations are only relations between tapes and labels.

When M always terminates in constant time k, we show $M \vDash^k R$ instead. Using the monotonicity Lemma 2.29, we can prove correctness and constant time at once. For non-constant running time, we show $M \downarrow T$ for a running time relation T. For that, we use the dual approach and apply the anti-monotonicity Lemma 2.26.

6.1 Nop

Using the tapes-lift (Definition 5.5) and Null (Definition 3.1), it is easy to define an n-tape machine $\mathsf{Nop} : \mathsf{TM}^n_\Sigma$ that does nothing. Asperti and Ricciotti [2] define this machine directly:

Definition 6.1 (Nop) $\mathsf{Nop} := \Uparrow_{\mathsf{nil}} \mathsf{Null}$.

Note that because Null is a 0-tape machine, and Nop is supposed to be an n-tape machine, the index-vector must be the vector $\mathsf{nil} : \mathbb{F}_n^0$.

Lemma 6.2 (Correctness of Nop**)** $\mathsf{Nop} \vDash^0 \mathsf{NopRel}$ *with* $\mathsf{NopRel} := \lambda t\ t'.\ t' = t$.

Proof We apply the monotonicity Lemma 2.29 of $\vDash^{\cdot}$, the correctness Lemma 3.2 of Null, and the correctness Lemma 5.6 of the tapes-lift. It remains to show:

$$\big(\lambda t\ t'.\ \mathsf{NullRel}\ t\ t' \wedge \big(\forall i : \mathbb{F}_n.\ i \notin \mathsf{nil} \to t'[i] = t[i]\big)\big) \subseteq \mathsf{NopRel}$$

Let $t, t' : \mathsf{Tape}_\Sigma^n$. To show the equality $t' = t$ we show $t'[i] = t[i]$ for all $i : \mathbb{F}_n$. This follows with the equality part of the relation $\Uparrow_{\mathsf{nil}} \mathsf{NullRel}$, since $i \notin \mathsf{nil}$. □

Note that the correctness relation of Nop can also be expressed using the identity relation Id:

$$\mathsf{NopRel} \equiv \mathrm{Id}.$$

We have the convention to define relations of concrete machines in λ-notation, i.e. not using relational operators. Also note that the tape t' is, per convention, always on the left side of the equality. These conventions make rewriting of tapes uniform; therefore, rewriting of tapes can be automated in Coq.

6.2 WriteString

The machine WriteString d str writes a fixed string $str : \mathcal{L}(\Sigma)$ in the direction d. It is defined by recursion over the string:

Definition 6.3 (WriteString**)**

$$\begin{aligned}
&\mathsf{WriteString}\ d\ (\mathsf{nil}) &&:= \mathsf{Nop}\\
&\mathsf{WriteString}\ d\ (s :: \mathsf{nil}) &&:= \mathsf{Write}\ s\\
&\mathsf{WriteString}\ d\ (s :: str') &&:= \mathsf{WriteMove}\ s\ d;\ \mathsf{WriteString}\ d\ str'.
\end{aligned}$$

Note that this is our only machine we define per recursion. The way we prove correctness in constant time (depending on the length of str) is still the same.

The machine writes all symbols of the string str to the tape and moves in the tape in direction d after each (but the last) symbol. When it terminates, the head of the tape is under the last written symbol, which is the last symbol of str. It terminates in constant time, after $2 \cdot |str| - 1$ steps.

The derived relation for WriteString is also defined per recursion over the string:

Lemma 6.4 WriteString $d\ str \vDash^{2\cdot|str|-1} R'\ d\ str$ *with*

$$\begin{aligned} R'\ d\ (\mathsf{nil}) &:= \mathsf{NopRel} \\ R'\ d\ (s :: \mathsf{nil}) &:= \mathsf{DoActRel}(\lfloor s \rfloor, \mathsf{N}) \\ R'\ d\ (s :: str') &:= \mathsf{DoActRel}(\lfloor s \rfloor, d) \circ R'\ d\ str'. \end{aligned}$$

Proof By induction on $str : \mathcal{L}(\Sigma)$, using the monotonicity Lemma 2.29, the correctness of Nop (Lemma 6.2), the correctness of Write and WriteMove (which are defined using DoAct; Lemma 3.4), and correctness of sequential composition for constant time (Lemma 4.6). □

We define the actual relation of WriteString in terms of a function on tapes that is also defined by recursion over str:

Lemma 6.5 (Correctness of WriteString**)** *Let* $d : \mathsf{Move}$ *and* $str : \mathcal{L}(\Sigma)$.

$$\mathsf{WriteString}\ d\ str \vDash^{2\cdot|str|-1} \mathit{WriteStringRel}\ d\ str$$

with $\mathit{WriteStringRel}\ d\ str := \lambda t\ t'.\ t' = \mathit{writeStringFun}\ d\ t\ str$ *and*

$$\begin{aligned} \mathit{writeStringFun}\ d\ t\ (\mathsf{nil}) &:= t \\ \mathit{writeStringFun}\ d\ t\ (s :: \mathsf{nil}) &:= \mathsf{wr}\ t\ (\lfloor s \rfloor) \\ \mathit{writeStringFun}\ d\ t\ (s :: str') &:= \mathit{writeStringFun}\ d\ \big(\mathsf{doAct}\ t\ (\lfloor s \rfloor, d)\big)\ str' \end{aligned}$$

Proof We apply the monotonicity Lemma 2.29 and have to show:

$$R'\ d\ str \subseteq \mathit{WriteStringRel}\ d\ str$$

This can be shown by induction on str. □

Note that we could as well use the Mirror operator instead of parametrising the machine WritingString over the direction. In this particular example the parametrising approach seems to be easier.

6.3 MovePar

The two-tape machine MovePar $d_0\ d_1$ combines two Move machines. It first moves the 0th tape in direction d_0 and after that the 1st tape in direction d_1.[1]

Definition 6.6 (MovePar**)** $\mathsf{MovePar}\ d_0\ d_1 := \Uparrow_{[0]}(\mathsf{Move}\ d_0);\ \Uparrow_{[1]}(\mathsf{Move}\ d_1)$.

[1] To avoid confusion with zero-based indices used throughout this thesis, we write "the 0th or 1st tape", instead of "the first or second tape."

Lemma 6.7 (Correctness of MovePar**)** MovePar $d_0\ d_1 \vDash^3 \mathit{MovePairRel}\ d_0\ d_1$ *with*

$$\mathit{MovePairRel}\ d_0\ d_1 := \lambda t\ t'.\ t'[0] = \mathsf{mv}\ d_0\ t[0] \ \wedge\ t'[1] = \mathsf{mv}\ d_1\ t[1]$$

Proof We have to show:

$$\Uparrow_{[0]}(\mathit{DoActRel}(\emptyset, d_0)) \circ \Uparrow_{[1]}(\mathit{DoActRel}(\emptyset, d_1)) \subseteq \mathit{MovePairRel}\ d_0\ d_1$$

We assume tape vectors $t, t', t'' : \mathsf{Tape}_\Sigma^2$, such that $(\Uparrow_{[0]}(\mathit{DoActRel}(\emptyset, d_0)))\ t\ t'$ and $(\Uparrow_{[1]}(\mathit{DoActRel}(\emptyset, d_1)))\ t'\ t''$. We have to show $t''[0] = \mathsf{mv}\ d_0\ t[0]$ and $t''[1] = \mathsf{mv}\ d_1\ t[1]$. By definition, we know $t'[0] = \mathsf{mv}\ d_0\ t[0]$ and $t'[1] = t[1]$ (because $1 \notin [0]$). We also know $t''[1] = \mathsf{mv}\ d_1\ t'[1]$ and $t''[0] = t'[0]$ (because $0 \notin [1]$). The goal follows trivially. □

Note that this kind of proof is very mechanical: We only need to unfold the relations and rewrite tapes. Indeed, these steps are automated in Coq. Thus, we also do not present more proofs of this kind on paper.

6.4 CopySymbols

The machine CopySymbols $h : \mathsf{TM}_\Sigma^2$, where $h : \Sigma \to \mathbb{B}$, is a compound machine involving a While-loop. It reads a symbol on tape 0, writes it to tape 1, and moves both tapes to right, until the read symbol satisfies h. If there was no current symbol on tape 0, it also terminates.

We first define the machine for the step. Since we want to apply the While operator on the step machine, it must be labelled over $\mathcal{O}(1)$. $\lfloor() \rfloor$ means to break out of the loop and $\emptyset$ means to repeat the loop.

Definition 6.8 (CopySymbolsStep**)** CopySymbolsStep $h :=$

```
Switch (⇑[0] Read)
       (λ(s : O(Σ)).
          match s
          [ ⌊x⌋ ⇒
             if h(x)
             then  Return (⇑[1](Write x))  ⌊()⌋
             else  Return (⇑[1](Write x); MovePar R R) ∅
          | ∅   ⇒ Return Nop ⌊()⌋
          ])
```

Note that "match $[\cdots]$" denotes pattern matching our type theory.

Lemma 6.9 (Correctness of CopySymbolsStep**)**

$$\mathsf{CopySymbolsStep} \vDash^{7} \mathsf{CopySymbolsStepRel}$$

with

$$\mathsf{CopySymbolsStepRel} := \lambda t\ (l, t').\ \begin{cases} t'[0] = t[0] \wedge t'[1] = \mathsf{wr}\ t[1]\ \lfloor x \rfloor \wedge l = \lfloor () \rfloor & \mathsf{current}\ t[0] = \lfloor x \rfloor \wedge h(x) \\ t'[0] = \mathsf{mv}\ R\ t[0] \wedge t'[1] = \mathsf{doAct}\ t[1]\ (\lfloor x \rfloor, R) \wedge l = \emptyset & \mathsf{current}\ t[0] = \lfloor x \rfloor \wedge \neg h(x) \\ t' = t \wedge l = \lfloor () \rfloor & \textit{else} \end{cases}$$

Proof Mechanical, with case-analysis over $\mathsf{current}\ t[0]$. □

We define CopySymbol by applying the While operator to CopySymbolsStep:

Definition 6.10 (CopySymbols**)** $\mathsf{CopySymbols}\ h := \mathsf{While}(\mathsf{CopySymbolsStep}\ h)$.

The correctness of CopySymbols can be expressed using a recursive function on tapes:

Lemma 6.11 (Correctness of CopySymbols**)** $\mathsf{CopySymbols}\ h \vDash \mathsf{CopySymbolsRel}\ h$ *with* $\mathsf{CopySymbolsRel}\ h := \lambda t\ t'.\ t' = \mathsf{copySymbolsFun}\ h\ t$ *and*

$$\mathsf{copySymbolsFun}\ h\ t := \begin{cases} [t[0];\ \mathsf{wr}\ t[1]\ \lfloor x \rfloor] & \mathsf{current}\ t[0] = \lfloor x \rfloor \wedge h(x) \\ \mathsf{copySymbolsFun}\ h\ [\mathsf{mv}\ R\ t[0];\ \mathsf{doAct}\ t[1]\ (\lfloor x \rfloor, R)] & \mathsf{current}\ t[0] = \lfloor x \rfloor \wedge \neg h(x) \\ t & \mathsf{current}\ t[0] = \emptyset \end{cases}$$

Note that the function copySymbolsFun *is not structural recursive. It terminates because tapes have only finitely many symbols.*

Proof To show: $\mathit{WhileRel}\ \mathsf{CopySymbolsStepRel} \subseteq \mathsf{CopySymbolsRel}$. By While-induction (Lemma 4.22). □

We observe that the running time of CopySymbols only depends on the 0th tape. Therefore, we define a function $\mathsf{copySymbolsSteps} : \mathsf{Tape}_{\Sigma} \to \mathbb{N}$ that overestimates the number of steps needed for the loop, depending on the 0th tape. Note that While requires one additional step for each repeat of the loop.

Lemma 6.12 (Running time of CopySymbols**)** $\mathsf{CopySymbols} \downarrow \mathsf{CopySymbolsT}$ *with* $\mathsf{CopySymbolsT} := \lambda t\ k.\ \mathsf{copySymbolsSteps}(t) \leqslant k$ *and*

$$\mathsf{copySymbolsSteps}(t) := \begin{cases} 8 + \mathsf{copySymbolsSteps}(\mathsf{mv}\ R\ t) & \mathsf{current}\ t = \lfloor x \rfloor \wedge \neg h(x) \\ 8 & \textit{otherwise} \end{cases}$$

Proof We have to show $\mathrm{CopySymbolsT} \subseteq \mathit{WhileT}\ \mathrm{CopySymbolsStepRel}\ (\lambda\_\ k.\ 7 \leqslant k)$, using the co-induction Lemma 4.24. Let $\mathrm{copySymbolsSteps}\ t[0] \leqslant k$. We choose $k_1 := 7$. We have two cases.

1. We assume $\mathrm{CopySymbolsStepRel}\ t\ (\lfloor() \rfloor, t')$. Therefore, we know that either $\mathrm{current}\ t[0] = \lfloor x \rfloor$ with $h(x) = \mathrm{true}$, or $\mathrm{current}\ t[0] = \emptyset$. In both cases, we have $\mathrm{copySymbolsSteps}\ t[0] = 8$. Thus, we have:

$$k_1 \leqslant \mathrm{copySymbolsSteps}\ t[0] = 8 \leqslant k$$

2. We assume $\mathrm{CopySymbolsStepRel}\ t\ (\emptyset, t')$. Therefore, we have $\mathrm{current}\ t[0] = \lfloor x \rfloor$ with $h(x) = \mathrm{false}$, and $t'[0] = \mathrm{mv}\ R\ t[0]$. Then, we have:

$$1 + k_1 + \mathrm{copySymbolsSteps}\ t'[0] = \mathrm{copySymbolsSteps}\ t[0] \leqslant k \qquad \square$$

Using the Mirror operator, we can define a machine CopySymbolsL that copies and goes to the left instead. We also have to "mirror" the correctness relations and their respective functions. We do not repeat the definitions here.

Definition 6.13 (CopySymbolsL**)** CopySymbolsL h := Mirror(CopySymbols h).

6.5 MoveToSymbol

We can define a machine MoveToSymbol $h\ f : \mathrm{TM}^1_\Sigma$, where $h : \Sigma \to \mathbb{B}$ and $f : \Sigma \to \Sigma$. This machine behaves similar as CopySymbols h. Instead of copying the symbols from one tape to another tape, it "translates" the symbols it reads, until it reads a symbol that satisfies the boolean predicate h. We leave out the correctness and running time statements, as they can be derived from the statements about CopySymbols above.

Definition 6.14 (MoveToSymbol**)**

```
MoveToSymbolStep h f :=
  Switch (Read)
      (λ(s : 𝒪(Σ)). match s
          [ ⌊x⌋ ⇒
            if  h(x)
            then  Return (Write (f x)) ⌊()⌋
            else  Return (WriteMove (f x) R) ∅
          | ∅  ⇒ Return Nop ⌊()⌋
          ])

  MoveToSymbol h f := While(MoveToSymbolStep h f)
MoveToSymbolL h f := Mirror(MoveToSymbol h f)
```

Chapter 7

Generalised Register Machines

We can define Turing machines by combining sub-machines in an imperative style, where we use primitive machines like DoAct as the primitive instructions of our language for Turing machines. Furthermore, we can reuse machines in bigger contexts (w.r.t. number of tapes and symbols). Recall that we do not reason about internal states of machines anymore.

At this point, we want to abstract from concrete tapes. We think of tapes as *registers* that may either contain a value of an arbitrary encodable type, or may be resetted. Note that this notion of a "general register machine" is more general than usual register machines, because registers usually can only contain numbers.

When we speak of "programming" Turing machines, this means that instead of using primitive machines (like DoAct) we only use machines that directly change the value of a tape (like ConstrS, which increases a number). Using the definition of value-containment, we can also formalise a "callee-saving" convention for computation of functions. Furthermore, we show a general pattern how to program and verify Turing machines, and present more complex case studies in this chapter.

7.1 Value-Containment

We first want to define what it means that a tape t **contains** a value x, written as $t \simeq x$. Tapes, as defined in Definition 2.12, are essentially a list of symbols, so we have to linearise values to strings.

Definition 7.1 (Encodable types) *We say that a type X is encodable on a finite alphabet Σ_X, if there is a function* $\mathtt{encode} : X \to \mathcal{L}(\Sigma_X)$.

Morally, the encoding function should be injective. There should also be a decoding function, such that the pair $(\mathtt{encode}, \mathtt{decode})$ is a retraction on $\mathcal{L}(\Sigma)$. As we do not need any of these facts, we leave them out of this definition.

We can map encodings with retractions:

Definition 7.2 (Map encodings) *Let X be encodable on Σ and $f : \Sigma \hookrightarrow \Gamma$ be a retraction. Then X is also encodable on Γ with the following encoding function:*

$$\mathit{encodeMap}\ \mathit{encode}_\Sigma\ f\ (x) := \mathsf{map}\ f\ (\mathit{encode}_\Sigma(x)).$$

Mapping of encodings are compatible with composition of retractions. This means that if we map the encoding twice, this is the same as mapping the encoding with the composition of both retractions:

Lemma 7.3 (Composition and encoding mapping) *Let X be encodable on Σ with the function $\mathit{encode}_\Sigma : X \to \mathcal{L}(\Sigma)$. Let $f : \Sigma \hookrightarrow \Gamma$ and $g : \Gamma \hookrightarrow \Delta$ be retractions. Then there is (extensionally) only one canonical way how X can be encoded on Δ, i.e.:*

$$\mathit{encodeMap}\ (\mathit{encodeMap}\ f\ \mathit{encode}_\Sigma)\ g\ x = \mathit{encodeMap}\ \mathit{encode}_\Sigma\ (g \circ f)\ x$$

For every following type X we define an alphabet Σ_X to encode X on. If X is an type constructor and Y is encoded on Σ_Y, we define $\Sigma_{X(Y)}$ as the alphabet for $X(Y)$:

Definition 7.4 (Basic encodings)

1. *We encode 1 on the empty alphabet $\Sigma_1 := \bot$ with $\mathit{encode}\ () := \mathsf{nil}$.*
2. *The type $\mathbb{B}$ is encoded on itself, i.e. $\Sigma_\mathbb{B} := \mathbb{B}$ and $\mathit{encode}(b) := [b]$.*
3. *Let X be encodable on Σ_X. Then $\mathcal{O}(X)$ is encodable on*

   $$\Sigma_{\mathcal{O}(X)} ::= \mathsf{NONE} \mid \mathsf{SOME} \mid (x : \Sigma_X)$$

   With the retraction $\mathtt{RetrOpt} : \Sigma_X \hookrightarrow \Sigma_{\mathcal{O}(X)}$ and

   $$\begin{aligned} \mathit{encode}\ \emptyset &:= [\mathsf{NONE}] \\ \mathit{encode}\ \lfloor x \rfloor &:= \mathsf{SOME} :: \mathit{encode}(x) \end{aligned}$$

4. *Let X be encodable on Σ_X and Y on Σ_Y. Then $X + Y$ is encodable on*

   $$\Sigma_{X+Y} ::= \mathsf{INL} \mid \mathsf{INR} \mid (x : \Sigma_X) \mid (y : \Sigma_Y)$$

   With the retractions $\mathtt{RetrInl} : \Sigma_X \hookrightarrow \Sigma_{X+Y}$, $\mathtt{RetInr} : \Sigma_Y \hookrightarrow \Sigma_{X+Y}$ and

   $$\begin{aligned} \mathit{encode}\ (\mathsf{inl}\, x) &:= \mathsf{INL} :: \mathit{encode}(x) \\ \mathit{encode}\ (\mathsf{inr}\, y) &:= \mathsf{INR} :: \mathit{encode}(y) \end{aligned}$$

5. *Let* X *be encodable on* Σ_X *and* Y *on* Σ_Y. *Then* $X \times Y$ *is encodable on*

$$\Sigma_{X\times Y} ::= (x : \Sigma_X) \mid (y : \Sigma_Y)$$

With the retractions $\mathtt{RetrFst} : \Sigma_X \hookrightarrow \Sigma_{X\times Y}$, $\mathtt{RetrSnd} : \Sigma_Y \hookrightarrow \Sigma_{X\times Y}$ *and*

$$\mathtt{encode}(x, y) := \mathtt{encode}(x) +\!\!+ \mathtt{encode}(y)$$

6. *Let* X *be encodable on* Σ_X. *Then* $\mathcal{L}(X)$ *is encodable on*

$$\Sigma_{\mathcal{L}(X)} ::= \mathsf{NIL} \mid \mathsf{CONS} \mid (x : \Sigma_X)$$

With the retraction $\mathtt{RetrList} : \Sigma_X \hookrightarrow \Sigma_{\mathcal{L}(X)}$ *and*

$$\begin{aligned}\mathtt{encode}\ (\mathsf{nil}) \quad &:= [\mathsf{NIL}]\\ \mathtt{encode}\ (x :: ls) &:= \mathsf{CONS} :: \mathtt{encode}(x) +\!\!+ \mathtt{encode}(ls)\end{aligned}$$

7. *Natural numbers* $\mathbb{N}$ *are encodable on* $\Sigma_{\mathbb{N}} ::= \mathsf{S} \mid \mathsf{0}$ *with*

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

Note that we implicitly map the encoding functions in this definition. Also note that $\mathtt{RetrLft} : X \hookrightarrow X + Y$ *is similar to* $\mathtt{RetrInl} : \Sigma_X \hookrightarrow \Sigma_{X+Y}$.

This are all encodings we need in this thesis. We say that X is *minimally encodable* on Σ_X, if X is encodable on Σ_X according to Definition 7.4. Mapping of encodings introduces ambiguity: It is not only possible to encode multiple types on the same alphabet, but there may also be several ways how to encode the same type on the same alphabet. For example, the minimal alphabet of the type $X + X$ is Σ_{X+X}. Although there is only one way to encode $X + X$ on Σ_{X+X}, there are two possibilities how to encode X: using the retraction $\mathtt{RetrInl}$ or $\mathtt{RetrInr}$. We must deal with this problem, when we program and verify Turing machines, by explicitly specifying the right retractions.

If X is encodable on Σ, we encode values of X on tapes with an extended alphabet Σ^+ that has an additional start and stop symbol.

Definition 7.5 (Σ^+) *Let* $\Sigma : \mathbb{T}$. *Then* $\Sigma^+ ::= \mathsf{START} \mid \mathsf{STOP} \mid \mathsf{UNKNOWN} \mid (s : \Sigma)$. *Also, we define the retraction* $\mathtt{RetrPlus} : \Sigma \hookrightarrow \Sigma^+$.

The UNKNOWN symbol is important later. We define what $t \simeq x$ means:

Definition 7.6 ($t \simeq x$) *Let* X *be encodable on* Σ *and* $t : \mathsf{Tape}_{\Sigma^+}$.

$$t \simeq x := \exists\, ls.\ t = \mathsf{midtape}\ ls\ \mathsf{START}\ (encode(x) +\!\!+ [\mathsf{STOP}])$$

In case the encoding is not clear, we write $t \simeq_f x$ *with* $t : \mathsf{Tape}_{\Gamma^+}$ *and* $x : X$, *when* X *is minimally encodable on* Σ_X *and* $f : \Sigma_X \hookrightarrow \Gamma$.

Note that in Definition 7.6, we have to map the encoding of x to the extended alphabet Σ^+ using $\mathtt{RetrPlus}$. If $t \simeq x$, the head of t stands on the start symbol. To the right, there is the encoding, which is terminated by one stop-symbol. To the left of the head, there may be arbitrarily many "rest" symbols.

We find the convention useful that there are no further symbols beyond the stop symbol. In future work we also want to reason about space usage of machines. If a tape contains a value, the size of the tape (i.e. the number of symbols on it), only depends on the length of the encoding and the number of rest symbols ls to the left. To avoid data leaks, we have the convention to only add symbols on the left side. We also only write on tapes, if the head of the tape is on the right-most symbol.

Definition 7.7 (Right tape) $\mathtt{isRight}(t) := \exists m\ rs.\ t = \mathsf{midtape}\ ls\ m\ \mathsf{nil}$. *With other words, a tape* t *is right, if and only if* $\mathsf{current}(t) \neq \emptyset \wedge \mathsf{right}(t) = \mathsf{nil}$.

7.2 Alphabet-Lifting

In Section 5.2, we defined the alphabet-lift operator. We noted that we need a semantically *relevant* default symbol in the smaller alphabet Σ. All "programmed" Turing machines are defined on an extended alphabet Σ^+. It seems therefore useful to introduce a uniform default symbol $\mathsf{UNKNOWN}$ to Σ^+. Intuitively, nothing can go wrong, because, by definition, $\mathsf{UNKNOWN}$ is not part of any encoding. This intuition is confirmed by the following lemma:

Lemma 7.8 ($\mathtt{surject}$ and value-containment) *Let* $f : \Sigma \hookrightarrow \Gamma$ *be a retraction. Let* $x : X$, *where* X *is encodable on* Σ, *and* $t : \mathsf{Tape}_{\Gamma^+}$:

$$\mathtt{mapTape}\ (\mathtt{surject}\ (\mathtt{RetrPlus} \circ f)\ \mathsf{UNKNOWN})\ t \simeq x \ \leftrightarrow\ t \simeq_f x$$

Note that the encoding on the right side is mapped with the retraction f.

We write $\Uparrow_f M$ for $\Uparrow_{(\mathtt{RetrPlus} \circ f, \mathsf{UNKNOWN})} M$. Note that we also use the $\Uparrow$ notation for the tapes-lift, but it is always clear whether we mean the tape-lift or alphabet-lift. We use the notation $\Uparrow_{f;\, I} M := \Uparrow_I (\Uparrow_f M)$, for first applying the alphabet-lift, and after that the tapes-lift.

7.3 Value-Manipulating Machines

We have defined what it means that a tape contains a value. Now we want to define machines that manipulate values, e.g. increase or decrease a number. Another very useful operation is to copy a value from one tape to another tape.

7.3.1 Write Value

We define a wrapper around the machine WriteString from Section 6.2. We parametrise WriteValue over the encoding $\mathtt{str} : \mathcal{L}(\Sigma_X)$ of a value $x : X$.[1]

Definition 7.9 (WriteValue**)** *Let* $\mathtt{str} : \mathcal{L}(\Sigma)$

$$\mathsf{WriteValue}(\mathtt{str}) := \mathsf{WriteString\ L}\ (\mathsf{rev}\ (\mathsf{START} :: \mathsf{map}\ \mathtt{RetrPlus}\ \mathtt{str} +\!\!+ [\mathsf{STOP}]))$$

The machine writes the stop symbol, the encoding $\mathtt{str}$, and the start symbol in reversed order from right to left. After the execution of the machine, the tape contains the value x with the encoding $\mathtt{str}$, under the precondition that the tape was initially right.

Lemma 7.10 (Correctness of WriteValue**)** *Let* X *be encodable over* Σ_X *and* $\mathtt{str} : \mathcal{L}(\Sigma)$.

$$\mathsf{WriteValue}(\mathtt{str}) \vDash^{3+2\cdot|\mathtt{str}|} \mathit{WriteValueRel}(\mathtt{str})$$

$$\mathit{WriteValueRel}(\mathtt{str}) := \lambda t\ t'.\ \forall (x : X).\ \mathtt{encode}(x) = \mathtt{str} \to \mathtt{isRight}\ t[0] \to t'[0] \simeq x.$$

7.3.2 Constructor and Deconstructor Machines

For each encodable type X, there is a set of machines: constructor machines and one deconstructor machine. These machines are defined on the minimal alphabet Σ_X^+. Constructor machines, informally, "apply" a constructor of the inductive type X to the value on the tape $t[0]$. In general, we have one constructor machine for each constructor of the inductive type. If the constructor has additional arguments, these are encoded on more input tapes $t[1], t[2], \cdots$. There is only one deconstructor machine for each type. The deconstructor machine for X has one label for every constructor of X. It makes a case-distinction over the value of the tape, and it terminates in the label that corresponds to the constructor. Deconstructor machines may have additional tapes for storing additional arguments of constructors.

[1] The reason why WriteValue is not parametrised over values $x : X$ is, that we would also need to parametrise the machine over the encoding function of X. With this approach, only the correctness Lemma 7.10 and the correctness relation are parametrised over the encoding.

Natural numbers

We consider the inductive type $\mathbb{N}$, that has two constructors: S and 0. The constructor machine ConstrO writes the number 0 on a right tape. The constructor machine ConstrS assumes that $t[0]$ contains a number n, and increases that number.

We define the 0-constructor with WriteValue:

Definition 7.11 (ConstrO**)** ConstrO := WriteValue [O].

Lemma 7.12 (Correctness of ConstrO**)** ConstrO $\vDash^5$ $\mathtt{ConstrORel}$ *with*

$$\mathtt{ConstrORel} := \lambda t\ t'.\ \mathtt{isRight}\ t[0] \to t'[0] \simeq 0$$

Proof With Lemma 2.29 and 7.10. □

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

Definition 7.13 (ConstrS**)** ConstrS := WriteMove S L; Write START

Lemma 7.14 (Correctness of ConstrS**)** ConstrS $\vDash^3$ $\mathtt{ConstrSRel}$ *with*

$$\mathtt{ConstrSRel} := \lambda t\ t'.\ \forall n.\ t[0] \simeq n \to t'[0] \simeq S\ n.$$

CaseNat is the deconstructor machine for $\mathbb{N}$. It reads a number from tape $t[0]$. If it is 0, CaseNat leaves the number unchanged and terminates in the label false. Else, it decreases the number and terminates in true.

Initially, the tape $t[0]$ contains some number n. Thus, the head of $t[0]$ must be on the start symbol. The machine moves one step right and reads the first symbol of the encoding. This symbol either is S or O. If the machine read the symbol O, this means that the number n is 0, so the machine moves back to the start symbol and terminates in the label false, indicating that the number was 0. On the other side, if the machine read S, this means the number n is the successor of some number n'. To decrement n, it overwrites the current S with a new start symbol and terminates on this symbol in the label true.

Definition 7.15 (CaseNat**)**

```
CaseNat :=
  Move R;
  Switch Read
      (λ(s : 𝒪(Σ⁺_ℕ)). match s
          [ ⌊S⌋ ⇒ Return (Write START) true
          | ⌊O⌋ ⇒ Return (Move L) false
          | _ ⇒ _
          ])
```

Note, that the placeholder "_" stands for some unspecified machine, e.g. in this case Return Nop b where $b : \mathbb{B}$ can be chosen arbitrarily. We do not give this part explicitly here, because the premise of the correctness relation of CaseNat guaranties that there must be a symbol under the head that is either S or O, so this part of the machine must never execute.

Lemma 7.16 (Correctness of CaseNat**)** CaseNat $\models^5 CaseNatRel$ *with*

$CaseNatRel :=$
$\lambda t\ (l, t').\ \forall(n : \mathbb{N}).\ t[0] \simeq n \rightarrow$
match l, n
[false, $0 \Rightarrow t'[0] \simeq 0$
| true, S $n' \Rightarrow t'[0] \simeq n'$
| _ , _ $\Rightarrow \bot$
]

Proof We know that $t[0] \simeq n$. This means $t[0] =$ midtape ls START ($S^n + O :: STOP$) for some ls. The proof is mechanical, like the above correctness proofs. It makes a case-distinction over n. □

Note that for the case that n is 0, we could also write $t' = t$ in the correctness relation. But we have the following convention: When some tape does not change, we use the weaker postcondition that the tape t' still contains the same values (and that the same tapes are right). This convention has the advantage that when we apply the alphabet-lift with a retraction f, we would otherwise get assumptions in correctness proofs like:

$$mapTapes\ (surject\ (RetrPlus \circ f)\ \mathsf{UNKNOWN})\ t =$$
$$mapTapes\ (surject\ (RetrPlus \circ f)\ \mathsf{UNKNOWN})\ t'$$

To show that $t'[i]$ contains the same value as $t[i]$, we would have to apply a lemma. Using this convention we get this for free.

Sum Types

Let X be encodable on Σ_X and Y on Σ_Y. CaseSum : $\mathsf{TM}^1_{\Sigma^+_{X+Y}}(\mathbb{B})$ reads a value $s : X + Y$. If it is inl x (or inr l), it replaces the INL (or INR) symbol with a new start symbol and terminates in true (or false).

Definition 7.17 (CaseSum**)**

CaseSum :=
Move R;
Switch Read

$(\lambda(s : \mathcal{O}(\Sigma^+_{X+Y}))$. match s
$[\lfloor \mathsf{INL} \rfloor \Rightarrow$ Return (Write START) true
$|\lfloor \mathsf{INR} \rfloor \Rightarrow$ Return (Write START) false
$|\_ \Rightarrow \_$
$])$

Lemma 7.18 (Correctness of CaseSum**)** CaseSum $\vDash^5$ $\mathit{CaseSumRel}$ *with*

$\mathit{CaseSumRel} :=$
$\lambda t\ (l, t').\ \forall(s : X + Y).\ t[0] \simeq s \to$ match l, s
$[$ false , inl $x \Rightarrow t'[0] \simeq x$
$|$ true , inr $y \Rightarrow t'[0] \simeq y$
$|\_ , \_ \Rightarrow \bot$
$]$

We have one constructor machine for inl and inr, respectively. They are both analogous. They overwrite the start symbol with INL or INR and write a new start symbol one step further left.

Definition 7.19 (ConstrInl**)** ConstrInl := WriteMove INL L; Write START.

Lemma 7.20 (Correctness of ConstrInl**)** ConstrInl $\vDash^3$ $\mathit{ConstrInlRel}$ *with*

$$\mathit{ConstrInlRel} := \lambda t\ t'.\ \forall(x : X).\ t[0] \simeq x \to t'[0] \simeq \mathsf{inl}\, x$$

Option Types

The types $\mathcal{O}(X)$ and $X + 1$ are isomorphic. Furthermore, the alphabets to encode these types on, $\Sigma_{\mathcal{O}(X)}$ and Σ_{X+1}, are also isomorphic. We can use this fact to derive a deconstructor machine and constructor machines for the type $\mathcal{O}(X)$. Let $f : \Sigma_{X+1} \hookrightarrow \Sigma_{\mathcal{O}(X)}$ be the canonical retraction. Then, we can define CaseOption as $\Uparrow_f$ CaseSum. However, in the case that $o = \emptyset$, the output tape t' contains $()$. Thus, the tape has form $t' =$ midtape ls START [STOP] (for some ls). However, we want that the tape is right in this case. That is, because $\mathcal{O}(X)$ is not a recursive data type and the information that $o = \emptyset$ is already encoded in the label of the state in which CaseOption terminates. So in this case, the tape moves one step to the right:

Definition 7.21 (CaseOption**)**

CaseOption := If ($\Uparrow_f$ CaseSum) Then (Return Nop true) Else (Return (Move R) false)

Lemma 7.22 (Correctness of CaseOption**)** CaseOption $\vDash^7$ $\mathit{CaseOptionRel}$ *with*

$$
\begin{array}{l}
\mathtt{CaseOptionRel} := \\
\quad \lambda t\ (l, t').\ \ \forall (o : \mathcal{O}(X)).\ \ t[0] \simeq o \to \mathsf{match}\ l, o \\
\qquad [\,\mathsf{false}\,,\ \ \emptyset\ \ \Rightarrow \mathtt{isRight}(t'[0]) \\
\qquad |\,\mathsf{true}\,,\ \ \lfloor x \rfloor \Rightarrow t'[0] \simeq x \\
\qquad |\,\_\,,\_ \Rightarrow \bot \\
\qquad]
\end{array}
$$

Note that we have to add 2 extra steps: one for the conditional and one for the Move in the $\emptyset$-case.

We can also derive the $\lfloor\cdot\rfloor$ constructor of $\mathcal{O}(X)$ from the inl constructor of $X + 1$.

Definition 7.23 (ConstrSome**)** ConstrSome $:= \Uparrow_f$ ConstrInl.

Lemma 7.24 (Correctness of ConstrSome**)** ConstrSome $\vDash^3$ $\mathtt{ConstrSomeRel}$ *with*

$$\mathtt{ConstrSomeRel} := \lambda t\ t'.\ \forall (x : X).\ t[0] \simeq x \to t'[0] \simeq \lfloor x \rfloor$$

For the $\emptyset$ constructor, we simply use WriteValue.

Definition 7.25 (ConstrNone**)** ConstrNone := WriteValue [NONE].

Lemma 7.26 (Correctness of ConstrNone**)** ConstrNone $\vDash^3$ $\mathtt{ConstrNoneRel}$ *with*

$$\mathtt{ConstrNoneRel} := \lambda t\ t'.\ \mathtt{isRight}\ t[0] \to t'[0] \simeq \emptyset$$

Product Types

The deconstructor and constructor machines for product types have to explicitly copy parts of values. Thus, their running time depends on the value. We use the machines MoveToSymbol and CopySymbols.

Let X and Y be encodable on Σ_X and Σ_Y. The deconstructor CasePair : $\mathsf{TM}^2_{\Sigma^+_{X\times Y}}$ copies the first component of the pair $(x, y) : X \times Y$ to tape 1, the second component remains on tape 0. The machine works as follows: first it writes the stop symbol on tape 1. Then, it seeks the last character of the encoding of x on tape 0, copies the encoding of x from right to left to tape 1, including the start symbol. At this point, tape 0 contains (x, y) and tape 1 contains x. Then it again moves tape 0 to last symbol of the encoding of x on tape 0 and overwrites it with a new start symbol.

Definition 7.27 (CasePair**)**

$$
\begin{array}{rl}
\mathsf{CasePair} := & \Uparrow_{[1]}(\mathsf{WriteMove\ STOP\ L}); \\
& \Uparrow_{[0]}(\mathsf{MoveToSymbol\ f\ id;\ Move\ L}); \\
& \mathsf{CopySymbolsL\ g}; \\
& \Uparrow_{[0]}(\mathsf{MoveToSymbol\ f\ id;\ Move\ L;\ Write\ START})
\end{array}
$$

Where $f : \Sigma^+_{X\times Y} \to \mathbb{B}$ *is true for symbols that are mapped from the alphabet* Σ_Y. $g(s)$ *is* true *if and only if* $s = \mathsf{START}$.

The correctness and termination proofs of CasePair are quite technical. We have lemmas about the tape-functions of CopyValueL, MoveToSymbol, and the respective running time functions. However, it is interesting to note that we have to do a case-distinction, whether y is empty. If it is empty, then the first MoveToSymbol f id moves to the stop symbol. If it is not empty, it moves to the first symbol of the encoding of y. In both cases, Move L moves to the last symbol of x (or to the start symbol, if x is empty).

Lemma 7.28 (Correctness of CasePair**)** CasePair $\vDash \mathit{CasePairRel}$ *with*

$$\mathit{CasePairRel} := \lambda t\ t'.\ \forall(p : X \times Y).\ t[0] \simeq p \to \mathsf{isRight}\ t[1] \to t'[0] \simeq \pi_2(p) \wedge t'[1] \simeq \pi_1(p)$$

Lemma 7.29 (Running time of CasePair**)** CasePair $\downarrow \mathit{CasePairT}$ *with*

$$\mathit{CasePairT} := \lambda t\ k.\ \exists(p : X \times Y).\ t[0] \simeq p \wedge 34 + 16 \cdot |\mathit{encode}(\pi_1(p))| \leqslant k$$

The constructor copies the first component x from tape 0 to tape 1, which initially contains the second component y, from right to left. By that, it overwrites the start symbol on tape 1 with the last symbol of the encoding of x.

Definition 7.30 (ConstrPair**)**

$$\mathsf{ConstrPair} := \Uparrow_{[0]}(\mathsf{MoveToSymbol}\ h\ \mathrm{id});\ \mathsf{CopySymbolsL}\ g$$

Where g *is the same function as in Definition 7.27.* $h(s)$ *is* true *if and only if* $s = \mathsf{STOP}$.

Lemma 7.31 (Correctness of ConstrPair**)** ConstrPair $\vDash \mathit{ConstrPairRel}$ *with*

$$\mathit{ConstrPairRel} := \lambda t\ t'.\ \forall(x : X)\ (y : Y).\ t[0] \simeq x \to t[1] \simeq y \to t'[0] \simeq x \wedge t'[1] \simeq (x, y)$$

Lemma 7.32 (Running time of ConstrPair**)** ConstrPair $\downarrow \mathit{ConstrPairT}$ *with*

$$\mathit{ConstrPairT} := \lambda t\ k.\ \exists(x : X).\ t[0] \simeq x \wedge 19 + 12 \cdot |\mathit{encode}(x)| \leqslant k$$

List Types

The definition of CaseList is quite complex. For brevity we only state the correctness statements of the match-machine here. The machine CaseList : $\mathsf{TM}^2_{\Sigma^+_{\mathcal{L}(X)}}(\mathbb{B})$ expects a list on tape 0. If it is nil, it terminates in the label false and leaves the tapes unchanged. If the list is $x :: xs$, it moves x to tape 1 and terminates in the label true.

Lemma 7.33 (Correctness of CaseList**)** CaseList $\vDash \mathit{CaseListRel}$ *with*

$$
\begin{array}{l}
\mathtt{CaseListRel} := \\
\quad \lambda t\ (l, t').\ \forall (xs : \mathcal{L}(X)).\ t[0] \simeq xs \to \mathsf{isRight}\ t[1] \to \mathsf{match}\ l, xs \\
\qquad [\,\mathsf{false}, \mathsf{nil} \ \ \Rightarrow t'[0] \simeq \mathsf{nil} \wedge \mathsf{isRight}\ t'[1] \\
\qquad |\,\mathsf{true}, x :: xs' \ \ \Rightarrow t'[0] \simeq xs' \wedge t'[1] \simeq x \\
\qquad |\,\_, \_ \ \ \Rightarrow \ \ \bot \\
\qquad]\ .
\end{array}
$$

The nil constructor is defined using WriteValue.

Definition 7.34 (ConstrNil**)** ConstrNil := WriteValue [NIL].

Lemma 7.35 (Correctness of ConstrNil**)** ConstrNil $\vDash^5$ ConstrNilRel *with*

$$\mathtt{ConstrNilRel} := \lambda t\ t.\ \mathsf{isRight}\ t[0] \to t'[0] \simeq \mathsf{nil}$$

The "::" constructor machine ConstrCons expects a list on tape 0 and a value $x : X$ on tape 1. It moves the head of tape 1 to the last symbol of the encoding of x and copies them on tape 0 from right to left. Thereby, it overwrites the current start symbol on tape 1, but also copies the start symbol from tape 1 to tape 0. After that, the machine overwrites this start symbol with CONS and writes a new start symbol on tape 0.

Definition 7.36 (ConstrCons**)**

$$
\begin{array}{rl}
\mathsf{ConstrCons} := & \Uparrow_{[1]}(\mathsf{MoveToSymbol}\ h\ \mathtt{id};\ \mathsf{Move\ L}); \\
& \Uparrow_{[1;0]}(\mathsf{CopySymbolsL}\ h'); \\
& \Uparrow_{[0]}(\mathsf{WriteMove\ CONS\ L};\ \mathsf{Write\ START})
\end{array}
$$

With $h, h' : \Sigma^+_{\mathtt{List}(X)} \to \mathbb{B}$ *such that* $h(s) = \mathsf{true}$ *if and only if* $s = \mathsf{STOP}$*, and* $h'(s) = \mathsf{false}$ *if and only if* s *is mapped from the alphabet* Σ_X*.*

Lemma 7.37 (Correctness of ConstrCons**)** ConstrCons $\vDash$ ConstrConsRel *with*

$$
\begin{array}{l}
\mathtt{ConstrConRel} := \\
\quad \lambda t\ t'.\ \forall (xs : \mathcal{L}(X))\ (x : X).\ t[0] \simeq xs \to t[1] \simeq x \to t'[0] \simeq x :: xs \wedge t'[1] \simeq x.
\end{array}
$$

Finite Types

All finite types Σ can be encoded on themself, i.e. $\mathtt{encode}(x) = [x]$ for $x : \Sigma$. The deconstructor machine for finite types moves the head from the starting symbol to the symbol and reads it. After that, it moves the head one further so that the tape is right and terminates in the label corresponding to the read symbol.

Definition 7.38 (CaseFin**)** *Let* $\mathtt{def} : \Sigma$.

$$\begin{aligned}&\mathsf{CaseFin} :=\\ &\quad \mathsf{Move\ R;\ Switch\ Read}\,(\lambda s.\ \mathsf{match}\ s\ [\,\lfloor x\rfloor \Rightarrow \mathsf{Return}\ (\mathsf{Move\ R})\ x \mid \_ \Rightarrow \mathsf{Return\ Nop}\ \mathtt{def}\,])\end{aligned}$$

Lemma 7.39 (Correctness of CaseFin**)** $\mathsf{CaseFin} \vDash^5 \mathtt{CaseFinRel}$ *with*

$$\mathtt{CaseFinRel} := \lambda t\ (l, t').\ \forall (x : \Sigma).\ t[0] \simeq x \to \mathtt{isRight}\ t'[0]\ \wedge\ l = x$$

7.3.3 Copy Values

In Section 6.4, we defined a machine that copies symbols until a certain symbol from one tape to another tape. In Section 6.5, we also defined a machine that moves the head to a certain symbol. We want to use these machines to define a machine that copies a value from tape $t[0]$ on tape $t[1]$. By the convention we set up in Section 7.1, if we write new symbols on a tape, this tape must be right. With other words, the machine CopySymbols can assume that the "target" tape is right. Formally, the correctness relation is defined as:

Definition 7.40 (Correctness relation of CopyValue**)**

$$\mathtt{CopyValueRel} := \lambda t\ t'.\ \forall (x : X).\ t[0] \simeq x \to \mathtt{isRight}\ t[1] \to t'[0] \simeq x \wedge t'[1] \simeq x$$

The algorithm of CopyValue works as follows. First, the machine moves the head of tape 0 right, from the start symbol to the stop symbol. Then it copies the symbols, from the stop symbol to the start symbol, to tape 1.

Definition 7.41 (CopyValue**)**

$$\mathsf{CopyValue} := \Uparrow_{[0]}(\mathsf{MoveToSymbol}\ (\lambda x.\ x = \mathsf{STOP})\ \ \mathtt{id});\ \mathsf{CopySymbolsL}\ (\lambda x.\ x = \mathsf{START})$$

Note that the second parameter of MoveToSymbol is the translation function. In this case, the machine should simply move the head and leave the symbols unchanged, so we choose the identity function for the parameter.

Lemma 7.42 (Correctness of CopyValue**)** $\mathsf{CopyValue} \vDash \mathtt{CopyValueRel}$.

The running time of CopyValue is linear in the size of the encoding of x.

Lemma 7.43 (Running time of CopyValue**)**

$$\mathsf{CopyValue} \downarrow (\lambda t\ k.\ \exists (x : X).\ t[0] \simeq x \wedge 25 + 12 \cdot |\mathtt{encode}(x)| \leqslant k)$$

Although CopyValue terminates for all tapes, we restricted running time relations to reasonable tapes, i.e. in this case tape vectors t that actually have a symbol on $t[0]$. The reason for that is that we want to relate the value to the number of steps.

7.3.4 Translate Values

In Section 7.1, we observed that the containment relation is ambiguous: if X is minimally encodable on Σ_X, there may be two retractions $f_1, f_2 : \Sigma_X \hookrightarrow \Gamma$ on the alphabet Γ. In Section 6.5, we gave the MoveToSymbol machine the feature to translate the symbols it reads. Using this feature, we define a machine that translates between the two possibilities of the encoding of X on Γ. For that, we have to define a function $\mathit{translate}\ f_1\ f_2 : \Gamma^+ \to \Gamma^+$.

Definition 7.44 (Translation between two encodings) *Let $f_1, f_2 : \Sigma \hookrightarrow \Gamma$ be two retractions on Γ.*

$$\mathit{translate}\ f_1\ f_2\ (\tau) := \tau \qquad (\textit{if } \tau \in \{\mathsf{UNKNOWN}, \mathsf{START}, \mathsf{STOP}\})$$

$$\mathit{translate}\ f_1\ f_2\ (\tau) := \begin{cases} f_2(\sigma) & f_1^{-1}(\tau) = \lfloor \sigma \rfloor \\ \mathsf{UNKNOWN} & f_1^{-1}(\tau) = \emptyset \end{cases}$$

This function translates a symbol of Γ to Σ, using the partial inversion function f_1^{-1} and afterwards applies the injection f_2 from Σ to Γ and the (implicit) injection from Γ to Γ^+.

Definition 7.45 (Translate**)**

$$\begin{aligned}\mathsf{Translate}\ f_1\ f_2 := \mathsf{MoveToSymbol}\ (\lambda x.\ x = \mathsf{STOP})\ (\mathit{translate}\ f_1\ f_2);\\ \mathsf{MoveToSymbolL}\ (\lambda x.\ x = \mathsf{START})\ \mathsf{id}\end{aligned}$$

Lemma 7.46 (Correctness of Translate**)** $\mathsf{Translate}\ f_1\ f_2 \vDash \mathit{TranslateRel}\ f_1\ f_2$ *with*

$$\mathit{TranslateRel}\ f_1\ f_2 := \lambda t\ t'.\ \forall (x : X).\ t[0] \simeq_{f_1} x \to t[0] \simeq_{f_2} x$$

7.3.5 Reset Tapes

The aforementioned machine $\mathsf{MoveToSymbol}\ (\lambda x.\ x = \mathsf{STOP})\ \mathsf{id}$ can also be used to "reset" a tape. This means that if the tape contains some value, after the execution of this machine the tape is right. This is especially important to prevent memory leaks in loops, where we may want to write on a tape multiple times. We must always reset every "used" tape before writing on it again.

Definition 7.47 (ResetTape**)** $\mathsf{ResetTape} := \mathsf{MoveToSymbol}\ (\lambda x.\ x = \mathsf{STOP})\ \mathsf{id}$

Lemma 7.48 (Correctness of ResetTape**)** $\mathsf{ResetTape} \vDash \mathit{ResetTapeRel}$ *with*

$$\mathit{ResetTapeRel} := \lambda t\ t'.\ \forall (x : X).\ t[0] \simeq x \to \mathsf{isRight}(t'[0])$$

7.4 Extending Alphabets

Consider that we have machines $M_1 : \mathsf{TM}^n_{\Sigma^+_X}$ and $M_2 : \mathsf{TM}^n_{\Sigma^+_Y}$ that operate on values of type X and Y, where X is encodable on Σ_X and Y on Σ_Y. We could combine these machines using the alphabet-lift, to get a machine $M : \mathsf{TM}^n_{\Sigma^+}$. The choice of Σ is relevant. For example, we could choose $\Sigma := \Sigma_X + \Sigma_Y$. Then, if we want to use M for another machine $M' : \mathsf{TM}^n_{\Gamma^+}$, we have to give a retraction $f : \Sigma_X + \Sigma_Y \hookrightarrow \Gamma$. However, there is a problem of generality:

Fact 7.49 *There exist types X, Y, Z with retractions $f_1 : X \hookrightarrow Z$ and $f_2 : Y \hookrightarrow Z$, such that there is no retraction $(f_1 + f_2) : X + Y \hookrightarrow Z$.*

Proof There is no injective function $1 + 1 \to 1$. □

That means that we would have to show a precondition to combine two retractions $f_1 : \Sigma_X \hookrightarrow \Gamma$ and $f_2 : \Sigma_Y \hookrightarrow \Gamma$ and get a retraction $(f_1 + f_2) : \Sigma_X + \Sigma_Y \hookrightarrow \Gamma$:

Fact 7.50 *If the retractions $f_1 : X \hookrightarrow Z$ and $f_2 : Y \hookrightarrow Z$ have disjoint images in Z, then we can define a retraction $(f_1 + f_2) : X + Y \hookrightarrow Z$.*

We could live with the restriction that we have to show that the retractions have disjoint images. However, this approach does not scale good when we have to add more alphabets and retractions, we would have to show that all retractions have mutually disjoint images. It is easier and more general, to parametrise the machine M over the alphabet Σ and the two retractions f_1 and f_2. In the definition of M, we apply the tapes-lift on M_1 and M_2.

7.5 Designing Machines

As noted above, when we speak of "programming" Turing machines, we mean that we use tapes as registers, and use the control-flow operators and lifting operators to compose machines. Moreover, we only use machines that directly change the values of registers (tapes). We have machines that do a case-distinction on values, and machines that apply constructors to the value that a tape contains.

Our While 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, which is then translated to a Turing machine in "programming style".

We make an informal distinction between three *roles* of tapes: *input tapes*, *output tapes*, and *internal tapes*. Each kind of tape has an invariant, which is encoded in

the correctness relation of the machine. For input tapes, we have the invariant that the value on the tape is not changed by the computation. Output tapes are initially right, and contain a value after the execution. Internal tapes are right before and after the execution.

The deconstructor machines do case-distinctions over the value on a tape, and may alter the value on the tape thereby. Each deconstruction is reversible using the corresponding constructor machine. However, if we use deconstructors inside a loop, we can not restore the value that the tape contained before the loop. We have to copy the value to an internal tape, use the copy for the computation, and leave the "original" value unchanged. After the loop, the internal tape for the copy is resetted.

With this distinction of input, output, and internal tapes, we can formalise a convention for functional computation of binary functions: The tapes $t[0]$ and $t[1]$ are the input tapes for x and y, $t[2]$ is the output tape for $f\ x\ y$, and all other tapes are internal tapes.

Definition 7.51 (Functional computation correctness relation) *Let* $f : X \to Y \to Z$, *where* X *and* Y *are encodable on* Σ. *Then* $\mathsf{FunRel}(f) \subseteq \mathsf{Tape}_{\Sigma_+}^{3+n} \times \mathsf{Tape}_{\Sigma_+}^{3+n}$ *is defined as:*

$$\begin{aligned}\mathsf{FunRel}(f) := \lambda t\ t'.\ \forall (x : X)\ (y : Y).\\ & t[0] \simeq x \to t[1] \simeq y \to\\ & \mathsf{isRight}\ t[2] \to\\ & (\forall (i : \mathbb{F}_n).\ \mathsf{isRight}\ t[3+i]) \to\\ & t'[0] \simeq x\ \wedge\ t'[1] \simeq y\ \wedge\\ & t'[2] \simeq f\ x\ y\ \wedge\\ & \left(\forall (i : \mathbb{F}_n).\ \mathsf{isRight}\ t'[3+i]\right).\end{aligned}$$

We say that a machine M *computes the function* f, *if* $M \vDash \mathsf{FunRel}\ f$.

Similarly, we can define the running time function of such a machine.

Definition 7.52 (Functional computation running time relation) *Let* $\mathsf{steps} : X \to Y \to \mathbb{N}$, *where* X *and* Y *are encodable on* Σ. *Then* $\mathsf{FunT}(\mathsf{steps}) \subseteq \mathsf{Tape}_{\Sigma_+}^{3+n} \times \mathbb{N}$ *is defined as:*

$$\begin{aligned}\mathsf{FunT}(\mathsf{steps}) := \lambda t\ k.\ \exists (x : X)\ (y : Y).\\ & t[0] \simeq x\ \wedge\ t[1] \simeq y\ \wedge\\ & \mathsf{isRight}\ t[2]\ \wedge\\ & (\forall (i : \mathbb{F}_n).\ \mathsf{isRight}\ t[3+i])\ \wedge\\ & \mathsf{steps}\ x\ y \leqslant k\end{aligned}$$

Note that these definitions can be generalised to unary functions or functions with higher arity.

Internal tapes may be used to copy input-values on, if their value changes in the loop. The general design for implementing a machine M that computes a binary function f, is the following:

$$\begin{aligned} M := {} & \Uparrow_{[0;3]} \mathsf{CopyValue};\ \Uparrow_{[1;4]} \mathsf{CopyValue}; \\ & \Uparrow_{[\dots]} \mathsf{WriteValue}\ \cdots; \\ & \Uparrow_{[3;4;2;5;6;7;\cdots]} \mathsf{Loop}; \\ & \Uparrow_{[3]} \mathsf{Reset};\ \Uparrow_{[4]} \mathsf{Reset};\ \Uparrow_{[5]} \mathsf{Reset};\ \cdots \end{aligned}$$

with $\mathsf{Loop} := \mathsf{While}(\mathsf{Step})$. Step is labelled over $\mathcal{O}(1)$, where $\lfloor() \rfloor$ means to break out of the loop and $\emptyset$ to continue. Note that Step and Loop have only "access" to the copies of x and l, but not to the "original" on $t[0]$ and $t[1]$ of M. This also means that Step has two tapes less than M. The copy of x and y is on the 0st and 1st tape of Step, the output is on tape 2, all further tapes are internal tapes. Before the loop, some tapes may be initialised with values (using either WriteValue, or constructors). After the loop, all tapes that Step did not reset, including the tapes that contain the modified copies of x and y and additional internal tapes, are resetted by M.

Before we can define the machine M, we first have to decide on which alphabet M is defined. If the machine operates on only one data type, then we define M over the smallest alphabet to encode this type on. On the other hand, if M operates on more alphabets, we use the technique from Section 7.4.

For the verification of M, we first have to give the correctness relation of Step. It should encode the loop invariant as general as possible. Then we can define the correctness relation LoopRel of Loop and prove $\mathsf{Loop} \vDash \mathsf{LoopRel}$. After that, we define running time relations StepT and LoopT, and show $\mathsf{Step} \downarrow \mathsf{StepT}$, and after that $\mathsf{Loop} \downarrow \mathsf{LoopT}$. Finally, we prove $M \vDash \mathsf{FunRel}\ f$ and $M \downarrow \mathsf{FunT}\ steps$, where $f : X \to Y \to Z$ is the function that M computes and $steps : X \to Y \to \mathbb{N}$ the running time function.

7.6 Case Studies

This is the end of the framework for verified programming of Turing machines. As a benchmark for it, and to test the above general design pattern for function-computing machines, we implement a few machines that compute functions.[2]

[2]Note that we amend the design pattern a bit for convenience in Chapter 8.

7.6.1 Addition

We want to implement the functions $\mathtt{add}, \mathtt{mult} : \mathbb{N} \to \mathbb{N} \to \mathbb{N}$, according to Definition 7.51. First, we define the machine Add that computes the addition function. We reuse this machine to implement a machine Mult that computes the multiplication function.

The algorithm that our machine implements can be described with the following pseudocode:

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

The output tape is the tape that is represented by the variable $\mathtt{a}$. First, we copy the input $\mathtt{n}$ to this tape, and the number $\mathtt{m}$ to an internal tape. In the loop, as long as we can decrease the copy of $\mathtt{m}$, we increment $\mathtt{a}$. After the loop, we reset the copy $\mathtt{b}$ 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 Table 7.1.

Tape of Add	Variable	Role	Tape in AddStep
0	$\mathtt{m}$	Input	–
1	$\mathtt{n}$	Input	–
2	$\mathtt{a}$	Output	0
3	$\mathtt{b}$	Internal	1

Table 7.1: Tape assignment for Add and AddStep

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 $\mathtt{a}$ and $\mathtt{b}$, that are stored on tape 0 and 1, as also visualised in Table 7.1. The decrement operation and test whether $\mathtt{b}$ was 0 is implemented using the deconstructor machine CaseNat. In the case that $\mathtt{b}$ is 0, the step machine terminates in $\lfloor() \rfloor$, so that the loop brakes. In case $\mathtt{b}$ is greater than 0, CaseNat decreases $\mathtt{b}$ and the step machine increases $\mathtt{a}$. Then it terminates in $\emptyset$, so that the loop continues.

Definition 7.53 (AddStep**)**

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

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

Lemma 7.54 (Correctness of AddStep**)** AddStep $\vDash^{9}$ AddStepRel *with*

$$
\begin{array}{l}
\mathtt{AddStepRel} \;:= \\
\quad \lambda\, t\ (l, t').\ \forall (a\ b : \mathbb{N}). \\
\qquad t[0] \simeq a \to t[1] \simeq b \to \\
\qquad \mathsf{match}\ l, b \\
\qquad [\ \lfloor () \rfloor, O \Rightarrow t'[0] \simeq a\ \wedge\ t'[1] \simeq b \\
\qquad \mid \emptyset, S\ b' \Rightarrow t'[0] \simeq S\ a\ \wedge\ t'[1] \simeq b' \\
\qquad \mid \_, \_ \Rightarrow \bot \\
\qquad]\ .
\end{array}
$$

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 7.55 (Correctness of AddLoop**)** AddLoop $\vDash$ AddLoopRel *with*

$$\mathtt{AddLoopRel} := \lambda t\ t'.\ \forall (a\ b : \mathbb{N}).\ t[0] \simeq a \to t[1] \simeq b \to t'[0] \simeq a + b \wedge t'[1] \simeq 0.$$

Proof Using the While-induction Lemma 4.22. In 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 $t''[1] \simeq 0$. □

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 7.56 (Running time of AddLoop**)** AddLoop $\downarrow$ AddLoopT *with*

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

Now we can define the full machine Add:

Definition 7.57 (Add**)**

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

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 right. 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 right after the execution of the sub-machine. If a cell of the table is empty, the tape has not changed, then read further left in the same row. In Table 7.2, we have an example of an execution protocol for Add.

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

Table 7.2: Execution protocol of Add

From the execution protocol in Table 7.2, we can 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 right. Execution protocols serve as outlines of the formal correctness proofs. We conclude the correctness of Add.

Lemma 7.58 (Correctness of Add) $\mathsf{Add} \vDash \mathsf{FunRel}\ \mathit{add}$.

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 7.59 (Running time of Add) $\mathsf{Add} \downarrow \mathsf{FunT}\ \mathit{stepsAdd}$ *with*

$$\mathit{stepsAdd}\ m\ n := 98 + 22 \cdot m + 12 \cdot n$$

Sub-Machine	Running time	Accumulated running time
$\Uparrow_{[1;2]}$ CopyValue	$37 + 12 \cdot n$	$98 + 22 \cdot m + 12 \cdot n$
$\Uparrow_{[0;3]}$ CopyValue	$37 + 12 \cdot m$	$60 + 22 \cdot m$
$\Uparrow_{[2;3]}$ AddLoop	$9 + 10 \cdot m$	$22 + 10 \cdot m$
$\Uparrow_{[3]}$ Reset	12	12

Table 7.3: Accumulated sub-running times of Add

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 accumulate the running times from down to top and have to add one additional step for each sequence operator. This is visualised in Table 7.3.

Tape of Mult	Variable	Role	Tape in MultStep	Tape in Add
0	m	Input	–	–
1	n	Input	1	0 (input)
2	c	Output	2	1 (input)
3	c'	Internal	3	2 (output)
4	–	Internal	4	3 (internal)
5	m'	Internal	0	–

Table 7.4: Tape assignment for Mult

Input	CaseNat	Add	Reset	CopyValue	Reset
$0 : m$	$0 : m'$				
$1 : n$		$0 : n$			
$2 : c$		$1 : c$	$0 : \dashv$	$1 : n + c$	
$3 : \dashv$		$2 : n + c$		$0 : n + c$	$0 : \dashv$
$4 : \dashv$		$3 : \dashv$			

Table 7.5: Execution protocol of MultStep for $m = S\ m'$

7.6.2 Multiplication

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 can not 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 resetted afterwards.

The tape-assignment is visualised in Table 7.4. The step machine, MultStep, has only access to the copy of m, which is mapped to tape 0 of MultStep. In the following, we use also use the name m for its copy in the context of MultStep.

Definition 7.60 (MultStep**)**

$$\begin{aligned}\mathsf{MultStep} := \mathsf{If}\ & \Uparrow_{[0]} \mathsf{CaseNat}\\ & \mathsf{Then\ Return}\ \left(\Uparrow_{[1;2;3;4]} \mathsf{Add};\ \Uparrow_{[2]} \mathsf{Reset};\ \Uparrow_{[3;2]} \mathsf{CopyValue};\ \Uparrow_{[3]} \mathsf{Reset}\right)\ \emptyset\\ & \mathsf{Else}\quad \mathsf{Return\ Nop}\ \lfloor() \rfloor\,.\end{aligned}$$

An execution protocol of MultStep for the case $m = S\ m'$ (where m is the copy) is shown in Table 7.5. The correctness relation of MultStep is analogous to AddStep.

Lemma 7.61 (Correctness of MultStep**)** MultStep $\vDash^9$ MultStepRel *with*

$$\begin{aligned}&\mathtt{MultStepRel}\ :=\\ &\quad \lambda\, t\ (l, t').\ \forall (c\ m\ n : \mathbb{N}).\\ &\qquad t[0] \simeq m \to t[1] \simeq n \to t[2] \simeq c \to \mathtt{isRight}\ t[3] \to \mathtt{isRight}\ t[4] \to\\ &\qquad \mathsf{match}\ l, m\\ &\qquad [\, \lfloor() \rfloor, O \Rightarrow t'[0] \simeq 0 \ \wedge\ t'[1] \simeq n \ \wedge\ t'[2] \simeq c \ \wedge\ \mathtt{isRight}\ t'[3] \ \wedge\ \mathtt{isRight}\ t'[4]\\ &\qquad |\, \emptyset, S\ m' \Rightarrow t'[0] \simeq m' \ \wedge\ t'[1] \simeq n \ \wedge\ t'[2] \simeq n + c \ \wedge\ \mathtt{isRight}\ t'[3] \ \wedge\ \mathtt{isRight}\ t'[4]\\ &\qquad |\, \_, \_ \ \Rightarrow\ \bot\\ &\qquad]\,.\end{aligned}$$

Note that in the correctness Lemma 7.54 of AddStep, we also prove termination in constant running time. This is not true for MultStep, because it calls Add, which has not 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 right, then after the execution of the loop, the tapes contain 0, n, and $m \cdot n + c$ and the other tapes are right. Thus, when we instantiate the c-tape with the value 0, the output tape contains $m \cdot n$.

Lemma 7.62 (Correctness of MultLoop**)** MultLoop $\vDash$ MultLoopRel *with*

$$\begin{aligned}\mathtt{MultLoopRel} :=& \lambda t\ t'.\ \forall (c\ m\ n : \mathbb{N}).\\ & t[0] \simeq m \to t[1] \simeq\!\!\to t[2] \simeq c \to \mathtt{isRigth}\ t[3] \to \mathtt{isRight}\ t[4] \to\\ & t'[0] \simeq 0 \wedge t'[1] \simeq n \wedge t'[2] \simeq m \cdot n + c \wedge \mathtt{isRight}\ t'[3] \wedge \mathtt{isRight}\ t'[4]\end{aligned}$$

It is now easy to define the rest of Mult.

Definition 7.63 (Mult**)**

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

It follows that Mult computes the function $\mathtt{mult}$, w.r.t. Definition 7.51.

Lemma 7.64 (Correctness of Mult**)** Mult $\vDash$ FunRel mult.

For brevity, we omit the concrete running time function here.

Lemma 7.65 (Running time of Mult**)** *There is a function* $\mathtt{multSteps} : \mathbb{N} \to \mathbb{N} \to \mathbb{N}$, *such that* $\mathtt{multSteps} \in \mathcal{O}(m \cdot n^2)$ *and* Mult $\downarrow$ FunT multSteps.

7.6.3 Mapping of Sum Functions

Let $f : X \to Z$ and $g : Y \to Z$. Then we can define the canonical function $(f + g) : X + Y \to Z$ by $(f + g)(\mathsf{inl}\, x) := f(x)$ and $(f + g)(\mathsf{inr}\, l) := g(l)$. We want to define an operator that takes machines M_1, M_2 that compute the unary functions f and g, and yields a machine MapSum that computes the function $(f + g)$.

First we have to define precisely, which machine has which alphabet. Let X, Y be encodable over Σ_X, Σ_Y. We assume $M_1, M_2 : \mathsf{TM}^{2+n}_{\Sigma^+_M}$, i.e. both machines have the same alphabet, and they have at least two tapes (i.e. an input and output tape). We assume that Σ_M includes Σ_X, Σ_Y, and Σ_Z. Formally, this means that we assume retractions f_X, f_Y, f_Z between $\Sigma_X, \Sigma_Y, \Sigma_Z$ and Σ_M. Let Σ be the alphabet for MapSum, with a retraction $f_M : \Sigma_M \hookrightarrow \Sigma$. Furthermore, it should be possible to encode $X + Y$ on Σ, so we assume a retraction $f_{X+Y} : \Sigma_{X+Y} \hookrightarrow \Sigma$. We notice that there are now two possibilities how to encode X (and dually Y) on Σ: via the retractions $f_M \circ f_X$ and $f_{X+Y} \circ \mathtt{RetrInl}$, see Figure 7.1. These retractions might be extensionally equal for concrete choices of alphabets, but we want to be as general as possible. As a consequence, we need to translate between these representations, using Translate of Section 7.3.4. Note that there is only one way to encode Z on Σ, namely via $f_M \circ f_Z$.

The machine MapSum first makes a case-distinction on the value $s : X + Y$ on the input tape 0. If it is x (on the alphabet Σ_{X+Y}), it is translated to the alphabet of M_1. Then we execute the alphabet-lifted machine M_1. This writes the output $f_1(x)$ on the output tape 1 and leaves x unchanged. After that, MapSum translates x back to the alphabet Σ_{X+Y} and applies the constructor inl, so the input tape contains $\mathsf{inl}\, x$ again. The case when the input was $s = \mathsf{inl}\, y$ is symmetric.

Definition 7.66 (MapSum**)**

$$
\begin{array}{l}
\mathsf{MapSum} := \\
\quad \mathsf{If}\ \Uparrow_{f_{X+Y};\,[0]} \mathsf{CaseSum} \\
\quad \mathsf{Then}\ \Uparrow_{[0]} (\mathsf{Translate}\ (f_{X+Y} \circ \mathtt{RetrInl})\ (f_M \circ f_X)); \\
\qquad\quad \Uparrow_{f_M} M_1; \\
\qquad\quad \Uparrow_{[0]} (\mathsf{Translate}\ (f_M \circ f_X)\ (f_{X+Y} \circ \mathtt{RetrInl})); \\
\qquad\quad \Uparrow_{f_{X+Y};\,[0]} \mathsf{ConstrInl} \\
\quad \mathsf{Else}\ \Uparrow_{[0]} (\mathsf{Translate}\ (f_{X+Y} \circ \mathtt{RetrInr})\ (f_M \circ f_Y)); \\
\qquad\quad \Uparrow_{f_M} M_2; \\
\qquad\quad \Uparrow_{[0]} (\mathsf{Translate}\ (f_M \circ f_Y)\ (f_{X+Y} \circ \mathtt{RetrInr})); \\
\qquad\quad \Uparrow_{f_{X+Y};\,[0]} \mathsf{ConstrInr}
\end{array}
$$

Lemma 7.67 (Correctness of MapSum**)** *If* $M_1 \vDash \mathtt{FunRel}(f_1)$ *and* $M_2 \vDash \mathtt{FunRel}(f_2)$, *then* $\mathsf{MapSum} \vDash \mathtt{FunRel}(f_1 + f_2)$.

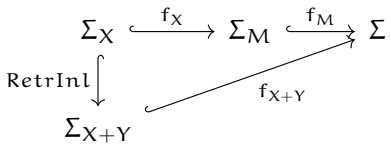


Figure 7.1: Non-commutative diagram of the retractions involved in the definition of MapSum

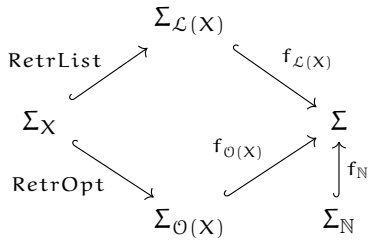


Figure 7.2: Retractions involved in the definition of Nth

7.6.4 List Access

In this section, we show how to implement machines that compute the function $\mathrm{nth} : \mathcal{L}(X) \to \mathbb{N} \to \mathcal{O}(X)$.

Let X be encodable on Σ_X. We implement a machine $\mathsf{Nth} : \mathsf{TM}^4_{\Sigma^+}$ that computes the function $\mathrm{nth} : \mathcal{L}(X) \to \mathbb{N} \to \mathcal{O}(X)$. We assume retractions $f_{\mathcal{L}(X)} : \Sigma_{\mathcal{L}(X)} \hookrightarrow \Sigma$, $f_{\mathcal{O}(X)} : \Sigma_{\mathcal{O}(X)} \hookrightarrow \Sigma$, and $f_{\mathbb{N}} : \Sigma_{\mathbb{N}} \hookrightarrow \Sigma$. Now there are two ways to encode X on Σ: via $f_{\mathcal{O}(X)} \circ \mathsf{RetrOpt}$ and $f_{\mathcal{L}(X)} \circ \mathsf{RetrList}$, see Figure 7.2.

The tape assignment is visualised in Table 7.6.

Tape of Nth	Variable	Role	Tape in NthStep
0	xs	Input	–
1	n	Input	–
2	x	Output	2
2	xs (copy)	Internal	0
3	n (copy)	Internal	1

Table 7.6: Tape assignment for Nth and NthStep

The step machine first destructs n. If it was 0 and the list is empty, it writes $\emptyset$ to the output tape. Else, it applies the $\lfloor\cdot\rfloor$ constructor to the head. Note that before we can apply the constructor, x has to be translated to the $\Sigma_{\mathcal{O}(X)}$ alphabet. In case n was non-zero, if the list is empty, the machine applies the $\emptyset$ constructor to the output tape. Else, it deletes the head (which was temporary stored on the output tape) and

continues the loop.

Definition 7.68 (NthStep**)**

$$
\begin{array}{l}
\mathsf{MapSum} := \\
\quad \mathsf{If}\ \Uparrow_{\mathsf{f}_{\mathbb{N}};\,[1]} \mathsf{CaseNat} \\
\quad \mathsf{Then\ If}\ \Uparrow_{\mathsf{f}_{\mathcal{L}(X)};\,[0;2]} \mathsf{CaseList} \\
\qquad \mathsf{Then\ Return}\ (\Uparrow_{[2]} \mathsf{Reset})\ \emptyset \\
\qquad \mathsf{Else}\ \ \mathsf{Return}\ (\Uparrow_{\mathsf{f}_{\mathcal{O}(X)};\,[2]} \mathsf{ConstrNone})\ \lfloor () \rfloor \\
\quad \mathsf{Else}\ \ \mathsf{If}\ \Uparrow_{\mathsf{f}_{\mathcal{L}(X)};\,[0;2]} \mathsf{CaseList} \\
\qquad \mathsf{Then\ Return}\ (\Uparrow_{[2]} (\mathsf{Translate}\ (\mathsf{f}_{\mathcal{L}(X)} \circ \mathtt{RetrList})\ (\mathsf{f}_{\mathcal{O}(X)} \circ \mathtt{RetrOpt}); \\
\qquad\qquad \Uparrow_{\mathsf{f}_{\mathcal{O}(X)}} \mathsf{ContrSome}))\ \lfloor () \rfloor \\
\qquad \mathsf{Else}\ \ \mathsf{Return}\ (\Uparrow_{\mathsf{f}_{\mathcal{O}(X)};\,[2]} \mathsf{ConstrNone})\ \lfloor () \rfloor
\end{array}
$$

The correctness relation of NthStep has to capture all four cases.

Lemma 7.69 (Correctness of NthStep**)** NthStep $\vDash$ NthStepRel *with*

$$
\begin{array}{l}
\mathtt{NthStepRel} := \\
\quad \lambda t\ (l, t'). \\
\qquad \forall (xs : \mathcal{L}(X))\ (n : \mathbb{N}). \\
\qquad\quad t[0] \simeq xs \to t[1] \simeq n \to \mathtt{isRight}\ t[2] \ \ \to \\
\qquad\quad \mathsf{match}\ l, n, xs \\
\qquad\quad [\, \emptyset, S\ n', x :: xs' \Rightarrow t'[0] \simeq xs' \wedge t'[1] \simeq n' \ \wedge\ \mathtt{isRight}\ t'[2] \\
\qquad\quad |\, \lfloor () \rfloor, S\ n', \mathsf{nil} \Rightarrow t'[0] \simeq l \ \wedge t'[1] \simeq n' \wedge t'[2] \simeq \emptyset \\
\qquad\quad |\, \lfloor () \rfloor, 0, x :: xs' \Rightarrow t'[0] \simeq xs' \wedge t'[1] \simeq 0 \wedge t'[2] \simeq \lfloor x \rfloor \\
\qquad\quad |\, \lfloor () \rfloor, 0, \mathsf{nil} \Rightarrow t'[0] \simeq l \wedge t'[1] \simeq n \wedge t'[2] \simeq \emptyset \\
\qquad\quad |\, \_, \_, \_ \Rightarrow \bot \\
\qquad\quad]\ .
\end{array}
$$

Now, we can prove the correctness of the loop, NthLoop := While NthStep. When the loop terminates, the value of the variable n is $n - (S\ |xs|)$ and the new value of xs is $\mathtt{skipn}\ (S\ n)\ xs$, where $\mathtt{skipn} : \mathbb{N} \to \mathcal{L}(X) \to \mathcal{L}(X)$ applies tl n-times. Note that if $n < |xs|$, the new value of n will be 0 and the machine returns a $\lfloor \cdot \rfloor$ value.

Lemma 7.70 (Correctness of NthLoop**)** NthLoop $\vDash$ NthLoopRel *with*

$$
\begin{array}{l}
\mathtt{NthLoopRel} := \lambda t\ t'.\ \forall (xs : \mathcal{L}(X))\ (n : \mathbb{N}).\ t[0] \simeq xs \to t[1] \simeq n \to \mathtt{isRight}\ t[2] \to \\
\qquad\qquad t'[0] \simeq \mathtt{skipn}\ (S\ n)\ l \wedge t'[1] \simeq n - (1 + |xs|) \wedge t'[2] \simeq \mathtt{nth}\ xs\ n.
\end{array}
$$

The full machine Nth makes copies of n and xs and runs NthLoop on the copies. After that, it resets the internal tapes 2 and 3 with the copies of n and xs. Note that

for the correctness of Reset it is only important that there is a value on the tape. Only the running time (which we omit here) does depend on the actual value.

Definition 7.71 (Nth)

$$\mathsf{Nth} := \Uparrow_{[0;3]} \mathsf{CopyValue};\ \Uparrow_{[1;4]} \mathsf{CopyValue};\ \Uparrow_{[3;4;2]} \mathsf{NthLoop};\ \Uparrow_{[3]} \mathsf{Reset};\ \Uparrow_{[4]} \mathsf{Reset}$$

Note that we copy and reset values of different types. We have to insert this semantic information in correctness proofs when we apply the respective correctness lemmas of Copy and Reset in the relation-derivation phase of correctness proofs.

It follows that Nth computes the function $\mathtt{nth}$.

Lemma 7.72 (Correctness of Nth) $\mathsf{Nth} \vDash \mathtt{FunRel}\ \mathtt{nth}$.

Chapter 8

Simulating a Call-By-Value λ-Calculus Machine

In this chapter, we present a large benchmark for our framework. We implement a multi-tape Turing machine that simulates another abstract machine. First, we define and motivate the semantics of this machine. After that, we implement and verify the simulator Turing machine. We show that the halting problem of the abstract machine reduces to the halting problem of multi-tape Turing machines.

8.1 Heap Machine

The abstract machine we present here is a variant of the heap machine in Kunze et al. [13]. They formally show in Coq that their variant of the machine can simulate the language L, which implements the call-by-value subset of the λ-calculus. Thus, we implement a simulator that also simulates L. Here we define the language L only briefly. For more a more thorough treatment of this language, we refer to [10] and [13]. The concrete definition of the machine we are considering is due to F. Kunze.

Call-By-Value λ-Calculus

Terms of this language are De Bruijn terms, and are inductively defined by:

$$s, t, u, v \ : \ \mathsf{Ter} \ ::= \ (n : \mathbb{N}) \mid s\ t \mid \lambda s$$

The language L uses simple substitution:

$$\begin{aligned} k_u^k &:= u \\ n_u^k &:= n && \text{if } n \neq k \\ (st)_u^k &:= (s_u^k)(t_u^k) \\ (\lambda s)_u^k &:= \lambda(s_u^{Sk}) \end{aligned}$$

The reduction relation $s \succ t$ is inductively defined on terms:

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

Heap Machine

Instead of substituting expressions, the heap machine works with closures. A closure is a program together with an environment. Variable bindings are implemented as pointers to a linked list of closures, called *heap*. Programs are lists of commands:

Definition 8.1 (Program)

$$\begin{aligned} n : Var &:= \mathbb{N} \\ c : Com &::= \mathsf{VAR}(n : Var) \mid \mathsf{APP} \mid \mathsf{LAM} \mid \mathsf{RET} \\ P, Q : Pro &:= \mathcal{L}(Com) \end{aligned}$$

Programs essentially are linearised expressions of L. The function γ translates terms of L to programs of the heap machine:

$$\begin{aligned} \gamma \quad n &:= [\mathsf{VAR}\ n] \\ \gamma\ (s\ t) &:= \gamma\ s +\!\!+ \gamma\ t +\!\!+ [\mathsf{APP}] \\ \gamma\ (\lambda s) &:= \mathsf{LAM} :: \gamma\ s +\!\!+ [\mathsf{RET}] \end{aligned}$$

The heap is implemented as a list of heap entries. A heap entry may be empty or contain a closure and a pointer to the next heap entry. Pointers are implemented as list indices on the heap entry list.

Definition 8.2 (Closures and heaps)

$$\begin{aligned} a, b : Add &:= \mathbb{N} \\ g : Clos &:= Add \times Pro \\ e : Entr &:= \mathcal{O}(Clos \times Add) \\ H : Heap &:= \mathcal{L}(Entr) \end{aligned}$$

States of the abstract machine are triples of two closure lists and a heap:

$$(T, V, H) \ : \ \mathcal{L}(Clos) \times \mathcal{L}(Clos) \times Heap.$$

T is called the control stack. It contains the closures that the machine has to process. The second stack V is called the argument stack. For example, the APP command fetches the functions and arguments from that stack.

We first explain what each rule of the reduction predicate $(T, V, H) \succ (T', V', H')$ does, and then we define it formally.

Application Rule When the first closure on the control stack is $(a, \mathsf{APP} :: P)$, the machine fetches two closures g and (b, Q) from the argument stack. The closure g corresponds to the argument of the application. The second closure (b, Q) corresponds to the called function, where b is the pointer to the environment in that the argument is free. It binds the argument to g, by putting a new heap entry (g, b) on the heap. The heap machine continues executing (c, Q) and the "rest closure" (a, P), where c is the pointer to the new heap entry.

λ-Rule If the head control closure is $(a, \mathsf{LAM} :: P)$, the machine splits the linearised program P into the body Q of the λ-expression and the rest program P'. Then it pushes the closure of the rest program (a, P') on the control stack, and the "body closure" (a, Q) on the argument stack. The splitting is realised with the function $\phi : \mathtt{Pro} \to \mathcal{O}(\mathtt{Pro} \times \mathtt{Pro})$. For example,

$$\phi[\mathsf{VAR}(0); \mathsf{LAM}; \mathsf{APP}; \mathsf{RET}; \mathsf{RET}; \mathsf{VAR}(1)] = \lfloor([\mathsf{VAR}(0); \mathsf{LAM}; \mathsf{APP}; \mathsf{RET}], [\mathsf{VAR}(1)])\rfloor\,.$$

The first RET matches to the LAM in the program above, so it is part of the first half. ϕ is formally defined with a tail-recursive auxiliary function:

Definition 8.3 (ϕ) *We define* $\phi\ P := \phi'\ 0\ \mathsf{nil}\ P$ *with the auxiliary function* $\phi' : \mathbb{N} \to \mathtt{Pro} \to \mathtt{Pro} \to \mathcal{O}(\mathtt{Pro} \times \mathtt{Pro})$ *which is defined by recursion on* $P : \mathtt{Pro}$*:*

$$\begin{aligned}
\phi'\ \ \ 0\ Q\ (\mathsf{RET} :: P) &:= \lfloor (Q, P') \rfloor \\
\phi'\ (S\ k)\ Q\ (\mathsf{RET} :: P) &:= \phi'\ k\ (Q +\!\!+ [\mathsf{RET}])\ P \\
\phi'\ \ \ k\ Q\ (\mathsf{LAM} :: P) &:= \phi'\ (S\ k)\ (Q +\!\!+ [\mathsf{LAM}])\ P \\
\phi'\ \ \ k\ Q\ \ \ (c :: P) &:= \phi'\ k\ (Q +\!\!+ [c])\ P \qquad \textit{if } c \neq \mathsf{RET} \neq \mathsf{LAM} \\
\phi'\ \ \ k\ Q\ \ \ \ \ \ \mathsf{nil} &:= \emptyset
\end{aligned}$$

Variable Rule If the first command on the first closure of the control stack is $\mathsf{VAR}(n)$, the machine looks up the nth entry in the environment on the heap at the address a. Then, it pushes this closure to the argument stack. The function $\mathtt{lookup} : \mathtt{Heap} \to \mathtt{Add} \to \mathtt{Var} \to \mathcal{O}(\mathtt{Clos})$ starts at the heap entry at the address a, and gets the nth entry. We write $H[a, n]$ for $\mathtt{lookup}\ H\ a\ n$. The function is defined by recursion on n:

Definition 8.4 (lookup) $H[a, n]$ *is defined by recursion on* n*:*

$$H[a, n] := \begin{cases} \lfloor g \rfloor & H[a] = \lfloor\lfloor (g, b) \rfloor\rfloor \wedge n = 0 \\ H[b, n-1] & H[a] = \lfloor\lfloor (g, b) \rfloor\rfloor \wedge n > 0 \\ \emptyset & \textit{else} \end{cases}$$

where $H[\cdot] : \mathcal{O}(\mathtt{Entr})$ *is the standard list lookup function.*

We formally define the reduction rules. Note that there is no reduction rule for the command RET, because the purpose of RET is solely to mark the end of the encoding of the body of a λ-expression.

Definition 8.5 (Semantics of the heap machine)

$$\begin{array}{lll}
((a, (\mathsf{APP} :: P)) :: T, g :: (b, Q) :: V, H) & \succ ((c, Q) :: (a, P) ::_{tr} T, V, H') & \textit{if } \mathtt{put}\ H\ \lfloor (g, b) \rfloor = (c, H') \\
((a, (\mathsf{LAM} :: P)) :: T, V, H) & \succ ((a, P') ::_{tr} T, (a, Q) :: V, H) & \textit{if } \phi\ P = \lfloor (Q, P') \rfloor \\
((a, (\mathsf{VAR}(n) :: P)) :: T, V, H) & \succ ((a, P) ::_{tr} T, g :: V, H) & \textit{if } H[a, n] = \lfloor g \rfloor
\end{array}$$

with $\mathtt{put}\ H\ c := (|H|, H +\!\!+ [c])$.
The tail-recursion optimisation $(g ::_{tr} T) : \mathcal{L}(\mathtt{Clos})$ *is defined as follows:*

$$\begin{array}{lll}
(a, \mathsf{nil}) ::_{tr} T := T & & \\
(a, P) ::_{tr} T := (a, P) :: T & \textit{if } P \neq \mathsf{nil}
\end{array}$$

We write for $\succ^*$ *for the reflexive transitive closure of* $\succ$ *(cf. Definition 2.2) and* $\succ^k$ *for the relational power (cf. Definition 2.3). Furthermore, we use the following notations:*

$$\begin{array}{rcl}
\mathtt{halt}(T, V, H) & := & \forall T'\ V'\ H'.\ \neg\left((T, V, H) \succ (T', V', H')\right) \\
(T, V, H) \triangleright^k (T', V', H') & := & (T, V, H) \succ^k (T', V', H') \wedge \mathtt{halt}(T', V', H') \\
(T, V, H) \triangleright^* (T', V', H') & := & (T, V, H) \succ^* (T', V', H') \wedge \mathtt{halt}(T', V', H')
\end{array}$$

The tail-recursion optimisation ($::_{tr}$) makes sure that no closures with empty programs are pushed on a closure stack. This optimises space usage for tail recursive programs.

Lemma 8.6 (Basic properties about $\succ$)

1. *The relation* $\succ$ *is functional and computable, i.e. there is a function* $\mathtt{step} : \mathcal{L}(\mathtt{Clos}) \times \mathcal{L}(\mathtt{Clos}) \times \mathtt{Heap} \to \mathcal{O}\left(\mathcal{L}(\mathtt{Clos}) \times \mathcal{L}(\mathtt{Clos}) \times \mathtt{Heap}\right)$*, such that:*

$$\mathtt{step}(T, V, H) = \left\lfloor (T', V', H') \right\rfloor \leftrightarrow (T, V, H) \succ (T', V', H')$$

2. $\mathtt{halt}(T, V, H)$ *is decidable, i.e. there is a function* $\mathtt{isHalt} : \mathcal{L}(\mathtt{Clos}) \times \mathcal{L}(\mathtt{Clos}) \times \mathtt{Heap} \to \mathbb{B}$*, such that:*

$$\mathtt{isHalt}(T, V, H) = \mathsf{true} \leftrightarrow \mathtt{halt}(T, V, H)$$

8.2 Implementation of a Simulator

First, we notice that all types in Definition 8.2, except commands ($\mathtt{Com}$), are encodable on minimal alphabets, according to Definition 7.4:

Definition 8.7 (Encoding of heaps) *The type* $\mathtt{Com}$ *is isomorphic to the sum of* $\mathtt{Var}$ *and* $\mathtt{ACom} := \mathsf{APP} \mid \mathsf{LAM} \mid \mathsf{RET}$. $\mathtt{ACom}$ *is encodable on itself (since it is finite), and* $\mathtt{Var} = \mathbb{N}$ *is encodable on* $\Sigma_{\mathtt{Var}} := \Sigma_{\mathbb{N}}$. *Then* $\mathtt{Com}$ *is encodable on* $\Sigma_{\mathtt{Com}} := \Sigma_{\mathbb{N}+\mathtt{ACom}}$. *All other types are encodable according to Definition 7.4. For completeness, the alphabets are:* $\Sigma_{\mathtt{Pro}} := \Sigma_{\mathcal{L}(\mathtt{Com})}$, $\Sigma_{\mathtt{Add}} := \Sigma_{\mathbb{N}}$, $\Sigma_{\mathtt{Clos}} := \Sigma_{\mathtt{Add}\times\mathtt{Pro}}$, $\Sigma_{\mathtt{Entr}} := \Sigma_{\mathcal{O}(\mathtt{Clos}\times\mathtt{Add})}$, *and* $\Sigma_{\mathtt{Heap}} := \Sigma_{\mathcal{L}(\mathtt{Entr})}$.

First, we derive constructor and deconstructor machines for commands. As the type of commands essentially is the sum of two encodable types ($\mathtt{ACom}$ and $\mathbb{N}$), we can combine constructors and deconstructors from Section 7.3.2.

Definition 8.8 (CaseCom**)** $\mathsf{CaseCom} : \mathsf{TM}^1_{\Sigma^+_{\mathtt{Com}}}(\mathcal{O}(\mathtt{ACom}))$ *is defined by*

$$\begin{aligned}\mathsf{CaseCom} := \mathsf{If}\ &\mathsf{CaseSum}\\ \mathsf{Then}\ &\mathsf{Return\ Nop}\ \emptyset\\ \mathsf{Else}\ \ &\mathsf{Relabel}\ (\Uparrow_{\mathtt{RetrInl}}\mathsf{CaseFin})\ (\lfloor\cdot\rfloor)\end{aligned}$$

Note that $\mathtt{RetrInl} : \Sigma_{\mathtt{Var}} \hookrightarrow \Sigma_{\mathtt{Com}}$.

Lemma 8.9 (Correctness of CaseCom**)** $\mathsf{CaseCom} \vDash^{11} \mathtt{CaseComRel}$ *with*

$$\begin{aligned}&\mathtt{CaseComRel} :=\\ &\quad \lambda t\ (l, t').\ \forall(c : \mathtt{Com}).\ t[0] \simeq c \to \mathsf{match}\ l, c\\ &\qquad [\ \lfloor\mathsf{APP}\rfloor, \mathsf{APP} \Rightarrow \mathtt{isRight}\ t'[0]\\ &\qquad \mid \lfloor\mathsf{LAM}\rfloor, \mathsf{LAM} \Rightarrow \mathtt{isRight}\ t'[0]\\ &\qquad \mid \lfloor\mathsf{RET}\rfloor, \mathsf{RET} \Rightarrow \mathtt{isRight}\ t'[0]\\ &\qquad \mid \emptyset, \mathsf{VAR}\ n \Rightarrow t'[0] \simeq n\\ &\qquad \mid \_, \_ \Rightarrow \bot\\ &\qquad]\ .\end{aligned}$$

The VAR constructor is defined with ConstrInl and the constructor for $\mathtt{ACom}$ is defined with WriteSymbol.

For the functions ϕ and $\mathtt{lookup}$, we define machines that *compute* these functions in a lesser strict sense than Definition 7.51. The Turing machines only save the input-values that a caller machine needs again. For example, we have a machine Nth′ that saves the list, but not the index. The second change is that for functions with optional outputs, like $\mathtt{nth}$, we do not extend the alphabet with $\Sigma_{\mathcal{O}(X)}$. Instead, we encode whether the output value is $\lfloor\cdot\rfloor$ or $\emptyset$ with the label type $\mathbb{B}$. In case that the output is $\emptyset$, the machine terminates in false and the output tape stays right.

After we have defined and verified machines that compute ϕ and $\mathtt{lookup}$, we define a machine Step that simulates a single step of the heap machine.

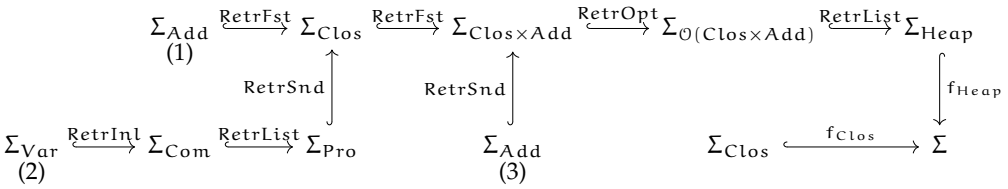


Figure 8.1: The retractions of Lookup.

8.2.1 Lookup

The machine $\mathsf{Lookup} : \mathsf{TM}^5_{\Sigma^+}(\mathbb{B})$ realises the heap lookup function. It uses the machine $\mathsf{Nth}' : \mathsf{TM}^4_{\Sigma^+}(\mathbb{B})$ which is like Nth from Section 7.6.4, but with the two changes mentioned above.

First we consider the alphabet of Lookup. We can encode closures on the heap alphabet Σ_{Heap}. However, when Step calls $H[a, n]$, a and n are encoded on an alphabet for the closures on the closure stack. Thus, we define Lookup on an alphabet Σ, with retractions $f_{Heap} : \Sigma_{Heap} \hookrightarrow \Sigma$ and $f_{Clos} : \Sigma_{Clos} \hookrightarrow \Sigma$. The second retraction correspond to the closure alphabet where a and n are encoded. The retraction

$$f_{Clos'} : \Sigma_{Clos} \hookrightarrow \Sigma := f_{Heap} \circ \mathsf{RetrList} \circ \mathsf{RetrOpt} \circ \mathsf{RetrFst}$$

is the canonical retraction of closures from the heap. In Figure 8.1, the three relevant retractions $\Sigma_{\mathbb{N}} \hookrightarrow \Sigma$ are visualised:

1. a heap address of a closure on the stack alphabet:

$$f_{add} := f_{Clos'} \circ \mathsf{RetrFst}$$

2. a variable in a command of a closure on the stack alphabet:

$$f_{var} := f_{Clos'} \circ \mathsf{RetrSnd} \circ \mathsf{RetrList} \circ \mathsf{RetrInl}$$

3. a pointer to the next heap entry:

$$f_{next} := f_{Heap} \circ \mathsf{RetrList} \circ \mathsf{RetrOpt} \circ \mathsf{RetrSnd}$$

Since the function lookup is tail-recursive, we define the step machine with the type $\mathsf{LookupStep} : \mathsf{TM}^5_{\Sigma^+}(\mathcal{O}(\mathbb{B}))$. It first calls the list lookup machine (Nth′). If it failed, Lookup immediately terminates in false. Else, it makes a case-distinction on the heap entry $e : \mathcal{O}(Clos \times Add)$. In case it is $\emptyset$, Lookup also terminates in false. If $e = \lfloor e' \rfloor$, it destructs $e' = (g, b)$ and makes a case-distinction over n. If $n = 0$, LookupStep resets the tapes for b and n, translates g from $f_{Clos'}$ to f_{Clos} and returns true. Else

it translates b from f_{next} to f_{add}, moves it to the tape that contained a, and repeats the loop. We visualise the execution in the latter two cases in Tables 8.1 and 8.2.

Input	Nth′	CaseOption	CasePair	CaseNat	CopyValue	Translate	Reset	Reset
$0 : H$	$0 : H$							
$1 : a$	$1 : \dashv$				$1 : b$	$0 : b$		
$2 : n$				$0 : n'$				
$3 : \dashv$			$1 : g$					$0 : \dashv$
$4 : \dashv$	$2 : \lfloor(g, b)\rfloor$	$0 : (g, b)$	$0 : b$		$0 : b$		$0 : \dashv$	

Table 8.1: Execution protocol of LookupStep in case $H[a] = \lfloor\lfloor(g, b)\rfloor\rfloor$ and $n = S\ n'$. It terminates in the label $\emptyset$. The translation is from f_{next} to f_{add}.

Input	Nth′	CaseOption	CasePair	CaseNat	Reset	Reset	Translate
$0 : H$	$0 : H$						
$1 : a$	$1 : \dashv$						
$2 : n$				$0 : 0$		$0 : \dashv$	
$3 : \dashv$			$1 : g$				$0 : g$
$4 : \dashv$	$2 : \lfloor(g, b)\rfloor$	$0 : (g, b)$	$0 : b$		$0 : \dashv$		

Table 8.2: Execution protocol of LookupStep in case $H[a] = \lfloor\lfloor(g, b)\rfloor\rfloor$ and $n = 0$. It terminates in the label $\lfloor\mathsf{true}\rfloor$. The translation is from $f_{Clos'}$ to f_{Clos}.

Definition 8.10 (LookupStep**)** *We define the machine* LookupStep : $\mathsf{TM}^5_{\Sigma^+}(\mathcal{O}(\mathbb{B}))$.

```
LookupStep :=
  If (Nth′ f_heap f_add)
  Then If (⇑_{f_heap∘RetrList; [4]} CaseOption)
       Then ⇑_{f_heap∘RetrList∘RetrOpt; [4;3]} CasePair;
            If (⇑_{f_var; [2]} CaseNat)
            Then Return (⇑_[4;1] CopyValue; ⇑_[1] Translate f_next f_add;
                         ⇑_[4] Reset; ⇑_[3] Reset) ∅
            Else Return (⇑_[4] Reset; ⇑_[2] Reset;
                         ⇑_[3] Translate f_Clos′ f_Clos) ⌊true⌋
       Else Return Nop ⌊false⌋
  Else Return Nop ⌊false⌋
```

The step machine terminates in $\lfloor\mathsf{false}\rfloor$ in two cases. Either the list lookup failed, or it returned the empty heap entry $H[a] = \lfloor\emptyset\rfloor$. In the successfully termination case (cf. Table 8.2), LookupStep resets the afterwards unneeded tapes.

Because no initialisation or clean-up is needed before or after the loop, we can define Lookup := While LookupStep. The correctness relation of Lookup says that if it terminated in the label true, we have $H[a, n] = \lfloor g\rfloor$ for some closure g, and tape 3 contains g w.r.t. the retraction f_{Clos}. Furthermore, the input tape 0 still contains

H and all other tapes are resetted. In case Lookup terminated in false, the postcondition only commits that $H[a, n] = \emptyset$. Note that there are two possible reasons for that: Either $H[a] = \emptyset$ or $H[a] = \lfloor\emptyset\rfloor$, but we do not separate these two failure cases.

Lemma 8.11 (Correctness of Lookup**)** Lookup $\vDash$ LookupRel *with*

$$\begin{aligned}
&\mathsf{LookupRel} :=\\
&\quad \lambda t\ (l, t').\ \forall H\ a\ n.\ t[0] \simeq H \to t[1] \simeq_{f_{add}} a \to t[2] \simeq_{f_{var}} n \to \mathsf{isRight}\ t[3] \to \mathsf{isRight}\ t[4] \to\\
&\qquad \mathsf{if}\ l\ \mathsf{then}\ \big(\exists g.\ H[a, n] = \lfloor g\rfloor \wedge\\
&\qquad\qquad t'[0] \simeq H \wedge \mathsf{isRight}\ t'[1] \wedge \mathsf{isRight}\ t'[2] \wedge t'[3] \simeq_{f_{Clos}} g \wedge \mathsf{isRight}\ t[4]\big)\\
&\qquad\ \mathsf{else}\ \ H[a, n] = \emptyset.
\end{aligned}$$

8.2.2 SplitBody

The machine SplitBody : $\mathsf{TM}^5_{\Sigma^+_{Pro}}(\mathbb{B})$ computes the function ϕ (cf. Definition 8.3). The implementation of this machine is straight-forward, because ϕ' is defined tail-recursively. The machine SplitBodyLoop := While SplitBodyStep computes ϕ', and SplitBody initialises the accumulators before executing the loop.

The step machine SplitBodyStep : $\mathsf{TM}^5_{\Sigma^+_{Pro}}(\mathcal{O}(\mathbb{B}))$ first makes a case-distinction over the program on tape 0. In the nil case, it returns $\lfloor$false$\rfloor$ so that the loop returns false. In the cons-case, it makes a case-distinction over the command, using CaseCom. In the case the head command is RET, it does another case-distinction over k. In the case that the head command is VAR(n), the machine applies the VAR constructor to n again and appends this command to Q. The case-machines use the parametrised auxiliary machine AppACom t that appends a finite command (i.e. t is either APP, LAM, or RET) to the command list, and the machine AppCom that appends a command on a tape to Q.

Definition 8.12 (SplitBodyStep**)** *We define the canonical retraction* $f_{Var} : Var \hookrightarrow \Sigma_{Pro}$ *by* $f_{Var} := \mathsf{RetrList} \circ \mathsf{RetrInl}$.

$$\begin{aligned}
&\mathsf{SplitBodyStep} :=\\
&\quad \mathsf{If}\ (\Uparrow_{[0;3]} \mathsf{CaseList})\\
&\quad \mathsf{Then\ Switch}\ (\Uparrow_{\mathtt{RetrList};\,[3]} \mathsf{CaseCom})\\
&\qquad (\lambda(c : \mathcal{O}(\mathcal{A}\mathrm{Com}))\mathsf{.\ match}\ c\\
&\qquad\quad [\ \lfloor\mathsf{RET}\rfloor \Rightarrow \mathsf{If}\ (\Uparrow_{f_{Var};\,[2]} \mathsf{CaseNat})\\
&\qquad\qquad\qquad\qquad \mathsf{Then\ Return}\ (\Uparrow_{[1;4]} \mathsf{AppACom\ RET})\ \emptyset\\
&\qquad\qquad\qquad\qquad \mathsf{Else}\ \ \mathsf{Return}\ (\Uparrow_{[2]} \mathsf{Reset})\ \lfloor\mathsf{true}\rfloor\\
&\qquad\quad |\ \lfloor\mathsf{LAM}\rfloor \Rightarrow \mathsf{Return}\ (\Uparrow_{f_{Var};\,[2]} \mathsf{ConstrS;\ AppACom\ LAM})\ \emptyset\\
&\qquad\quad |\ \lfloor\mathsf{APP}\rfloor \Rightarrow \mathsf{Return}\ (\mathsf{AppACom\ APP})\ \emptyset\\
&\qquad\quad |\ \emptyset \qquad\ \Rightarrow \mathsf{Return}\ (\Uparrow_{f_{Var};\,[3]} \mathsf{ConstrVar};\ \Uparrow_{1;3;4} \mathsf{AppCom})\ \emptyset\\
&\qquad\quad])\\
&\quad \mathsf{Else}\ \ \mathsf{Return\ Nop}\ \lfloor\mathsf{false}\rfloor
\end{aligned}$$

Input	CaseList	CaseCom	CaseNat	AppACom RET
$0 : P$	$0 : P'$			
$1 : Q$				$0 : Q +\!\!+ [\mathsf{RET}]$
$2 : k$			$0 : k'$	
$3 : \dashv$	$1 : \mathsf{RET}$	$0 : \dashv$		
$4 : \dashv$				$1 : \dashv$

Table 8.3: Execution protocol of SplitBodyStep for $P = \mathsf{RET} :: P'$ and $k = S\ k'$. The step machine terminates in the label $\emptyset$, thus the loop continues. Note that tape 4 is only used as an internal tape for AppACom.

Input	CaseList	CaseCom	ConstrVar	AppCom
$0 : P$	$0 : P'$			
$1 : Q$				$0 : Q +\!\!+ [\mathsf{VAR}(n)]$
$2 : k$				
$3 : \dashv$	$1 : \mathsf{VAR}(n)$	$0 : n$	$0 : \mathsf{VAR}(n)$	$1 : \dashv$
$4 : \dashv$				$2 : \dashv$

Table 8.4: Execution protocol of SplitBodyStep for $P = \mathsf{VAR}(n) :: P'$. The step machine terminates in the label $\emptyset$.

Execution protocols for the step machine are visualised in Table 8.3 and Table 8.4.

Note that the step machine resets the tape for k before it breaks out of the loop and returns true. Thus, no resetting is needed after the loop. The loop machine halts in true if and only if $\phi'\ k\ Q\ P$ is not $\emptyset$. If it is $\lfloor (P', Q') \rfloor$, then after the execution the first two tapes contain P' and Q', and the other tapes are right. If $\phi'\ k\ Q\ P = \emptyset$, the correctness relation only commits that the machine terminates in the label false.

Lemma 8.13 (Correctness of SplitBodyLoop**)** $\mathsf{SplitBodyLoop} \vDash \mathtt{SplitBodyLoopRel}$.

$$
\begin{aligned}
&\mathtt{SplitBodyLoopRel} :=\\
&\quad \lambda t\ (l, t').\ \forall P\ Q\ k.\ t[0] \simeq P \to t[1] \simeq Q \to t[2] \simeq k \to \mathtt{isRight}\ t[3] \to \mathtt{isRight}\ t[4] \to\\
&\qquad \mathsf{if}\ l\ \mathsf{then}\ \exists P'\ Q'.\ \phi'\ k\ Q\ P = \lfloor (Q', P') \rfloor \wedge\\
&\qquad\qquad t'[0] \simeq P' \wedge t'[1] \simeq Q' \wedge \big(\forall (i : \mathbb{F}_3).\ \mathtt{isRigth}\ t'[2 + i]\big)\\
&\qquad\quad \mathsf{else}\ \ \phi'\ k\ Q\ P = \emptyset
\end{aligned}
$$

Before the loop, the tapes for Q and k are initialised to nil and 0:

Definition 8.14 (SplitBody**)** *We define* SplitBodyLoop := While SplitBodyStep *and*

$$\mathsf{SplitBody} := \Uparrow_{[1]} \mathsf{ConstrNil};\ \Uparrow_{\mathtt{f_{Var}};\,[2]} \mathsf{ConstrO};\ \mathsf{SplitBodyLoop}$$

Lemma 8.15 (Correctness of SplitBody) $\mathsf{SplitBody} \vDash \mathtt{SplitLoopRel}$ *with*

$$
\begin{aligned}
&\mathtt{SplitBodyRel} :=\\
&\quad \lambda t\ (l, t').\ \forall P.\ t[0] \simeq P \to \mathtt{isRigth}\ t[1] \to (\forall (i : \mathbb{F}_3).\ \mathtt{isRigth}\ t[2+i]) \to\\
&\qquad \mathsf{if}\ l\ \mathsf{then}\ \exists P'\ Q.\ \phi\ P = \lfloor (Q, P') \rfloor \wedge\\
&\qquad\qquad t'[0] \simeq P' \wedge t'[1] \simeq Q \wedge \big(\forall (i : \mathbb{F}_3).\ \mathtt{isRigth}\ t'[2+i]\big)\\
&\qquad\quad \mathsf{else}\ \ \phi\ P = \emptyset.
\end{aligned}
$$

8.2.3 Step

We define a machine $\mathsf{Step}^{11}_{\Sigma} : \mathsf{TM}(\mathcal{O}(1))$ that simulates steps of the heap machine. The tapes 0, 1, and 2 contain the control stack T, the argument stack V, and the heap H, respectively. If the heap machine does a step $(T, V, H) \succ (T', V', H')$, Step terminates in $\emptyset$ and the resulting tapes contain T', V', H'. Otherwise, if (T, V, H) is a terminating state, Step terminates in the label $\lfloor () \rfloor$. Furthermore, if Step terminates in $\emptyset$, there is a successor state (T', V', H').

We implement auxiliary machines for each of the three step rules. Step first destructs T. For example, in the case that $T = (a, (\mathsf{APP} :: P)) :: T'$, the auxiliary machine for APP destructs V further and then calls a machine App that realises list-appending and a machine Lenght that computes the length of a list. After that, it pushes the new closures to T and V. As another example, if $T = (a, (\mathsf{VAR}(n) :: P)) :: T'$, the auxiliary machine for VAR calls Lookup and pushes the output closure to the argument stack and the reset closure to the control stack. If Lookup failed, Step immediately terminates in $\lfloor () \rfloor$, indicating that the state was a halting state.

The machine $\mathsf{Step} : \mathsf{TM}^{11}_{\Sigma^+}(\mathcal{O}(1))$ is defined on an arbitrary finite alphabet Σ^+ with retractions $f_{\mathtt{Stack}} : \Sigma_{\mathtt{List(Clos)}} \hookrightarrow \Sigma$ and $f_{\mathtt{Heap}} : \Sigma_{\mathtt{Heap}} \hookrightarrow \Sigma$. We omit the definition of Step here. Although it is quite complex, no new innovations are required, and the designing of the machine was reasonable easy. The machine first matches on the control stack. If it is empty, it immediately terminates in $\lfloor () \rfloor$.

Lemma 8.16 (Correctness of Step) $\mathsf{Step} \vDash \mathtt{StepRel}$ *with*

$$
\begin{aligned}
&\mathtt{StepRel} :=\\
&\quad \lambda t\ (l, t').\ \forall T\ V\ H.\ t[0] \simeq T \to t[1] \simeq V \to t[2] \simeq H \to (\forall (i : \mathbb{F}_8).\ \mathtt{isRight}\ t[3+i]) \to\\
&\qquad \mathsf{if}\ l = \emptyset\ \mathsf{then}\ \exists T'\ V'\ H'.\ (T, V, H) \succ (T', H', V') \wedge\\
&\qquad\qquad t'[0] \simeq T' \wedge t'[1] \simeq V' \wedge t'[2] \simeq H' \wedge \big(\forall (i : \mathbb{F}_8).\ \mathtt{isRight}\ t'[3+i]\big)\\
&\qquad\quad \mathsf{else}\ \ \mathtt{halt}(T, V, H)\ \wedge\ T = \mathsf{nil} \to\\
&\qquad\qquad t'[0] \simeq \mathsf{nil} \wedge t'[1] \simeq V \wedge t'[2] \simeq H \wedge \big(\forall (i : \mathbb{F}_8).\ \mathtt{isRight}\ t'[3+i]\big)
\end{aligned}
$$

For each developed machine, we also have running time proofs, but we omitted the running time lemmas for shortness. We have a function

$$\mathsf{stepSteps} : \mathcal{L}(\mathsf{HClos}) \to \mathcal{L}(\mathsf{HClos}) \to \mathsf{Heap} \to \mathbb{N}$$

such that Step terminates in $\mathsf{stepSteps}\ T\ V\ H$ steps when the first three tapes contain T, V, H, and the internal tapes are right.

Lemma 8.17 (Running time of Step) $\mathsf{Step} \downarrow \mathsf{StepT}$ *with*

$$\begin{aligned}\mathsf{StepT} := \lambda t\ k.\ \exists T\ V\ H.\ t[0] \simeq V \wedge t[1] \simeq T \wedge t[2] \simeq H \wedge (\forall i : \mathbb{F}_8.\ \mathsf{isRight}\ t[3+i]) \wedge \\ \mathsf{stepSteps}\ T\ V\ H \leqslant k\end{aligned}$$

8.2.4 Loop

We instantiate $\Sigma := \Sigma_{\mathcal{L}(\mathsf{Clos})} + \Sigma_{\mathsf{Heap}}$ and define Loop := While Step. This machine terminates if and only if the heap state that is encoded on the tapes is a terminating state. Moreover, if the heap state terminates with an empty control stack (nil, V', H'), the tapes contain (nil, V', H') after the execution.

Lemma 8.18 (Correctness of Loop) $\mathsf{Loop} \vDash \mathsf{LoopRel}$ *with*

$$\begin{aligned}&\mathsf{LoopRel} := \\ &\quad \lambda t\ t'.\ \forall T\ V\ H.\ t[0] \simeq T \to t[1] \simeq V \to t[2] \simeq H \to (\forall (i : \mathbb{F}_8).\ \mathsf{isRight}\ t[3+i]) \to \\ &\qquad \exists T'\ V'\ H'.\ (T, V, H) \rhd^* (T', V', H') \wedge \\ &\qquad T' = \mathsf{nil} \to t'[0] \simeq \mathsf{nil} \to t'[1] \simeq V' \to t'[2] \simeq H' \to \big(\forall (i : \mathbb{F}_8).\ \mathsf{isRight}\ t'[3+i]\big)\end{aligned}$$

For the running time function $\mathsf{loopSteps}$, we need the functions step and isHalt from Lemma 8.6. The running time-function of Loop is defined per recursion over the number k' of reduction steps of the heap machine.

Lemma 8.19 (Running time of Loop) $\mathsf{Loop} \downarrow \mathsf{LoopT}$ *with*

$$\begin{aligned}\mathsf{LoopT} := \lambda t\ k.\ \exists T\ V\ H.\ \exists T'\ V'\ H'\ k'.\ (T, V, H) \rhd^{k'} (T', V', H') \\ t[0] \simeq T \wedge t[1] \simeq V \wedge t[2] \simeq H \wedge (\forall i : \mathbb{F}_8.\ \mathsf{isRight}\ t[3+i]) \wedge \\ \mathsf{loopSteps}\ T\ V\ H\ k' \leqslant k\end{aligned}$$

with

$$\begin{aligned}&\mathsf{loopSteps}\ T\ V\ H\ k := \\ &\quad \mathsf{match}\ k \\ &\quad [\,0 \ \Rightarrow \mathsf{stepSteps}\ T\ V\ H \\ &\quad |\,S\ k' \ \Rightarrow \\ &\qquad \mathsf{match}\ \mathsf{step}(T, V, H)\end{aligned}$$

```
  [ ⌊(T′, V′, H′)⌋ ⇒
    if isHalt(T′, V′, H′)
    then 1 + stepSteps T V H + stepSteps T′ V′ H′
    else 1 + stepSteps T V H + loopSteps T′ V′ H′ k′
  | ∅ ⇒ stepSteps T V H
  ]
] .
```

Lemma 8.18 says that if Loop with tapes that encode (T, V, H) terminates, then heap machine also terminates. Dually, Lemma 8.19 says that if heap machine terminates, so does Loop when we encode (T, V, H) on the tapes.

The last step of the reduction of the halting problem of the heap machine to the halting problem of multi-tape Turing machines is to define a function

$$\mathit{initTapes} : \mathcal{L}(\mathit{Clos}) \times \mathcal{L}(\mathit{Clos}) \times \mathit{Heap} \to \mathsf{Tape}_{\Sigma}^{11}$$

such that the first three tapes of $\mathit{initTape}(T, V, H)$ contain T, V, or H, respectively, and the rest tapes are right.

Definition 8.20 ($\mathit{initTapes}$)

$$\mathit{initTapes}(T, V, H) := \mathit{initValue}(T) :: \mathit{initValue}(V) :: \mathit{initValue}(H) :: \mathit{initRight}^{8}$$

With $\mathit{initValue}(x) :=$ midtape nil START $(\mathit{encode}(x) +\!\!+ [\mathsf{STOP}])$ *and* $\mathit{initRight} :=$ midtape nil STOP nil.

Theorem 8.21 (Halting problem reduction) *Let* (T, V, H) *be a heap state. Then*

$$\big(\exists T'\ V'\ H'.\ (T, V, H) \triangleright^{*} (T', V', H')\big) \leftrightarrow \big(\exists c\ k.\ \mathsf{Loop}(\mathit{initTapes}(T, V, H)) \triangleright^{k} c\big)$$

Chapter 9

Conclusion

We have formalised multi-tape Turing machines in Coq. We developed a framework for programming and proving correctness and time complexity of multi-tape Turing machines. We have demonstrated the power of this framework by implementing and verifying a multi-tape Turing machine that simulates an abstract machine. This machine is a variation of the heap machine in Kunze et al. [13]. The two variants differ in that programs in our version are linearised lists of commands. In [13], the authors show that their heap machines can simulate terms of the programming language L, which is a subset of the call-by-value λ-calculus. It should be easy to formalise the reduction from the heap machine in [13] to our version. By that, we would formalise the reduction from the halting problem of L to the halting problem of multi-tape Turing machines. This is, however, beyond the scope of this thesis. It is an ongoing research project [9], to implement a Coq library of undecidability reductions, and this thesis provides one step towards this goal. The reduction from the halting problem of single-tape Turing machines to the Post correspondence problem (PCP) has already been mechanised in Coq in [12].

Differences to other mechanisations of Turing machines We build on Asperti and Ricciotti's framework from [2] inside the theorem prover Matita, and initially ported their definitions of tapes and Turing machines to Coq. We find their inductive definition of tapes appealing, because of its symmetric and finite nature. Because of the symmetric nature of their definition of tapes, it was easy to define an operator Mirror that mirrors the transition function of a machine. This is in contrast to the implementation of tapes in [17], where tapes are split into two halves, and the right half contains the current symbol. The finite nature made it possible to define an always terminating machine that moves the head to the right (or using Mirror to the left) end of the tape. This is in contrast to [6], where tapes are implemented as infinite streams of symbols. Our framework implements five major improvements on [2]. (1) By introducing labelled machines we make it unnecessary to reason about concrete states of machines. The authors in [2] already note

that reasoning about internal states is tedious and therefore do not include the terminating state in their definition of realisation. However, they also need a separate definition of realisation that includes the terminating state. (2) We have introduced a notion of time complexity that relates the inputs to the number of steps needed for the computation. (3) By introducing an operator Switch that generalises sequential composition and conditional, we simplified the verification of both operators and also introduced a useful operator that was used throughout the thesis. (4) We implemented general lifting operators that make it possible to compose small machines (w.r.t. the alphabet and number of tapes) to fairly complex machines. At this point, composing compound machines is reasonably easy, but we (5) have introduced another layer of abstraction. We have made it possible to directly manipulate values of encodable types.

In [15], Norrish concludes that:

> If register machines are unappealing because of their general fiddliness, Turing machines are an even more daunting prospect.

Certainly, Turing machines are not appealing. We have spent considerable efforts (ca. one year) to make programming and verifying Turing machines feasible. Interestingly, we ended up at a point, where programming and verifying Turing machines can be (in some sense) *easier* than for register machines, because register machines are restricted to natural numbers. The on-paper design, implementation and verification of the simulator was finished in three weeks.

Duality of realisation and termination We noted that our concepts for correctness and time complexity are dual in a sense. The (weak) notion of realisation says that *if* the machine terminates, then the output is correct w.r.t. a correctness relation R. On the other side, a machine terminates in a termination relation T, if for all pairs of input tapes t and step numbers k that are in T, the machine terminates in k steps given the input t. Realisation is monotone (cf. Lemma 2.24), and termination is anti-monotone (cf. Lemma 2.26). We find it remarkable that we use an inductive correctness relation for While (cf. Lemma 4.20) and a co-inductive running time relation (cf. Lemma 4.23).

Similarity of realisation and Hoare logic As already noted in [6], the notion of realisation is similar to Hoare logic, that is widely used for program verification. For example, consider the Hoare proof rule for sequential composition and the corresponding relational rule (for unlabelled machines $M_1, M_2 : \mathsf{TM}^n_\Sigma$):

$$\frac{\{A\}\ P_1\ \{B\} \qquad \{B\}\ P_2\ \{C\}}{\{A\}\ P_1;\ P_2\ \{C\}} \qquad \frac{M_1 \vDash R_1 \qquad M_2 \vDash R_2}{M_1;\ M_2 \vDash R_1 \circ R_2}$$

Sequential composition of machines amounts to relational composition of correctness relations (cf. Lemma 4.6). We encode preconditions and postconditions inside correctness relations. This means that if the precondition does not hold for input tapes t, this implies $R\ t\ (l, t')$. We are not aware of a Hoare-style calculus for reasoning about termination in a concrete number of steps related to the input, that is dual to Hoare logic, like in our duality between realisation and termination.

Problems of the framework The biggest problem of this framework is that encodability of types can be ambiguous. For example, there are more than three ways how to encode natural numbers on the alphabet of the heap machine simulator (cf. Section 8.2.1). We had to mentally keep track of in which encoding a value is encoded on a tape, and to translate values from one encoding to another. The greatest part of the total compilation time (which is less then 5 minutes) consists of rewriting tapes. This could probably be further optimised.

Comparison of proof assistants Asperti and Ricciotti [2] propose the formalisation of Turing machines as a benchmark for comparing proof assistants. We think that the formalisation and usability of finite sets could be a benchmark for itself. However, the task "formalise Turing machines in proof assistant X" is rather broad. There are many mathematical formalisations of Turing machines, and some might be easier to implement in one or the other proof assistant. For example, Isabelle does not support dependent types, so this concrete formalisation of Turing machines in this thesis would not be possible in Isabelle. Dependent types are quite central in our formalisation of Turing machines. For example, defining Switch without them would probably be considerable harder.

Future work When we defined the notion of value-containment (cf. Section 7.1), we had future work in mind where we formalise space-usage of machines. We were careful to avoid memory-leaks in the machines, but have not yet formalised this aspect of correctness. We can strengthen the correctness relations with commitments about the space-usage of each tape. Asperti and Ricciotti's inductive definition of tapes is very helpful in this regard, because their tapes never decrease the number of symbols. This means that the total space usage of a tape is just the number of symbols on the tape. Our idea is that we parametrise the definition of value-containment over the length l of the "rest list" on the left, and write $t \simeq_l x$. Note that, by definition, there are no symbols beyond the stop symbol on the right side of the tape, so the total size of the tape only depends on the length of the encoding and the length of the left rest. For example, CaseNat does not change the amount of totally allocated symbols, but decreases the length of the encoding and increases the length of the rest by one. On the other side, ConstrS "consumes" one rest symbol, i.e. it decreases the length of the rest by one and increases the length of the encoding by one. Thus, if the rest is empty, ConstrS allocates one new symbol.

Further future work could be to show that the running time function of the simulator is polynomial w.r.t. the number of steps and the length of the encoding of the initial heap machine state, see [11]. We could also formalise the reduction from multi-tape Turing machines to single-tape Turing machines, and from single-tape Turing machines to single-tape Turing machines with a binary alphabet. The framework can be used to program other simulator machines, for example, for the "naive" substitution-based machine in [13]. We could implement a universal Turing machines as in [2], and formalise results of computationally and complexity theory, for example the undecidability of the halting problem and Rice's theorem. The opposite reduction from multi-tape Turing machines to L, i.e. programming an L expression that simulates multi-tape Turing machines, is also open for future work. This should be a "less daunting prospect", because there is a framework for verified extraction of Coq terms to expressions of L, see [8].

Appendix A

Implementation in Coq

We outline some pearls of the implementation of the framework of this thesis in the theorem prover Coq. We do not assume additional axioms. The implementation compiles with the Coq versions 8.7 and 8.8. We use proof scripts to derive proof terms, using Coq's standard tactic language *ltac*. The complete the source code can be downloaded from the homepage of this thesis:

`https://www.ps.uni-saarland.de/~wuttke/bachelor/`

Used Libraries

We make use of the *typeclass* mechanism built into Coq. Typeclasses are first-citizen objects in Coq, that may be parametrised over types and other typeclasses. Coq uses proof-search to infer instances of typeclasses. For that, the user has to declare definitions or lemmas as typeclass instances. We refer to Castéran and Sozeau [5], for a more thorough description of this feature.

Base Library and Finite Types We use a modified version of the library for the lecture *Introduction to Computational Logic* at Saarland University. This library extends the standard library of Coq with additional functions and automation for lists and decidable predicates. On top of this library, we use the library of finite types that was developed by J. Menz in his bachelor's thesis [14]. The following listing outlines the definition of decidable predicates and finite types:

```
Definition dec (P:Prop) := {P} + {¬ P}.
Definition eq_dec (X:Type) := ∀ x 1, dec (x=1).
Structure eqType := EqType {
  eqtype :> Type;
  decide_eq : eq_dec eqtype
}.
Canonical Structure eqType_CS X (A: eq_dec X) := EqType X.
```

```
Class finTypeC (type: eqType) : Type := FinTypeC {
  enum : list type;
  enum_ok : ∀ x : type, count enum x = 1
}.
Structure finType : Type := FinType {
  type :> eqType;
  class : finTypeC type
}.
Canonical Structure finType_CS (X:Type) {p : eq_dec X} {class :
    finTypeC (EqType X)} : finType := FinType (EqType X).
```

The coercions (:>) make it possible to use finite types L:finType as types, i.e. we can write x:L. The canonical structures enable automatic inference of the structure, for example there is a function index : ∀ L:finType, L → nat, and we can write index true.

Retraction Library We implemented a library for retractions. We use a typeclass for the existence of retractions between two types.

```
Section Retract.
  Variable X Y : Type.
  Definition retract (f : X → Y) (g : Y → option X) :=
    ∀ x l, g l = Some x ↔ l = f x.
  Class Retract := {
    Retr_f : X → Y;
    Retr_g : Y → option X;
    Retr_retr : retract Retr_f Retr_g;
  }.
End Retract.
Arguments Retr_f { _ _ _ }.
Arguments Retr_g { _ _ _ }.
```

The *Vernacular* command **Arguments** makes X, Y, and the instance contextual implicit. This means that Retr_f : X → Y, where X and Y are inferred from the context. We define instances for the retractions in Section 2.1.3.

The retraction composition operator is not defined as an instance, to avoid diverging proof search during the typeclass inference, because it can be applied arbitrary many times. Although it is not declared as instance, the retraction composition function ComposeRetract can be manually applied:

```
ComposeRetract : ∀ A B C : Type, Retract B C → Retract A B →
    Retract A C
```

We embed composition of retractions into retraction operators. For example, instead of defining a retraction RetrLft : X ↪ X + Y, we define (and declare as an instance) an operator on retractions:

```
Retract_inl : ∀ A B C : Type, Retract A B → Retract A (B + C)
```

Inhabited Types We also have a library for inhabited types. Whenever we need a semantically irrelevant value, we can just write default and Coq infers and inserts a value with the following typeclass:

```
Class inhabitedC (X : Type) := {
  default : X;
}.
```

Vectors and $\mathbb{F}_k$ We use the type constructors Vector.t and Fin.t from Coq's standard library. However, many basic lemmas and functions are missing for this type. We have a small library that adds the missing functions and lemmas. It also provides some tactics, for example to make case distinctions over Fin.t (S n). We also introduce notations for elements of $\mathbb{F}_k$.

smpl We use Sigurd Schneider's smpl plugin[1]. It lets the user add tactics to a database (similar to HintDb) and provides a tactic that applies the first applicable tactic from the database. We use this plugin for proof automation, more on that below.

Mechanisation of encodable types

The Coq implementation of the definition of multi-tape Turing machines is straightforward. Therefore, we omit code-listings here. The interested reader of the PDF version of this thesis may click on the definitions or lemmas, to get to the corresponding Coq code.

We also use typeclasses to implement the notion of encodable types:[2]

```
Class codable (sig: Type) (X: Type) := {
  encode : X → list sig;
}.
Arguments encode {sig} {X} {_}.
Coercion encode : codable >-> Funclass.
```

When cX : codable sig X, the above **Coercion** vernacular makes it possible to write cX x. This applies the concrete encoding function given by the instance cX. We prefer this notation over encode x, due to the fact that encodability is ambiguous, as noted in Section 7.1. We define the alphabets in Definition 2.8 inductively, and also declare their proofs of finiteness.

[1] https://github.com/sigurdschneider/smpl

[2] Note that for technical reasons we do not parametrise codable over sig : finType.

We realise the type of the extended alphabet $\Sigma^+ := \mathtt{boundary} + \Sigma$, where

$$\mathtt{boundary} ::= \mathsf{START} \mid \mathsf{STOP} \mid \mathsf{UNKNOWN}$$

is defined as an inductive type.

Automation

Our proof scripts make use of the feature of *existential variables*. An existential variable ?X is a type (that may contain variables referring to more existential variables) and the environment in which it should be typed. Existential variables are refined during unification (e.g. using the tactic "apply") and created with tactics like "eapply". Before the proof script can be finished and saved with the **Qed** command, all existential variables must be bound to values.

Our tactic TMSimp destructs conjunctive assumptions in all hypotheses (i.e. logical conjunctions and logical existentials). It also introduces and names tapes to tmid, tmid0, etc. Furthermore, it instantiates and rewrites with hypotheses of the form $H : \forall j.\ j \notin I \to \mathtt{tmid0}[j] = \mathtt{tmid}[j]$, that come from the correctness Lemma 5.6 of the tapes-lift operator.

The tactic modpon H instantiates assumptions of the form $H : \forall x\ \cdots.\ P \to \cdots \to Q$. For each premise that the tactic could not solve automatically, it creates a new subgoal with existential variables for the quantified variables.

The tactic TM_Correct instantiates the existential variable for the relation $?R'$ and proves $M \vDash ?R'$. For that, it applies all correctness lemmas of the primitive machines, control-flow operators, lifts, constructors and deconstructors, and some other auxiliary machines. The user can declare more correctness and running time lemmas to be used by TM_Correct. For that, we use the Coq plugin *smpl*. For example, the following code declares the correctness and running time lemmas of LiftTapes:

```
Ltac smpl_TM_LiftN :=
  lazymatch goal with
  | [ ⊢ LiftTapes _ _ ⊨ _] ⇒
    apply LiftTapes_Realise; [ smpl_dupfree | ]
  | [ ⊢ LiftTapes _ _ ⊨c(_) _] ⇒
    apply LiftTapes_RealiseIn; [ smpl_dupfree | ]
  | [ ⊢ projT1 (LiftTapes _ _) ↓ _] ⇒
    apply LiftTapes_Terminates; [ smpl_dupfree | ]
  end.
Smpl Add smpl_TM_LiftN : TM_Correct.
```

The tactic smpl_dupfree automatically proves that a vector is duplicate-free.

Note that we do not register correctness lemmas to TM_Correct that require semantical information. For example, the correctness Lemmas 7.48 (Reset_Realise) and 7.42 (CopyValue) are parametrised over the encoding of X. The user has to applying the lemmas with the correct encoding manually.

Example Correctness Proof in Coq

We use the general approach how to prove correctness (or running time) of a Turing machine, as described in Chapter 6. For example, consider the following goal (we use the user-defined Coq notation ⊨c(k) to mean $\vDash^k$).

```
============================
Add_Step ⊨c(9) Add_Step_Rel
```

The outline of the proof script is:

```
Proof.
  eapply RealiseIn_monotone.
  { unfold Add_Step. TM_Correct. }
  { cbn. reflexivity. }
  { (* ... *) }
Qed.
```

The tactic eapply RealiseIn_monoton applies Lemma 2.24, and creates existential variables ?R′ and ?k′ and three subgoals:

```
3 focused subgoals
(shelved: 2)

============================
Add_Step ⊨c(?k') ?R'

subgoal 2 is:
 ?k' ⩽ 9
subgoal 3 is:
 ?R' ⊆ Add_Step_Rel
```

After focusing the first subgoal, we unfold the definition of Add_Step:

```
============================
If (LiftTapes CaseNat [|Fin1|])
   (Return (LiftTapes Constr_S [|Fin0|]) None)
   (Return Nop (Some tt)) ⊨(?k') ?R'
```

The tactic TM_Correct automatically applies the correctness lemmas of the conditional, tape-lifting, etc. It instantiates ?R′ and k′:

```
?R' := (LiftTapes_Rel [|Fin1|] CaseNat_Rel|_true ∘
        (⋃f:1 LiftTapes_Rel [|Fin0|] S_Rel|_f)||_None)
       ∪
       (LiftTapes_Rel [|Fin1|] CaseNat_Rel|_false ∘
        (⋃f:1 Nop_Rel|_f)||_(Some tt))
?k' := 1 + CaseNat_steps + Nat.max Constr_S_steps 0
```

where CaseNat_steps is the constant number of steps required for CaseNat (i.e. 5) and Constr_S_steps is 3. By simplification, the second goal reduces to $9 \leqslant 9$, which is solved by reflexivity of $\leqslant$.

The main part of the proof is the third goal (with R' substituted). We prove it with the following proof script:

```
{
  intros tin (yout, tout) H. cbn. intros a b HEncA HEncB.
  cbn in *. destruct H; TMSimp.
  – modpon H. destruct b; auto.
  – modpon H. destruct b; auto.
}
```

After the introduction of the variables and hypotheses HEncA: tin[@Fin0] ≃ a, HEncB: tin[@Fin1] ≃ b, and H: R' tin (yout, tout), we make a case-distinction over H (note that the head symbol in the definition of R' is ∪, so H reduces to a disjunction). This gives two sub-goals. In both subgoals, we use the automation tactic TMSimp. The first subgoal is:

```
tin, tout, tmid : tapes (boundary + sigNat) 2
H : ∀ n : nat, tin[@Fin1] ≃ n →match n with
                                | 0 ⇒ False
                                | S n' ⇒ tmid[@Fin1] ≃ n'
                                end
H0 : ∀ n : nat, tin[@Fin0] ≃ n →tout[@Fin0] ≃ S n
a, b : nat
HEncA : tin[@Fin0] ≃ a
HEncB : tin[@Fin1] ≃ b
H0_0 : tmid[@Fin0] = tin[@Fin0]
H1_0 : tout[@Fin1] = tmid[@Fin1]
============================
match b with
| 0 ⇒ False
| S b' ⇒ tout[@Fin0] ≃ S a ∧tmid[@Fin1] ≃ b'
end
```

In this case, we know that a must be equal to S a' for some a'. The assumption H is automatically instantiated with a and the proof HEncA. We finish the goal by case-distinction over a. The second goal is analogous.

Even for complex machines, the correctness proofs in Coq follow this pattern. It is important to note that the structure of the proof always follows the structure of the machine. Running time proofs are analogous.

Lines of Code

In Table A.1, we summarise the numbers of Coq code. We used the tool `coqwc` to count the lines. The case-studies are under the horizontal line in the middle.

Module	Spec	Proof
Preliminary (incl. loop and relations)	176	84
Definition of Turing machines	430	194
Primitive Machines	122	34
Control-flow operators	425	383
Lifting	362	193
Simple Machines	380	278
Value containment	394	119
Copying and writing values	411	288
Alphabet-Lift with values	133	147
Deconstructors and constructors	486	482
Notations and tactics for compound or programmed machines	165	15
MapSum	47	110
Addition and Multiplication machines	181	298
List functions machines	326	456
Heap machine simulator	981	1040
Total	5019	4121

Table A.1: Lines of specification and proof code for the "modules" (with several source files)

The total number of library lines is 153 spec and 2638 proof. The total compilation time is circa 4:30 minutes.[3]

Using Coq's `Extract` mechanism, we compiled the Coq implementation to Haskell, and we were able to count the total number of states of the heap machine simulator: The alphabet of Loop consists of 30 symbols and it has 11537 states.

[3]Measured on the following hardware: `Intel(R) Core(TM) i7-4710MQ CPU`; 8 cores @ 2.50GHz; compiled on GNU/Linux with `make -j8` and Coq 8.8.1.

Bibliography

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

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

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

[4] George S Boolos, John P Burgess, and Richard C Jeffrey. Computability and Logic. 5th, 2007.

[5] Pierre Castéran and Matthieu Sozeau. A Gentle Introduction to Type Classes and Relations in Coq. Technical report, Technical Report hal-00702455, version 1, 2012.

[6] Alberto Ciaffaglione. Towards Turing computability via coinduction. *Science of Computer Programming*, 126:31–51, 2016.

[7] Edsger W Dijkstra. Go to Statement Considered Harmful. In *Software pioneers*, pages 351–355. Springer, 2002.

[8] Yannick Forster and Fabian Kunze. Verified extraction from coq to a lambda-calculus. In *Coq Workshop*, volume 2016, 2016.

[9] Yannick Forster and Dominique Larchey-Wendling. Towards a library of formalised undecidable problems in Coq: The undecidability of intuitionistic linear logic. *LOLA workshop 2018, Oxford, UK*, 2018. URL `https://www.ps.uni-saarland.de/~forster/downloads/LOLA-2018-coq-library-undecidability.pdf`.

[10] 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.

[11] Yannick Forster, Fabian Kunze, and Marc Roth. The strong invariance thesis for a λ-calculus. *LOLA workshop 2017, Reykjavik, Iceland*, 2017. URL `https://www.ps.uni-saarland.de/~forster/downloads/LOLA-2017-strong-invariance-thesis.pdf`.

[12] Yannick Forster, Edith Heiter, and Gert Smolka. Verification of PCP-Related Computational Reductions in Coq. In *International Conference on Interactive Theorem Proving*, pages 253–269. Springer, 2018.

[13] Fabian Kunze, Gert Smolka, and Yannick Forster. Formal Small-step Verification of a Call-by-value Lambda Calculus Machine. Technical report, arxiv 1806.03205, Jun 2018.

[14] Jan Christian Menz. A Coq Library for Finite Types. Bachelor's thesis, Universität des Saarlandes, 2016. URL `https://www.ps.uni-saarland.de/~menz/bachelor.php`.

[15] Michael Norrish. Mechanised computability theory. In *International Conference on Interactive Theorem Proving*, pages 297–311. Springer, 2011.

[16] The Coq Proof Assistant. `http://coq.inria.fr`.

[17] 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.