Modeling Computer Viruses

MSc Thesis (*Afstudeerscriptie*)

written by

Luite Menno Pieter van Zelst
(born June 1st, 1980 in Hengelo, Overijssel, Netherlands)

under the supervision of **Dr A. Ponse**, and submitted to the Board of Examiners in partial fulfillment of the requirements for the degree of

MSc in Logic

at the *Universiteit van Amsterdam*.

Date of the public defense:	**Members of the Thesis Committee:**
July 2nd, 2008	Dr P. van Emde Boas
	Dr B. Löwe
	Dr A. Ponse
	Dr P.H. Rodenburg

INSTITUTE FOR LOGIC, LANGUAGE AND COMPUTATION

Preface

About half a year ago, Alban Ponse, my thesis supervisor, suggested that the topic of 'computer viruses' might prove to be interesting, especially for a theoretician like me. A lot of work has been done in applied research; in comparison the field of virus theory has been gathering dust for the last 25 years.

It must be said that I was not immediately convinced. I have long since felt that threat of viruses is largely artificial; not only to the extent that viruses are artificially created and not a spontaneous occurrence of life. One cannot help but wonder about the apparent symbiotic relationship between the anti-virus industry and the creators of computer viruses. After all, the anti-virus industry stands to lose a lot: the projected market size of security software for mobile phones alone is five billion dollars in 2011 [43].

Moreover, while reading several articles on the theory of computer viruses, I got seriously disgruntled. The theory of computer viruses as it was presented, did not appeal to me. However, the signs were unmistakable: I only get disgruntled if I am passionate about a subject and do not agree with what I read; perhaps even think of another ways of doing things. My academic curiosity was peeked.

I guess any thesis will in a way reflect the character of the author. Personally I find it satisfying to have my bases covered. The entire first chapter is the direct result of saying to myself: "Ok, this paper I'm reading uses Turing Machines. Let's first write down a concise definition of Turing Machines". Of course I hope to convince the reader that I am not a windbag. There is just more to Turing Machines than meets the eye initially. It is true that in this chapter I digress a little from the topic of computer viruses. I truly believe, however, that the topic of Turing Machines is of separate interest to the theoretician, whether he be a logician or computer scientist.

In a way, my initial disgruntlement is expressed in Chapter 2, where I review a prominent modeling of computer viruses. The thesis then culminates in Chapter 3 where I do propose another 'way of doing things'.

The last six months have been a wondrous academic journey. It was a time of many academic firsts for me. About halfway through, I was invited to submit an extended abstract of this thesis for consideration of the third international workshop on the Theory of Computer Viruses [44]. I was duly accepted as speaker. Thanks to the support of the Section Software Engineering of the Faculty of Science, I was able to travel to Nancy, France and present my ideas. As a result I am invited to submit a 15 page version of this thesis for publication in the Journal of Computer Virology [41]. This version if forthcoming. Being passionate about the results, it is a thrilling experience to convey them to an audience that does research in the same field.

Acknowledgements

I would like to thank my supervisor Alban Ponse for all his help and support. I could not have wished for a more dedicated supervisor. Our meetings were not only productive but also thoroughly enjoyable.

Furthermore, I would like to thank to several authors around the world for their kind replies to my queries. In particular: Fred Cohen, Jack B. Copeland and Jean-Yves Marion. We may at times agree or disagree, academically, but we do so in a spirit of friendship.

The gratitude I feel towards my family, especially my wife Karen and my father, is ill expressed in a few lines. Their love and support have been invaluable.

Luite van Zelst,
Utrecht, the Netherlands, June 2008.

Contents

Introduction

Computer viruses tap into a primordial human fear: that of loosing control. The public fascination with viruses exceeds our fascination with other ways in which we lose control over our computers, for example random hardware faults. A probable cause is the perceived prevalence of computer viruses: one expert website indicates that in March 2008 the 'NetSky' virus was at 22 percent (!), although without enlightening the reader what that means [45]. Another reason may be the way that the virus disaster strikes: through infection. We feel that we ought to be able to prevent it.

What simple rules should you and I follow to prevent our computers from being infected? Here is a common view: buy and install a virus scanner, adware scanner, email scanner, firewall, etc. On top of that, we might want to tell a commercial company what websites we visit, in order to prevent visiting 'suspect sites'. In short, we are willing to hand over our money and our privacy to feel more secure.

Still disaster may strike. And perhaps we have good reason to be pessimistic, for it has long since been established that "virus detection is undecidable". However, we would like to impress upon you that such a claim is very imprecise and based on a mathematical abstraction. In its entirety, the claim should be read as: "given any machine of infinite capacity and any program, can we decide whether the program is a virus for that machine?" The claim then is that we cannot. This might lead us to be overly pessimistic when we consider the question whether for a specific type of computer we can detect viruses within a specific class of programs.

The claim that virus detection is undecidable is based on one seminal work. In 1985 Fred Cohen wrote his doctoral thesis entitled "Computer viruses". He was the first to use the words "computer viruses" and he proved the undecidability of virus detection. Cohen based his 'undecidability' result on a formal model of computer viruses. In his model, viruses are represented as 'sequences of symbols' and computers are represented as 'Turing machines'. Are these representations appropriate? For if they are not so, Cohen's undecidability result is on loose ground.

Contents

The purpose of this thesis is to examine the following question: "How can we adequately model computers and computer viruses?". To that end we ask ourselves: Are computers adequately represented by Turing machines? What ingredients are essential for modeling computers? What are computer programs? What are computer viruses?

We have passed the 70$^{\text{th}}$ anniversary of Turing's famous 1936 article which first introduced a machine we now call the 'Turing machine' [28]. Since its publication, the concept of Turing machine has become mainstream and is presented in most computer science textbooks. The fact that every author presents his own flavour of Turing machines has lead to a sense of familiarity and confusion. Usually this is no obstacle: we have come to believe that the differences are inessential. Recently, when the results from this thesis were presented at a workshop, various members of the audience asked: "Why, aren't all Turing machines essentially the same?"[1] We will show in Chapter 1 that such a view is overconfident. To that end we rigorously define our own flavour of Turing machines and discuss what choices can make crucial difference.

[1] At the third international workshop on the Theory of Computer Viruses (TCV'08), at Loria, Nancy, France [44].

The fact that "all Turing machines are not the same" is also appreciated when we realise that the perceived view of Turing machines is radically different from what we may read in Turing's original article. Not only do the two kinds of machines differ in the way they work, we also interpret their 'computation' differently. It is a purpose in itself to bring this fact out into the open. Even though it is slightly off topic with respect to computer viruses, we pursue the investigation of Turing's original machines. This leads us to propose a new machine model that can unify the perceived view with Turing's work. We refer to the new machine as a 'Talkative Machine'.

Other machines that resemble Turing Machines (but are used and interpreted differently) include: 'Universal Turing Machines' (UTMs) and 'Persistent Turing Machines' (PTMs). As these models have a bearing on the definition of computer viruses, we will lay out the groundwork at the end of Chapter 1.

Does any kind of Turing Machine model modern computers? To answer that question we first have to establish what a computer is, exactly. Unfortunately, that is not a very exact subject matter. In Chapter 2 we will first try to establish the minimal elements any computer model should contain in order to sensibly define computer viruses. We will even tempt the reader to convince himself (herself) that a 'model of computation' (such as Turing Machines) is not the same as a 'model of computers'. From the minimal ingredients of a computer model we will argue that, in general, it is not appropriate to model viruses on the basis of Turing machines.

At this point we can discuss several attempts to do just that: combine viruses with Turing Machines. Our foremost interest is in the work of Cohen [5, 16]. Cohen's work merits special attention because it is the first academic research into computer viruses. Moreover, his central 'undecidability result' is often cited, and so far still stands. Surprisingly, an in depth review of his definitions is untrodden territory.

The fact that Cohen's work has not yet been thoroughly discussed, has not stopped other authors from suggesting different frameworks for the definition of computer viruses [17, 18, 21, 32, 33]. One avenue of approach can be considered 'variations on a theme of Cohen'. Among these approaches, we will discuss the definition of viruses based on UTMs [18] and PTMs [33]. We seem to be the first to discuss the Universal Protection Machine (UPM), a Turing Machine based computer model proposed by Cohen in his PhD thesis [5] (and not published elsewhere). We hope that our discussion compels the reader to be convinced that Turing Machines (and their derivatives) are unsuitable for modeling computer viruses.

At this point, you might well ask: "Then what is a suitable basis to define computer viruses?". Chapter 3 proposes just such a basis. We will introduce a new kind of machine which we call "Recursion Machine". Properly speaking it is perhaps not a 'machine', but rather a way to model computers using the theory of recursive functions. (We extend the theory of recursive function from the domain of natural numbers to arbitrary enumerably infinite domains.) At this higher level of modeling, we are not concerned with a specific implementation or with a specific representation of programs. Nevertheless we can model computers, computer viruses and some defenses against viruses. We show that we can apply the definition of viruses proposed by Adleman in our framework [17].

To Dr. Vincent van Oostrom,

who opened my eyes to the beauty of logic
and inspired me to be inquisitive and free-thinking.

Chapter 1

Computing Machines

1.1 Introduction to Turing Machines

To formalise the notion of computability, Turing introduced a machine we have come to call the 'Turing Machine' [28]. The merit of the Turing Machine is that it is easy to grasp the 'mechanical way' such a machine could work.

A Turing Machine consists of a tape, a tape head and the machine proper (which we will call 'control machine'). The tape is infinitely long and consists of discrete 'cells', also referred to as 'squares'. Each cell on the tape can contain a symbol. The tape head is at one cell at each moment, and can move one cell to the left or to the right.[1] The tape head can recognize (or 'read') the symbol in the cell and write a single symbol back to the tape cell.

The control machine determines what the Turing Machine does. It is a mechanical device that may be constructed in *any way* from finitely many small parts. The tape head and tape are driven by the control machine (and must be engineered to comply). We can describe the behaviour of the control machine as a set of rules of the form "if M is in machine state q and the tape head reads symbol s then write symbol s', move the tape head one position to the left or right (or stay), and enter into state q'".[2]

To convince you that such a mechanical machine could be build, we have to address the issue of building a tape that is infinitely long. Obviously, we cannot do that. But we can construct a tape that is 'long enough', even if we do not know how long it should be. We just make the tape longer every time it turns out it is not long enough. [3]

A Turing Machine in such mechanical terms does not allow for abstract reasoning. As mathematicians, we abstract from the (mechanical) machine and try to capture its functioning. There are probably as many ways to describe Turing Machines as there are mathematicians. Many descriptions of Turing Machines gloss over the particulars, with the effect that the proofs involving Turing Machines often amount to little more than hand waiving.

We will give a short overview of formalisations of Turing Machines in the literature. Astonishingly, the machines defined in most literature have a very different interpretation from the original interpretation in Turing's 1936 article. We will need more mathematical machinery to discuss Turing's interpretation, so we leave this to Section 1.3. For now, we start with one of the earliest works (if not the first) with the now

[1] A different point of view is that the tape head is at a fixed position and that is the tape that moves under the head. This distinction is immaterial.

[2] Turing envisaged machines that can write *or* move; he also envisaged machines that must write *and* move or stay. We have no reason to doubt his claim that the difference is inessential. We choose to define machines that write *and* move or stay, throughout the text.

[3] We would like to draw a parallel with the Eisinga Planetarium in Franeker, the Netherlands. In 1774, Eise Eisinga build a mechanical planetarium to predict the positions of the planets. Part of the mechanism of the planetarium is a moving plank that shows the number of the current year. To keep the planetarium running, the plank should be extended to include at least the current year. Thinking ahead, he left instruction on how to update the machine to keep it running ad infinitum. Fortunately, rather than instructing his successors to extend the plank into the neighbouring house, he thought of a more practical solution: take the plank out once every 22 years and paint new years over the old ones. The machine has now been in operation for more than two centuries. See [40] for more information about the planetarium.

common interpretation.

1.1.1 Turing Machines as Quadruples

In *Computability and unsolvability* (1958) Martin Davis very carefully constructs his 'Turing Machines' [1]. He stresses that it is important that

> ... the reader carefully distinguish between formal definitions and more or less vague explanations intended to appeal to intuition, and, particularly, that he satisfy himself that only formal definitions are employed in actual proof.

Figure 1.1 depicts how a Turing Machine functions according to Davis. Taking this advise to heart, we now reproduce his abstraction of Turing Machines.

Definition 1.1 (Quadruple)**.** Given the sets of symbols $Q = \{q_1, q_2, q_3, \ldots\}$, $\Sigma = \{s_0, s_1, s_2, \ldots\}$ and $M = \{L, R\}$ we define a (Turing Machine) quadruple as $\left(q_i, s_j, A, q_l\right)$ such that $q_i, q_j \in Q$, $s_j \in \Sigma$ and $A \in \Sigma \cup \{L, R\}$.[4]

Definition 1.2 (Turing Machine [1, Definition 1.3])**.** A Turing Machine Z is a finite, nonempty set of (Turing Machine) quadruples that contains no two quadruples whose first two symbols are the same.

Definition 1.3 (Instantaneous description [1, Definition 1.4])**.** An instantaneous description is a triple (T, q_i, T') such that $T, T' \in \Sigma^*$ and $q_i \in Q$. We call the set of all instantaneous descriptions ID.

The 'Turing Machine' Z is meant to capture the mechanical working of a Turing Machine. The instantaneous description is meant to represent the "entire present state of a Turing Machine", capturing the symbols on the tape, the machine state and the position of the tape head. The tape is represented by two finite strings: T and T'; their concatenation represents a finite part of the tape. To represent a longer part of the tape, the string may grow by concatenating it with symbol s_0; this symbol may be thought of as representing a blank square on the tape.

Combining machine and description, Davis defines a transition from one description to the next according to the rules (quadruples) in Z.[5] Written as $a \hookrightarrow_Z b$, the transition captures the 'functioning' of a Turing Machine.[6]

On top of these transitions, Davis defines the central notion of Turing Machines: computation.

Definition 1.4 (Computation of a Turing Machine [1])**.** A computation of a Turing Machine Z is a finite sequence $seq : \alpha \to ID$ such that:

$$(\forall\, i < \alpha - 1)\,(seq(i) \hookrightarrow_Z seq(i+1)) \qquad \text{and}$$
$$(\nexists x \in ID)(seq(\alpha - 1) \hookrightarrow_Z x)$$

We can introduce short notation for an entire computation by defining $a \to_Z b$ if and only if there is a computation $s : \alpha \to ID$ such that $a = s(0)$ and $b = s(\alpha - 1)$.

This is one of the points where Davis diverges from Turing's original approach [28]. Turing start his machine with an empty tape and with the machine in an initial internal state. The above abstract notion of computation leaves us free to choose a sequence with a first instantaneous description that does not represent an empty tape (or has another initial state). We can think of Davis' 'non-empty' descriptions as input. In terms of a concrete mechanical Turing Machine, the input should be written to the tape by some mechanism that is *external to the machine*.

[4] See [1, Definitions 1.1 and 1.2]. This is an example of a definition that allows the machine *either* to write a symbol *or* to move, but not both at the same time. For simplicity, we leave out the fourth quadruple which Davis discusses in chapter 1, section 4 and is intended for the definition of oracle-machines.

[5] For the full specification/definition, see [1, Definition 1.7].

[6] Davis writes $\alpha \to \beta(Z)$, the difference in notation is typographical.

Figure 1.1 – Three stages of a Turing Machine according to [1]

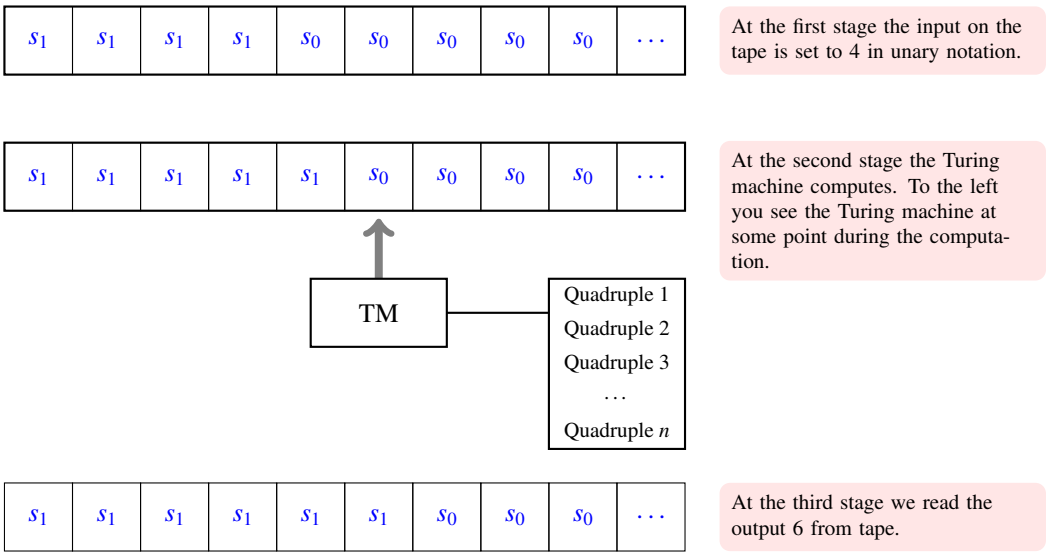
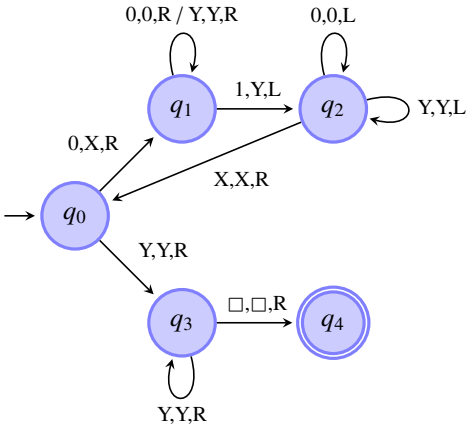


Figure 1.2 – A Turing Machine viewed as a finite state automaton with a tape



To the left we see a picture of a Turing Machine as automaton (based on an example in [4, Example 7.1]). It is designed to 'accept' an input string of the form 0^n1^n ($n \geq 1$) followed by at least one '□' (blank) symbol.
It starts in state q_0 and accepts the input if it enters state q_4. Every transition is labelled with three values: the symbol read on the tape, the symbol written back to tape and the move left or right.

Davis' abstraction is an elegant one, because to specify a machine it suffices to specify a finite set of quadruples. We need not specify the sets of state and tape symbols, they are subsumed by the set of quadruples. We can use this to our advantage when we want to compose computations of (possibly different) machines. For example, we can simulate the computation starting from a non-empty tape by a computation starting from an empty tape, using a composition:

$$\epsilon \rightarrow_I i \rightarrow_Z r$$

This composition should be read as: from the instantaneous description ϵ representing an empty tape with the tape head at the first square, we can 'compute' (by some machine I) the instantaneous description i that serves as input for the computation by Z which results in a description r.

Finally, Davis gives an example of how to give meaning to a computation. For a computation $b \rightarrow_Z c$, if b and c are of a specific format, we can interpret the computation as *computing a function* (from natural number to natural numbers).

We would like to hold up *Computability theory* by Cooper [12] as an example of *how not to* define Turing Machines. Mostly, Cooper follows Davis' abstraction, but he does not take Davis quote to heart. Cooper fails to mention instantaneous descriptions or define computations. Most strikingly, Cooper merely claims:

> How does T compute? Obviously, at each step of a computation, T must obey any applicable quadruple.

1.1.2 Turing Machines as Finite State Machines

The most common approach to defining Turing Machines is to view a Turing Machine as a finite-state machine with an infinite tape [2, 4, 9, 10]. The internal 'states' of the Turing Machine are regarded as the states of a finite automaton, and the change of state is viewed as a transition. This view allows the authors to express the program (mechanical construction) of the machine as a 'transition function' rather than as a set of quadruples.

For Davis the internal states have no intrinsic meaning. Finite state machines, however, may have states that have special meaning, and it is observable that the machine enters into such a state. For example, the transition into a state 'h' may be a signal that computation is over. We could also define states q_x that signal "the input has property x", or states q_y that signal that the input "does not have property y". The most common special states are 'accepting states' (meaning "the input is accepted") and 'rejecting states' (meaning "the input is not accepted"). Figure 1.2 shows a machine with a single 'accepting state'.

The definition of such a Turing Machine is along these lines. A machine M is a structure $(Q, A, R, H, \Sigma, \Gamma, q_0)$ such that:

- Q is a finite set of machine states.
- $A \subseteq Q$ is the set of 'accepting states'. If the automaton enters into an 'accepting state' the input string is said to have been accepted by the machine; the input string has a certain property.
- $R \subseteq Q$ is the set of 'rejecting states'. If the automaton enters into a 'rejecting state' the input string is said to have been rejected by the machine; the input string does not have a certain property.
- $H \subseteq Q$ is the set of 'halting states'. If the automaton enters into a 'halting state' the machine is said to have computed a function on the input string. The symbols that are on the tape at that time, is said to be the output.
- Σ is the 'input alphabet'. Symbols in Σ may be on the tape when the machine is started.
- The tape alphabet $\Gamma \supseteq \Sigma \cup \{\square\}$ contains at least the input alphabet and a distinct 'blank' symbol. Symbols from Γ may be written to the tape during computation.

- q_0 is the 'starting state'; when the machine is started the automaton is always in this starting state.
- The machine has a transition function $tr \subseteq (Q \backslash H) \times \Gamma \to Q \times \Gamma \times (\text{move left, stay, move right})$. Given a state (not a halting state) and a symbol on the tape it provides a new state (possibly a halting one), a symbol to write on the tape and a direction for the tape head to move.

So far we have only defined a structure, not captured 'Turing Machine computation'. Unfortunately, some authors do not define computation at all.[7] Authors that do define 'computation' do so based on their own particular flavour of instantaneous descriptions.[8] At times computation is called 'yielding' [9] or '(finitely many) moves' [4].

1.2 Turing Machines Redefined

We propose to reexamine the definition of a Turing Machine. The authors mentioned in the previous section vary in their definitions on some subtle issues. Do we need a special 'left-most square' symbol (as used only in [9, 28])? What is the status of the special 'blank' symbol ($\square$)? In particular: Should we be able to write blanks? Should we delete trailing blanks after each move (as in [4])? Can (and should) infinite tapes be represented as finite strings?

In this section we will answer such questions. To that end, we will construct a mathematical definition of Turing Machines from the ground up.

1.2.1 Turing Machine Tape

We will start by defining what a Turing Machine tape is, what cells on the tape are, and what content of the cells is. We will draw on the readers mathematical intuitions to support the choices we make. In a general sense we can call the tape a 'memory' which consists of infinitely many cells.[9] We define the memory (and express its infinite size) as:

$$M \quad \text{is an infinite set} \tag{1.1}$$

Each cell on the tape has content. We call the set of allowed content the 'base set' [34]. This will have the same function as the 'input alphabet' and is a finite set

$$B = \Sigma = \{b_0, \ldots, b_n\} \text{ (for some } n \in \omega) \tag{1.2}$$

We can map each cell on the tape to its content with a total function

$$S : M \to B \tag{1.3}$$

Different maps S indicate differences in tape content. For given sets M, B the set of all maps S determines all the possible configurations of the tape:

$$\mathbb{S}_{M,B} = \{S \mid S : M \to B,\ S \text{ is total}\} \tag{1.4}$$

A tape has a left-most cell and with two cells a and b either a is 'to the right of' b or vice versa; therefore we say that the tape is 'totally ordered' and that the tape memory M is countable. This means we can use the ordering of natural numbers to access the tape memory, so there is a bijection:

$$m : \omega \leftrightarrow M^{\dagger} \tag{1.5}$$

Now we can talk of the n^{th} element of the tape as $m(n)$ and of the content of the n^{th} cell as $S(m(n))$. However, in view of the fact that this property holds for the tape of any Turing Machine, we disregard M and m and redefine the set of tape configurations so that the content of the n^{th} cell is just written as $S(n)$.

[7] In this respect [11, Definition C.1] is exemplary, as it only provides the definition of this structure and gives an example of how the transition function 'is used'. Even so, the authors feel this is sufficient explanation to go on to state Church's thesis. To be fair to the authors of this otherwise excellent book, Turing Machines are defined in an appendix, being a side issue to the subject matter.

[8] Except for [2], where we see only a graphical representation of a part of a two-way infinite tape.

[9] This construction is inspired by the notion of Maurer computers [34] and closely resembles some constructions in [35].

[†] We will use ordinals throughout this thesis. For more information see Appendix A.

Definition 1.5 (Tape configurations)**.** For any base set B the set of tape configurations is defined as the set of all infinite B-sequences $\mathbb{S}_B = \{S \mid S : \omega \to B,\ S \text{ is total}\}$.

We would also like to be able to talk about a cell as 'not having content'. We do so be stating that a cell contains the special symbol $\square$ ('blank') that is not in the input alphabet. So we change our definition of B:

$$B = \Sigma \cup \{\square\} = \{\square, b_0, \ldots, b_n\} \text{ (for some } n \in \omega) \tag{1.6}$$

Using the 'blank' symbol, we can define the 'amount of content' on the tape as the 'size'. The intuition is that if there are finitely many non-blank symbols on the tape, there must be a point on the tape where all following symbols are blank.

Definition 1.6 (Size)**.** We define the size to be either finite (in ω) or infinite (the size is ω).

$$\begin{aligned} \text{size} &: B^\omega \to (\omega + 1) \\ \text{size}(S) &= \text{minimal } n \in N_{empty} \\ N_{empty} &= \{n \in (\omega + 1) \mid (\forall\, n' \geq n)\,(S(n') = \square)\} \end{aligned} \tag{1.7}$$

Note that size(S) is well defined since

1. $\omega \in N_{empty}$ (i.e. it has at least one element)
2. $N_{empty} \subseteq (\omega + 1)$ (i.e. it is well ordered)

Definition 1.7 (Finite content)**.** For a tape alphabet Σ and base set $B = \Sigma \cup \{\square\}$, we define $\mathbb{F}_B$ as the set of tape configurations with finite amount of content:

$$\mathbb{F}_B = \{S \in \mathbb{S}_B \mid \text{size}(S) < \omega\} \tag{1.8}$$

The finite content of sequences in $\mathbb{F}_B$ may still be scattered on the tape, i.e. there may be blanks before the last content symbol.

Definition 1.8 (Pure content)**.** We say that a tape configuration has pure content if the content is finite and contiguous. We define the set of tape configurations with pure content as

$$\mathbb{P}_B = \{S \in \mathbb{F}_B \mid (\forall\, n < \omega)\,(S(n) \neq \square \ \vee n \geq \mathit{size}(S))\}$$

There is a bijection between the set of tape configurations with pure content and the set of words (over the same alphabet), as seen in the following lemma. We can interpret this result as: words can be uniquely extended to tape configurations with pure content by adding infinitely many blanks.

Lemma 1.9. *Let $B = \Sigma \cup \{\square\}$. Then $\mathbb{P}_B \leftrightarrow \Sigma^*$.*

Proof. We define the following functions.

$$\begin{aligned} \mathit{head} &: \mathbb{P}_B \to \Sigma^*, \\ &\quad S \mapsto S \restriction \text{size}(S) \\ \mathit{ext} &: \Sigma^* \to \mathbb{P}_B, \\ &\quad t \mapsto t \cup \{(\alpha, \square) \mid \mathit{dom}(t) \leq \alpha < \omega\} \end{aligned}$$

Let $t \in \Sigma^*$ and $S \in \mathbb{P}_B$. Then

$$\begin{aligned} (\mathit{head} \circ \mathit{ext})(t) &= \mathit{head}(t \cup \{(\alpha, \square) \mid \mathit{dom}(t) \leq \alpha < \omega\}) \\ &= t \restriction \text{size}(t) \cup \{(\alpha, \square) \mid \mathit{dom}(t) \leq \alpha < \omega\} \restriction \text{size}(t) \\ &= t \cup \emptyset = t \\ (\mathit{ext} \circ \mathit{head})(S) &= \mathit{ext}(S \restriction \text{size}(S)) \\ &= S \restriction \text{size}(S) \cup \{(\alpha, \square) \mid \mathit{dom}(S \restriction \text{size}(t)) \leq \alpha < \omega\} \\ &= S \restriction \text{size}(S) \cup \{(\alpha, \square) \mid \text{size}(t) \leq \alpha < \omega\} \\ &= S \restriction \text{size}(S) \cup \{(\alpha, S(\alpha)) \mid \text{size}(t) \leq \alpha < \omega\} \\ &= S \restriction \text{size}(S) \cup\ (S \setminus S \restriction \text{size}(t)) = S \end{aligned}$$

Then *ext* is the inverse of *head* and both are a bijection. □

1.2.2 Turing Machine Computations

We are now ready to define Turing Machines. In the previous section we discussed infinite tapes; this is the first ingredient for Turing Machines. The second ingredient is a set of 'machines states'. It does not matter what a state *really is*; what matters is that we have several. Machines states can be thought of as the machine analogy of human emotions: without fully understanding what the emotions are, we do know that we might do different things depending on our 'state of mind'.

Definition 1.10 (Turing Machine)**.** A Turing Machine is a structure (Q, Σ, tr, q_0) where Q is a finite set of states, $q_0 \in Q$ and tr is the transition function:

$$tr : Q \times (\Sigma \cup \{\square\}) \to Q \times \Sigma \times \{-1, 0, 1\} \tag{1.9}$$

The 'state' of a Turing Machine is not fully captured by the states in Q, but by 'configurations' which we define to be triples (q, t, p) where:

- $q \in Q$ is the current state,
- $t \in \mathbb{S}_{\Sigma\cup\{\square\}}$ is the current content of the tape,
- $p \in \omega$ is the current position of the tape head.

Definition 1.11 (Configurations)**.** Let $M = (Q, \Sigma, tr, q_0)$ be a Turing Machine. The set of all configurations of M is defined as:

$$\mathbb{C}_M = \{(q, t, p) \mid q \in Q, t \in \mathbb{S}_{\Sigma\cup\{\square\}}, p \in \omega\}$$

The transition relation of a Turing Machine determines how a machine goes from one configuration to the next. We call this transition a 'move'.[10] The constraint on the transition is that it respects changes dictated by the transition function: the change of state $\in Q$, the new tape head position $\in \omega$, and the update of the tape square.

Definition 1.12 (Moves)**.** Moves are the binary relation on $\mathbb{C}_M$ defined by:

$$(q, t, p) \hookrightarrow_M (q', t', p') \quad \iff \quad tr(q, t(p)) = (q', t'(p), p' - p) \quad \wedge \quad t' = t\,[p \mapsto t'(p)]$$

The definition of moves only tells us *how* transitions occurs, not *which* transitions occur. To know that, we have to know which configuration we start out with. In other words: we have to agree how we start a Turing Machine. The first choice to make is: what 'input' (or sequence of symbols) is on the tape when the machine starts.

Definition 1.13 (Input convention)**.** Let $M = (Q, \Sigma, tr, q_0)$ be a Turing Machine. It is determined by some mechanism external to the Turing Machine (i.e. by us) which is the initial configuration. By convention, the Turing Machine starts in the start state $q_0 \in Q$, the tape head is at the left-most position and the tape contains pure content, which we call the 'input'. Thus the initial configuration of M is fully determined by the chosen input *in*, written as:

$$(q_0, in, 0) \text{ for some } in \in \mathbb{P}_{\Sigma\cup\{\square\}}$$

Remark 1.14. When we think of the Turing Machine as a mechanical device, the input is placed on the tape by some external mechanism (or person, e.g. by 'us'). We maintain that such an 'input mechanism' is not part of the mathematical definition of the Turing Machine, in a strict sense. Rather, such a mechanism is translated to a 'convention' that we should agree upon and adhere to. Admittedly, we could take the view that saying "machine M start with input *in*" can be translated to "there is a machine M' that starts with an empty tape, writes *in* and then performs like M". But this is an input convention itself, for it is not in the definition of M how to construct M' nor that M' starts with an empty tape.

[10] Similar to "microsteps" in [26] and to 'yields' in [9, 10]. We stick to the original name 'moves' from [28, Section 1].

We can now 'start' a Turing Machine on input *in*. Let us see what happens next. Observe that starting a Turing Machine with input *in* on the tape, gives rise to a unique sequence of configurations connected by moves. We introduce notation for this sequence.

Definition 1.15 (Configuration sequences)**.** Let M be a Turing Machine and $\mathbb{C}_M$ be the set of configurations of M. We write $\text{seq}_M(in)$ for the unique maximal sequence of configurations $\text{seq}_M(in) : \alpha \to \mathbb{C}_M$ such that

$$\text{seq}_M(in)(0) = (q_0, in, 0) \quad \wedge \quad (\forall\, \beta < \alpha - 1)\,(\text{seq}_M(in)(\beta) \hookrightarrow_M \text{seq}_M(in)(\beta + 1))$$

What is the length of such a sequence?

Definition 1.16 (Computation length)**.** Let M be a Turing Machine. We write $\text{moves}_M(in)$ for the length of the maximal sequence of configurations of M with input *in*, i.e. $\text{moves}_M(in) = dom(\text{seq}_M(in))$.

Definition 1.17 (Halt, diverge)**.** Let M be a Turing Machine. We say that M halts for input *in* if $\text{moves}_M(in) < \omega$. We say that M diverges for input *in* if $\text{moves}_M(in) = \omega$. An alternative (but equivalent) formalisation is to say that M halts for input *in* if:

$$(\exists \alpha < \omega)\,(\nexists\, c \in \mathbb{C}_M)\,(\text{seq}_M(in)(\alpha) \hookrightarrow_M c)$$

We would now like to capture a notion of computation. A simple notion is to start the machine with any tape configuration, wait until the machine stops and look at the content on the tape afterwards.

Definition 1.18 (Macrostep on tape configurations)**.** Let $M = (Q, \Sigma, tr, q_0)$ be a Turing Machine, $B = \Sigma \cup \{\square\}$ and $in, out \in \mathbb{S}_B$. We define the binary relation macrostep $\to_M \subseteq \mathbb{S}_B \times \mathbb{S}_B$ by, for some $q \in Q$, $p < \omega$,

$$\begin{aligned} in \to_M out \iff & \text{moves}_M(in) = \alpha + 1 < \omega \quad \wedge \\ & \text{seq}_M(in)(\alpha) = (q, out, p) \end{aligned}$$

The following lemma tells us that from pure content only pure content is reachable.

Lemma 1.19. *Let $M = (Q, \Sigma, tr, q_0)$ be a Turing Machine, $B = \Sigma \cup \{\square\}$ and $in, out \in \mathbb{S}_B$. The input convention determines that $\to_M \subseteq \mathbb{P}_B \times \mathbb{P}_B$.*

Proof. Suppose that $in \to_M out$ and (by the input convention) that $in \in \mathbb{P}_B$. Then by Definition 1.18, $\text{seq}_M(in)(\alpha) = (q \in Q, out, p \in \omega)$ for some $\alpha < \omega$. Then $out \in \mathbb{P}_B$ follows from the stronger claim that

$$(\forall\, \beta < \omega)\,(\text{seq}_M(in)(\beta) = (q \in Q, t \in \mathbb{P}_B, p \leq size(t)))$$

We prove this by induction on β.
Suppose $\beta = 0$. Then $\text{seq}_M(in)(0) = (q_0, in \in \mathbb{P}_B, 0 \leq \text{size}(in))$.
Suppose $\beta = \gamma + 1$. Then by Definition 1.15 $\text{seq}_M(in)(\gamma) \hookrightarrow_M \text{seq}_M(in)(\beta)$ and by the induction hypothesis and Definition 1.12:

$$\begin{aligned} \text{seq}_M(in)(\gamma) &= (q' \in Q, t' \in \mathbb{P}_B, p' \leq \text{size}(t')) \\ \text{seq}_M(in)(\beta) &= (q \in Q, t, p) \\ t' &= t\,[p \mapsto x] \end{aligned}$$

First of all, since $p' - p \in \{-1, 0, 1\}$ we have $p' \leq p + 1$.

- Suppose $p' < \text{size}(t')$, then since $t' \in \mathbb{P}_B$

  $$\begin{aligned} t = &\{(n, t'(n)) \mid 0 \leq n < p\} \cup \{(p, x \in \Sigma)\} \cup \\ &\{(n, t'(n)) \mid p < n < \text{size}(t')\} \cup \{(n, \square) \mid \text{size}(t') \leq n < \omega\} \end{aligned}$$

  From which we may conclude that $\text{size}(t) = \text{size}(t')$ and for any $n < \text{size}(t)$ either $t(n) = t'(n) \in \Sigma$ or $t(n) = x \in \Sigma$. Then $t \in \mathbb{P}_B$. Furthermore, $p \leq p' + 1 \leq \text{size}(t)$.

- Suppose $p' = \text{size}(t')$, then since $t' \in \mathbb{P}_B$

$$t = \{(n, t'(n)) \mid 0 \leq n < \text{size}(t')\} \cup \{(p, x \in \Sigma)\} \cup \{(n, \square) \mid \text{size}(t') \leq n < \omega\}$$

From which we may conclude that $\text{size}(t) = \text{size}(t') + 1$ and for any $n < \text{size}(t)$ either $t(n) = t'(n) \in \Sigma$ or $t(n) = x \in \Sigma$. Then $t \in \mathbb{P}_B$. Furthermore, $p \leq p' + 1 = \text{size}(t') + 1 = \text{size}(t)$.

□

By Lemma 1.9 and Lemma 1.19 we can view macrosteps as a relation of words over an alphabet. We will overload the notation of macrosteps.

Definition 1.20 (Macrostep on words)**.** Let $M = (Q, \Sigma, tr, q_0)$ be a Turing Machine. We define macrosteps as a relation of words over Σ by extending words to tape configuration by adding blanks (see Lemma 1.9):

$$t \in \Sigma^* \to_M t' \in \Sigma^* \iff ext(t) \to_M ext(t')$$

So if we stick to the input convention, we can describe distinct tape configurations with words over the input alphabet. That means that we if take a computation (macrostep) from start to finish, we need not concern ourselves with blanks on the tape. We can consider blanks to be part of the black-box functioning of the Turing Machine.
If we allow input from $\mathbb{F}_B$ (finite input with intermittent blanks) or if the machine can write blank symbols (also called 'erasure' in the literature) we are in trouble. In both cases if $in \to_M out$ then in general $out \in \mathbb{F}_B$. However, we cannot, in general, find the finite content part of a sequence in $\mathbb{F}_B$. To that end we would need more information: we can find the finite part if we can observe how many moves the machine has made since it started computing. (We can use this as an upper bound in our search through the sequence $out \in \mathbb{F}_B$.) We now state our output convention for the record.

Definition 1.21 (Output convention)**.** If a Turing Machine halts, the output consists of the entire content of the tape. No other facts about the machine are observable.

We can now interpret Turing Machines as operations on discrete input.

Definition 1.22 (Functional interpretation)**.** We say that a Turing Machine M computes the function f_M such that for all $in \in \Sigma^*$:

$$f_M(in) = \begin{cases} out & \text{if } \exists\, out \in \Sigma^* \text{ such that } in \to_M out \\ \uparrow & \text{otherwise} \end{cases}$$

The question arises if this interpretation of Turing Machines completely captures a notion of 'effective computabilty'. Unfortunately, we cannot claim that every effectively computable function $f : \Sigma^* \to \Sigma^*$ is computable by a Turing Machine with alphabet Σ.

Remark 1.23. Let M be a Turing Machine and $f_M : \Sigma^* \to \Sigma^*$. We define an ordering on Σ^* by $a \leq b \iff dom(a) \leq dom(b)$. This ordering compares the lengths of words. Then for any $\forall a \in \Sigma^*$ we have $f_M(a) \geq a$. This is not necessarily true for an arbritrary effectively computable function over Σ^*, for example consider f such that $f(x) \mapsto \epsilon$ (where ϵ is the empty string).

Nevertheless, we can compute any effectively computable function of words over subsets of the alphabet. Or to put it differently: by adding at least one extra symbol to the alphabet we can compute any effectively computable function of words over the original alphabet. In these terms, we can reformulate the Church-Turing thesis:

Conjecture 1.24. *For every effectively computable function $f : \Sigma^* \to \Sigma^*$ there is an alphabet Γ with $card(\Gamma) > card(\Sigma)$, an (effectively computable) injection $inj : \Sigma^* \to \Gamma^*$ and a Turing-machine $M = (Q, \Gamma, tr, q_0 \in Q)$ such that:*

$$f \simeq inj^{-1} \circ f_M \circ inj$$

We can make the conjecture more specific (and hopefully more intuitive). Conceive of a machine whose alphabet Σ is extended with a new 'erasure' symbol ⊠. Let the input be a word over Σ, but allow the output to contain ⊠ symbols. Then the output could be a word over Σ of arbitrary length, followed by a number of ⊠ symbols. The whole output string is still of greater or equal length as the input string, but the Σ substring may be of any (possibly smaller) length. The injection from the previous conjecture is the identity function, its partial inverse is defined as erasing an arbitrary tail of ⊠ symbols.

Conjecture 1.25. *For every effectively computable function $f : \Sigma^* \to \Sigma^*$ there is a Turing machine $M = (Q, \Sigma \cup \{⊠\}, tr, q_0 \in Q)$ (and an effectively computable function erase) such that:*

$$f \simeq erase \circ f_M$$

where

$$erase(t, n) = \begin{cases} erase(t\,[n \mapsto \square], n-1) & \text{if } t(n) = ⊠ \wedge n > 0 \\ t\,[n \mapsto \square] & \text{if } t(n) = ⊠ \wedge n = 0 \\ t & \text{otherwise} \end{cases}$$

$$erase(t) = erase(t, dom(t) - 1)$$

Conjectures 1.24 and 1.25 could be viewed as a definition of 'effective computability' of functions of words. This view of 'effective computability' deviates from the received view: effective computability is a notion about functions on the natural numbers [1, 2, 3, 6, 12]. By coding the natural numbers as words, it can be hypothesized that Turing Machines completely capture the notion of 'effective computability'. In Chapter 3 we will treat the subject of 'effective computability' for functions over domains other the natural numbers.

1.2.3 Turing Machine Equivalences

We say that two Turing Machines M and N are functionally equivalent if they compute the same function, i.e.:

$$M \equiv_{\text{func}} N \quad \Longleftrightarrow \quad f_M \simeq f_N \tag{1.10}$$

In comparing Turing Machines we only look at what we can observe according to our conventions: input and output.[11] Of course two functionally equivalent machines need not be equal. Thus we are free to define other means of comparison. To do so we have to be explicit about the convention of what is observable, and what is in the black box that is a Turing Machine.
For example, we might state that, in addition to input and output, if a machine stops for some input, it is observable *how many* moves were made since starting. Thus we could compare $\text{moves}_M(in)$ and $\text{moves}_{M'}(in)$ and create smaller equivalence classes. And since $\text{moves}_M(in)$ also depends on in it allows us to compare different inputs.

What we cannot do, according to the conventions, is have a peek at the content of the tape while the machine has not yet halted. To allow this, we can define a tape-content-convention. We can get the tape content after each move from the Turing Machine configuration at that point.

Definition 1.26 (Tape content convention)**.** Let M be a Turing Machine and $in \in \Sigma^*$. The tape content of M started with in is observable as the sequence of words $\text{tseq}_M(in) : \alpha \to \Sigma_M^*$ such that:

$$\text{tseq}_M(in)(\beta) = t \iff \text{seq}_M(in)(\beta) = (q, t, p) \qquad \text{(for some } q \in Q, p \in \omega)$$

Then $\text{tseq}_M(in)(\beta)$ is the content of the tape after β many moves. We analogously define $\text{sseq}_M(in)$ as the unique move-connected sequence of Turing Machine states and $\text{pseq}_M(in)$ as the tape head position sequence.

[11] By observing output we also observe that a machine halts; we cannot observe that a machine does not halt.

Using this convention we can look more closely at Turing Machines.

Definition 1.27 (Tape equivalence)**.** Let M, N be Turing Machines. Then M and N are tape-equivalent, written as $M \equiv_{\text{tape}} N$ if:

$$(\forall\ in \in \Sigma^*)\,(\text{tseq}_M(in) = \text{tseq}_N(in))$$

Some of the moves connecting the configurations could be viewed as 'just movement of the tape head'. In our formalisation we cannot distinguish between writing the same symbol to the tape or leaving the cell untouched. There are common formalisations of Turing Machines that do distinguish these, with for example $tr : Q \times \Sigma \to Q \times (\Sigma \cup \{-1, 1\})$. The drawback here is that a machine cannot write and move the tape head at the same time.

In any case, we like to be able to 'cut out' the moves where the tape content does not change. By the tape content convention we already have tape sequences available. We do not need another convention.

Definition 1.28 (Tape change sequence)**.** Let M be a Turing Machine and $in \in \Sigma_M^*$. We define a tape change sequences $\text{tcseq}_M(in)$ as:

$$\begin{aligned}
\text{tcseq}_M(in) :\ & \alpha \to \Sigma_M^* \\
\text{tcseq}_M(in)(\beta) = {} & \text{tcseq}_M(in)(g_{M,in}(\beta)) \\
& \text{where} \\
& g_{M,in}(0) = 0 \\
g_{M,in}(x+1) = {} & \begin{cases} g_{M,in}(x) & \text{if } \text{tseq}_M(in)(x+1) = \text{tseq}_M(in)(g_{M,in}(x)) \\ g(x)+1 & \text{otherwise} \end{cases}
\end{aligned}$$

Now we get our third notion of equivalence.

Definition 1.29 (Tape change equivalence)**.** Let M, N be Turing Machines. Then M and N are tape change equivalent, written as $M \equiv_{\text{change}} N$ if:

$$(\forall\ in \in \Sigma^*)\,(\text{tcseq}_M(in) = \text{tcseq}_N(in))$$

We can now express the fact that the (input and output) convention do matter.

Claim 1.30. *These three equivalence notion are strict refinements in this order:*

$$\equiv_{tape} \subset \equiv_{change} \subset \equiv_{func}$$

Remark 1.31. As an example of the importance of the output conventions, we would like to quote from [19, Section 2.1]. According to van Emde Boas:

> If one wants the model to produce output, one normally adopts an ad hoc convention for extracting the output from the final configuration. (For example, the output may consist of all nonblank tape symbols written to the left of the head in the final configuration).

Indeed, the suggested convention allows us to determine a finite output string; in order to do so the tape head position must be observable. While it is perfectly fine to adopt an ad hoc output convention, it would not be fine to omit the convention. The reader may verify that, even for machines that start with input in $\mathbb{P}_B$ and that do not write blanks, the functional equivalence derived from this output convention is incomparable to the functional equivalence defined in this section.

1.2.4 Turing Machines with Multiple Tapes

We could conceive of a Turing Machine as having multiple tapes instead of just one. We would like to extend our framework of the previous sections in such a way that the old definitions are a special case. We define an n-tape Turing Machine as a Turing Machine $M = (Q, \Sigma, tr, q_0)$ with the transition function

$$tr : Q \times (\Sigma \cup \{\square\})^n \to Q \times \Sigma^n \times \{-1, 0, 1\}^n$$

A configuration is now a triple (q, t, p) such that $q \in Q$, $t \in (\mathbb{S}_B)^n$ and $p \in \omega^n$. A move is $(q, t, p) \hookrightarrow (q', t', p')$ if $(q, from, q', to, pos) \in tr$ such that:

$$\begin{aligned} from &= (t_0(p_0), \ldots, t_{n-1}(p_{n-1})) \\ to &= \left(t'_0(p_0), \ldots, t'_{n-1}(p_{n-1})\right) \\ pos &= \left((p'_0 - p_0), \ldots, (p'_{n-1} - p_{n-1})\right) \\ t'_i &= t_i\,[p_i \mapsto t'_i(p_i)] \quad \text{(for all } i < n\text{)} \end{aligned}$$

If we extend the input convention to multiple tapes (the initial configuration is $(q_0, in, \mathbf{0})$ such that $in \in (\mathbb{P}_B)^n$ and $\mathbf{0}$ is an n-tuple with all elements zero) then the proof of Lemma 1.19 naturally extends to multiple tapes. That means that with multiple tapes, we can represent the content of each tape with a word over the alphabet. The natural extension of the meaning of Turing Machines follows.

Definition 1.32 (Functional interpretation)**.** Let $n < \omega$. We say that an n-tape Turing Machine M computes the function $f_M : (\Sigma^*)^n \to (\Sigma^*)^n$ such that, for all $in \in (\Sigma^*)^n$:

$$f_M(in) = \begin{cases} out & \text{if } \exists\, out \in (\Sigma^*)^n \text{ such that } in \to_M out \\ \uparrow & \text{otherwise} \end{cases}$$

We can choose to give an n-tape Turing Machine other meaning. For instance, we can only use the first tape for input and output and use the other tapes as 'working memory'. This can expressed as $(f(in)\downarrow \wedge f(in) = out) \iff (in, \epsilon, \ldots, \epsilon) \to_M (out, w_2, \ldots, w_{n-1})$ (where ϵ is the empty string and $\epsilon, w_2, \ldots, w_{n-1}, in, out \in \Sigma^*$). We can think of other variations, for instance designating some tapes for input, other for output, etc. In each case we have to be specific about our definitions and conventions.

As proved in [9, Theorem 2.1], multiple tape Turing Machines (with the first tape as input/output tape) are equivalent in the sense that for any n-tape Turing Machine M there is a single tape Turing Machine M' such that $f_M \simeq f_{M'}$. We have no reason to doubt that this extends to multiple tape Turing Machines as defined in this section. The proof is beyond the scope of this thesis, however.

1.3 Persistent Turing Machines

There is an ongoing debate about the status of Turing Machine computations [26, 36, 38]. Do Turing Machines *capture* all modern notions of computation? Are there notions of computation that are more expressive than 'computing functions' and yet realistic?

We will now explore 'Persistent Turing Machines' (PTMs), which are due to Goldin et al. [26]. Our specific interest in PTMs stems from [33], wherein Hao, Yin and Zhang model computer viruses on (Universal) PTMs. We will discuss to what extent (Universal) PTMs model 'computers' in Section 2.1; we discuss whether they are suitable to define computer viruses in Section 2.4.3. Universal PTMs are introduced in Section 1.5.4. In this section we will introduce the reader to the basic definition of PTMs.

Definition 1.33 (Persistent Turing Machine)**.** A Persistent Turing Machine (PTM) is a 3-tape Turing Machine (3TM) as defined in Section 1.2.4, where each tape has special meaning:

- The first tape contains the input. In principle, the machine will not write on this tape (this restricts the transition function).
- The second tape is the work tape. The content of this tape is persistent; this is the distinguishing feature of PTMs.
- The content of the third tape after computation has halted, is designated as 'output'.

We intend to repeatedly feed the PTM with new input and wait for the output. This process of feeding is meant to be sequential - and the machine may 'remember' what input was fed to it. When we say that the content of the work tape is persistent, we mean that in the sequential process of feeding inputs to the machine we leave the work tape undisturbed.
As correctly noted in [36, Section 5.2], a 3-tape Turing Machine can be simulated by a normal (modern 1-tape) Turing Machine. A function computed by a 3-tape machine can be computed by some 1-tape machine. The observation that the authors fail to make, is that if we give new meaning to what the machine does, this new meaning might not be equivalent to the old. Never mind that we use three tapes for a PTM; this just eases the definitions (and our intuitions).

We can formally define PTMs on the basis 3TMs as defined in Section 1.2.4. This is a slight deviation from [26] as our definitions are more restrictive in the use of blanks. (The input and output is restricted to pure content).

Definition 1.34 (PTM Macrosteps [26])**.** Let M be a 3-tape Turing Machine with alphabet Σ. We define PTM macrosteps $\underset{M}{\longrightarrow}$ on top of macrosteps of a 3-tape Turing Machine (see Section 1.2.4). Let $i, w, w', o \in \mathbb{P}_{\Sigma\cup\{\square\}}$ and ϵ be a tape containing only blanks. Then

$$w \xrightarrow[M]{i/o} w' \iff (i, w, \epsilon) \to_M (i, w', o)$$

Macrosteps only express the fact that we can feed the PTM a single input. To capture the sequential process that the PTM is meant to carry out, we define PTM sequences. These sequences should capture three notions. First that the input is unknown beforehand (we remain free to choose it). Second, the work tape content is determined by the machine, and is used in the next step. Last, the output is determined by the machine and a given input.

Definition 1.35 (PTM sequences)**.** A PTM sequence for machine M is a sequence $p : \alpha \to (\mathbb{P}_{\Sigma^*\cup\{\square\}})^3$ such that $p(0) = (i, \epsilon, o)$ and for all $n < \alpha$ if $p(n) = (i, w, o)$ and $p(n+1) = (i', w', o')$ then

$$(i, w, \epsilon) \to_M (i, w', o')$$

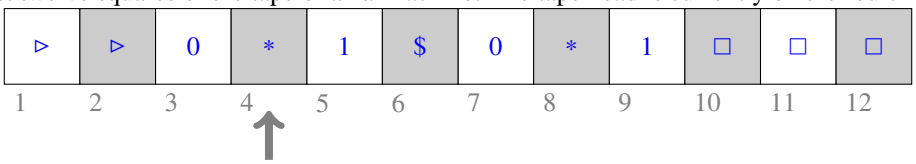
Goldin et al. define a 'persistent stream language' (PSL) as a sequence of (i, o) pairs, on top of PTM sequences. This leads them to the notion of PSL-equivalence: two machines are equivalent if their language is the same. As PSL-equivalence is strictly larger than functional equivalence (of the underlying 3TMs), Persistent Turing Machines (with the notion of PSL) seem to capture a more fine-grained notion of computation. Our intuitions (and hopefully the reader's intuition) supports this claim: a PTM can deal with an infinite stream of inputs that are unknown beforehand At the same time its 'memory' (the work tape) is persistent. It is obvious that such a machine could learn and interact with its environment. Such things are far from obvious for modern Turing Machines with a functional interpretation. For an in-depth discussion of PSL, see [26].

1.4 Turing's Original Machines

In this section we will discuss the machines that were introduced by Turing's original article [28]. As we will see these machines are rather different from the modern Turing Machines we have discussed in the previous sections. According to the historical research by Copeland in [13] the best name for these machines is 'logical computing machines'. Since this name is not well known, we will mostly refer to his machines as 'Turing's original machines' or, in this section, just as 'machines'. Figure 1.3 shows an example of such a machine.

We will support our analysis of Turing's article by quoting extensively from [28], mostly in the footnotes. Unless otherwise specified, quotes are from [28]. First of all, we notice that Turing's article does not clearly distinguish between prescribed properties of machines and properties that are just convenient. For now, assume that all properties are prescribed. We will investigate this issue in the next section (Section 1.4.1).

Figure 1.3 – Turing's original machine
The first twelve squares of the tape of an *a*-machine. The tape head is currently on the fourth square.



Definition 1.36 (machine)**.** A 'machine' has the following properties

- A machine has finitely many states (or '*m*-configurations').[12]
- A machine has a tape divided into squares that can contain one symbol at a time.[13]
- The machine has a 'tape head' that reads one symbol at a time, called the 'scanned symbol'.[14]
- The machine makes 'moves': go to a new state and erase the symbol, write the symbol or move one square to the left or right.[15]
- The possible moves of the machine at any stage are fully determined by the 'scanned symbol' and the *m*-configuration. This pair is called the 'configuration'.[16]

Definition 1.37 (*a*-machine)**.** If at all stages of movement, the machine has at most one possible move, we call it an *a*-machine ('automatic machine').[17]

Remark 1.38 (*c*-machine). If at any stage the machine has at more than one possible move, Turing calls it a *c*-machine ('choice machine'). This choice is determined by something external to the machine.[18] Such a machine is closely related to what is commonly known as a 'non-deterministic Turing Machine'; whereas *a*-machines can be thought of as 'deterministic Turing Machines'.

Turing distinguishes two kind of symbols: symbols 'of the first kind' (called figures) and symbols 'of the second kind'. We shall call the second kind 'auxiliaries'. The auxiliaries are "rough notes to assist the memory" and are "liable to erasure". Figures, once printed, are not liable to erasure.[19]

Definition 1.39 (Computing machine)**.** A 'computing machine' is an *a*-machine that only prints the figures '0' or '1' (and possibly some auxiliaries).

Definition 1.40 (Computed sequence)**.** We say that a machine is 'properly started' if it is set in motion from an initial state, supplied with a blank tape. The subsequence of figures on the tape of an *a*-machine that was properly started, is called the 'sequence computed by a machine'.

[12] [28, Section 1]: "We may compare a man in the process of computing a real number to a machine which is only capable of a finite number of conditions $q_1, q_2, \ldots, q_R$ which will be called '*m*-configurations'."

[13] [28, Section 1]: "The machine is supplied with a 'tape' [...] running through it, and divided into sections (called 'squares') each capable of bearing a 'symbol'."

[14] [28, Section 1]:"The symbol on the scanned square may be called the 'scanned symbol'. The 'scanned symbol' is the only one of which the machine is, so to speak, 'directly aware'."

[15] [28, Section 1]: "In some of the configurations in which the scanned square is blank (i.e. bears no symbol) the machine writes down a new symbol on the scanned square: in other configurations it erases the scanned symbol. The machine may also change the square which is being scanned, but only by shifting it one place to right of left. In addition to any of these operations the *m*-configuration may be changed."

[16] [28, Section 1]: "The possible behaviour of the machine at any moment is determined by the *m*-configuration q_n and the scanned symbol..."

[17] [28, Section 2]: "If at any stage the motion of a machine [...] is *completely* determined by the configuration, we shall call the machine an 'automatic machine' (or *a*-machine)."

[18] [28, Section 2]: "For some purposes we might use machines (choice machines of *c*-machines) whose motion is only partially determined by the configuration.[...].When such a machine reaches one of these ambiguous configurations, it cannot go on until some arbitrary choice has been made by an external operator."

[19] [28, Section 1]: "It will only be these rough notes which will be liable to erasure".

The definition of a 'computed sequence' is somewhat unsatisfactory. First of all, it defines a start of moves, but no end. Implicitly, we have to look at the subsequence of figures *in the limit of the moves*, i.e. 'after infinitely many moves'. The question arises if such a subsequence exists (in the limit). Turing remedies this issue with the following (unnamed) convention.

Definition 1.41 (Alternation convention)**.** Uneven squares are called F-squares and even squares E-squares. A machine shall only write figures on F-squares. "The symbols on F-squares form a continuous sequence".[20] We call this sequence the F-sequence. Only "symbols on E-squares will be liable to erasure".

The intuition behind this convention is that each square with 'real' content (F-square) is accompanied by an auxiliary square (the E-square to it's immediate right). The auxiliary squares are just used to make notes to aid the computation.

The alternation convention gives rise to two kind of machines. One: machines whose F-sequence is finite, called 'circular'. Two: machines whose F-sequence is infinite, called 'circle-free'.[21] According to Turing, a machine may be circular because "it reaches a configuration from which there is no possible move, or if it goes on moving and possibly printing [auxiliaries], but cannot print any more [figures]". Here "configuration" can mean any of three things:

- An 'm-configuration': the 'state' of the machine.
- A 'configuration': the 'state' of the machine combined with the scanned symbol.
- A 'complete configuration', consisting of "the number of the scanned square, the complete sequence of all symbols on the tape, and the m-configuration".

Considering 'configurations', we can conclude that Turing allows for machines whose moves are not determined for *all* m-configurations. In terms of a 'transition function': the function may be partial. Considering 'complete configurations', we can conclude that Turing, in principle, does not disallow the tape head 'falling off the left end of the tape': if the only possible move on the first square would go to the left the machine breaks down.

The issue of falling off the left side is addressed by Turing in his second example [28, Section 3]. Although he never explicitly formulates the following convention, his examples stick to it.

Definition 1.42 (Left-hand convention)**.** The first moves of a machine print ⊳⊳ on the tape.[22] A machine never goes left if the scanned symbol is the leftmost ⊳. A machine never changes or erases a ⊳.

Turing does not explicitly disallow machines to print auxiliaries on F-squares. Once an auxiliary symbol is printed on an F-square, however, it cannot be erased. (This partially supports the left-hand convention.) Notice that the *computed sequence* and the F-sequence are not exactly the same: the *computed sequence* is the figures-subsequence of the F-sequence. Either sequence can only become longer. Therefore, if a machine is circle-free and obeys the alternation convention, then the computed sequence is guaranteed to exist in the limit of the moves.

1.4.1 Turing's Conditions

Summarizing the previous section, Turing's a-machine is subject to the following conditions.

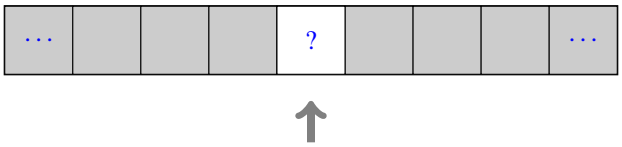
1. Figures can only be written on F-squares.
2. F squares are 'write-once' (not "liable to erasure").
3. The F-squares form a continuous sequence.

[20] Where 'continuous' means: "There are no blanks until the end is reached".

[21] [28, Section 2]: "If a computing machines never writes down more than a finite number of symbols of the first kind, it will be called 'circular'. Otherwise it is said to be 'circle-free' ."

[22] Turing uses the symbol 'ɔ'. His examples consistently use two such symbols at the start of the tape, even though one would suffice. We speculate that the reason for this choice is the fact that the second left-hand symbol is on the E-square that accompanies the first F-square. The first F-square also contains a left-hand symbol, so there is no apparent reason why the machine should need to write notes on the accompanying E-square.

Figure 1.4 – The move of a machine is only determined by the current symbol and m-configuration.



4. The first F-square and first E-square contain an '$\triangleright$' symbol that signals the left end of the tape. The $\triangleright$ symbol is also 'write once'. The machine should not move to the left when it reads the leftmost $\triangleright$ symbol.

In the definition of Turing's machine, we stated that the possible moves of a machine are determined by the configuration (scanned symbol and m-configuration). A move is now (re)defined (in accordance with [28, Section 5]) as consisting of three things: an m-configuration (state) the machine transitions to, a symbol that is to be printed on the current scanned square, and a direction for the tape head to move to. Erasure is expressed as writing a 'blank' symbol ($\square$). We can think of 'the moves of an a-machine being determined' as a transition function.

Definition 1.43 (Transition function)**.** The moves of an a-machine are determined by a 'transition function' tr. Let Q be the set of m-configurations (states), $\mathbb{F}$ the set of figures, $\mathbb{A}$ the set of auxiliaries and L, N, R be the possible movements (left, no movement, right). Then tr is defined as:

$$tr : Q \times (\mathbb{F} \cup \mathbb{A}) \to Q \times (\mathbb{F} \cup \mathbb{A}) \times \{L, N, R\}$$

When we write down the function as a list of quintuples, we call it a 'program'.

Can we determine if an arbitrary function is a 'valid' program, i.e. satisfies Turing's conditions? If we 'run' the machine (defined by the program) and it violates the conditions, we know that the function is not a valid program. But how can we be sure that it will *never* violate a condition? We would like to express Turing's conditions as a property of the transition function, without having to run the machine. We shall now show, however, that this is not possible.

Let us regard the transition function (program) decoupled from the tape. It its most unrestricted form we have a transition function $tr : Q \times \Sigma \to Q \times \Sigma \times \{L, N, R\}$ (directly mirroring Turing's quintuples). We assume without loss of generality that tr is a total function (for each partial function there is an equivalent total one w.r.t. computable sequences). Thus the result of the transition function is fully determined by the input: the current state and the symbol on the scanned square. Notably, we do not give the position of the tape head as input to the function. From the point of view of the transition function, the tape extends arbitrarily many squares *to both sides*. This situation is depicted in Figure 1.4.

In order to satisfy Turing's conditions, we have to distinguish at least six (two times three) situations:

- Is the tape head positioned on a F-square or E-square?
- Are we in one of three positions:
  - On the leftmost blank F-square? The machine is allowed to write on the F-square.
  - To the left of the leftmost blank F-square? The machine is not allowed to write on any F-square until we move enough squares to the right to reach the leftmost blank F-square.
  - To the right of the leftmost blank F-square? The machine is not allowed to write on any F-square until we move enough squares to the left to reach the leftmost blank F-square.

Does the internal state carry this information? There is no intrinsic reason why it should. We could, however, transform the program to be defined on a new set of states Q' that contains six states for every state in Q. For example, let $q \in Q$. Then let Q' contain $q_{F,l}, q_{F,b}, q_{F,r}$ signifying being on a F-square satisfying

Figure 1.5 – Blank tape with left hand markers and distinct F and E markers.

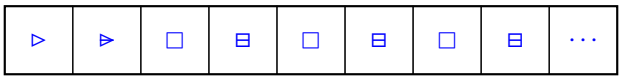
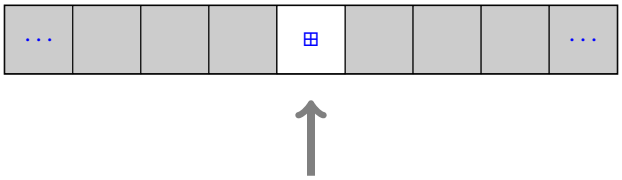


Figure 1.6 – A machine with a distinct 'left most blank symbol'.



the conditions "left of", "on the leftmost blank" and "right of"; similarly, let $q_{E,l}$, $q_{E,b}$, $q_{E,r} \in Q'$ be the three states that signify being on an E-square.

We should then add the appropriate transitions to keep track of the changes of situation; for example if the tape head moves to the right $q_{F,r}$ should transition to $q_{E,r}$. So every move causes a change in state since the tape head moves from an E-square to an F-square or vice versa. Only now can we check if the transformed function satisfies the conditions. We can guarantee a correct F-sequence if the transition function is the identity function w.r.t. the scanned symbol, if the machine is in the states qF, l and qF, r.

You may have noticed that we could do with less states if we also look at the information carried by the current symbol. For example, we could do with 2 x 2 states by purging the 2 states for "On the leftmost blank F-square". It is enough to know that we are "not to the right of the leftmost blank F-square" *and* the current square is blank. In any case, for an arbitrary program, the states do not carry all this information.

Do the symbols on the tape carry the needed information? Let us denote the set of symbols that may appear on the F-squares and E-squares by $\mathbb{A}_F \supset \mathbb{F}$ and $\mathbb{A}_E \supset \mathbb{A}$ respectively. For any auxiliary that we wish to write on an F-square (notably $\triangleright$) we can introduce a new symbol that we only use for E-squares, and use the original for F-squares. In essence, we may assume that $\mathbb{A}_E$ and $\mathbb{A}_F$ are disjoint. Let us at least distinguish between F-blanks ($\square$) and E-blanks ($\boxminus$) (similarly for left-hand markers: $\triangleright$ and $\blacktriangleright$ respectively). So $\mathbb{A}_F \supseteq \mathbb{F} \cup \{\square, \triangleright\}$ and $\mathbb{A}_E = \mathbb{A} \cup \{\boxminus, \blacktriangleright\}$. Figure 1.5 exemplifies this idea.

Now if we encounter an F-blank we still have to distinguish between the left-most F-blank and other F-blanks. Suppose we can and that the left-most F-blank is marked by a new symbol $\boxplus$. This symbol can appear on the tape only once. So if we are in the situation depicted by Figure 1.6, we have sufficient information to know that the machine may write an output symbol. This leaves us with one problem: once the machine writes over the symbol $\boxplus$, we have no $\boxplus$ left on the tape, so we do not have a marker for the left most blank symbol on the tape. In fact, we cannot place the marker again because we have lost track of its position.

Wrapping up: by making the symbols for F-squares distinct from those on E-squares we have a partial solution. Since the states do not necessarily carry any information, we can not express Turing's conditions as properties of the configuration, and therefore of transition functions.

We envisage several solutions.

1. We can provide the transition function with more information.
   If we let the sets of symbols for F- and E-squares be disjoint, we only have to provide the function with exactly enough information: *true* for being on the left-most blank F-square and $false$ otherwise. This seems to defeat the purpose of the separation of program and tape: we can only provide the value of this Boolean by keeping track of the 'complete configuration' (including the whole tape content and tape head position). Nevertheless, let:

   $$tr : Q \times (\mathbb{A}_F \cup \mathbb{A}_E) \times \{true, false\} \rightarrow \ Q \times (\mathbb{A}_F \cup \mathbb{A}_E) \times \{L, N, R\}$$

   Then the following restrictions ensure that such a machine adheres to Turing's conditions (if the

Boolean is correct):

$$tr(q_i, \square, false) = \left(q_j, \square, m\right)$$
$$tr(q_i, s, b) = \left(q_j, s', m\right) \text{ such that either } s, s' \in \mathbb{A}_F \text{ or } s, s' \in \mathbb{A}_E$$
$$tr(q_i, s, b) = \left(q_j, s, m\right) \text{ if } s \in \{\triangleright, \blacktriangleright\}$$
$$tr(q_i, \triangleright, b) = \left(q_j, \triangleright, m\right), \text{ such that } m \neq L$$
$$tr(q_i, s, b) = \left(q_j, s, m\right) \text{ if } s \in \mathbb{A}_F$$

2. We can restrict the movement of the tape head on blank F-squares so that a machine can only move to the right if it writes a non-blank symbol. Let:

   $$tr : Q \times (\mathbb{A}_F \cup \mathbb{A}_E) \rightarrow Q \times (\mathbb{A}_F \cup \mathbb{A}_E) \times \{L, N, R\}$$

   Then the following restrictions ensures that such a machine adheres to Turing's conditions:

   $$tr(q_i, \square) = \left(q_j, s, m\right) \text{such that either } m \neq R \text{ or } (m = R \wedge s \neq \square)$$
   $$tr(q_i, s) = \left(q_j, s', m\right) \text{ such that either } s, s' \in \mathbb{A}_F \text{ or } s, s' \in \mathbb{A}_E$$
   $$tr(q_i, \triangleright) = \left(q_j, \triangleright, m\right), \text{ such that } m \neq L$$
   $$tr(q_i, s) = \left(q_j, s, m\right) \text{ if } s \in \{\blacktriangleright\} \cup \mathbb{A}_F$$

   This approach has the drawback that we cannot write on an E-square without having written to all F-squares to its left. It is unclear if machines with this restriction are as expressive as Turing's original machines.

We propose a third alternative in the next section.

1.4.2 Relaxing Turing's conditions

We shall drop all of Turing's conditions but one. We still enforce the condition that the F-squares are write-once. Now the question arises if the class of such machines also characterises the set of computable sequences. Clearly, the set of all such machines is a superset of the set of machines *with* Turing's conditions, so the corresponding set of computable sequences contains at least the set of computable sequences according to Turing. We need only be concerned if such machines do not compute more sequences. That all depends on the interpretation of machines. Turing only gives meaning to 'circle-free' machines, the meaning of 'circular' machines is undefined.

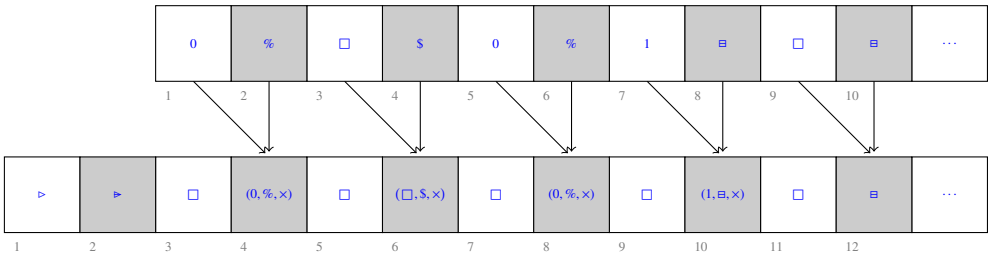
Definition 1.44 (Computable sequence)**.** "A sequence is said to be computable if it is computed by a circle-free machine." [28, Section 2]

By prescribing that F-squares are write-once, we ensure that the F-sequence always has a continuous (though possibly finite) initial segment, even in the limit of infinitely many moves. We will use that property later on. If, in the limit, there are still intermittent blanks in the F-sequence (i.e. the initial continuous segment is not infinite), the machine does not compute a sequence according to Definition 1.44. We need not, therefore, be concerned about such machines. What if, in the limit, the machine does produce a proper continuous, infinite F-sequence? We have to prove that we can find an equivalent machine that adheres to Turing's conditions and produces the same computed sequence.

Suppose we have a machine M with the relaxed condition. We will create a new machine from M. The first step is to translate to a machine M' that does all the computation on the E-squares and leaves the F-squares intact. We have to explode the set of symbols. We define a set of symbols $\mathbb{A}_T = (\mathbb{A}_F \times \mathbb{A}_E \times \mathbb{A}_M) \cup \mathbb{F} \cup \{\square, \boxminus, \triangleright, \blacktriangleright\}$ where $\mathbb{A}_M = \{*, \times\}$.[23] The set $\mathbb{A}_M$ is used later on as a Boolean marker.

[23] The set $\mathbb{A}_T$ is not a 'flat' set of symbol; we could however choose a real set of symbols $\mathbb{A}'_T$ that is in bijection with $\mathbb{A}_T$. For notational convenience we will stick with $\mathbb{A}_T$.

Figure 1.7 – Translation 1 & 2. Two squares are mapped to a single E-square.



We include $\mathbb{F}$ so we can still write figures on the F-squares if we want to. We include triples in $\mathbb{A}_F \times \mathbb{A}_E \times \mathbb{A}_M$ to be able to simultaneously write the content of one F-square and one E-square and one marker. For the full mapping, see Table 1.1; a graphical representation can be found in Figure 1.7.

Then we create M'' from M' with a second translation. This translation has the purpose of accomplishing three things:

- Start by writing '$\triangleright \triangleright \square$' on the tape.
- Mark the right-most E-square that the machine has visited as $(\_, \_, *)$ and other visited E-squares as $(\_, \_, \times)$.
- Search through the visited E-squares if it is the leftmost E-square containing a non blank F-square symbol such that the F-square to its left is still blank and then write this symbol to the F-square to its left.

To accomplish the translation, we will first *replace* quintuples according to Table 1.2. Then we add states and transitions according to Table 1.3.[24] The machine M'' has q_s as new start state. Now since our M has an infinite computed sequence, we know that M'' will infinitely often visit a square that is further to the right than the machine has travelled so far For each such move, M'' 'executes' its 'search$(\cdot)$' subroutine which properly extends the F-sequence of M''. That means that M'' obeys Turing's conditions and that its computed sequence is the same as the computed sequence of M. In other words, the class of machines with relaxed conditions characterises the same class of computable sequences.

1.4.3 Talkative Machines

The proof of equivalence in the previous section reveals an interesting fact: we do not need the F-squares for computation. Any information that the machines needs to remember may be written down on E-squares. In a way, the F-squares of M'' only convey information to an external observer. We can think of other ways to convey this information. For example, let the machine make a distinct sound for any figure that it conveys to the external observer. It is appropriate to think of sound, because sounds have two essential properties: they are observable and they are fleeting. In essence, the machine is 'talking' to you.

When we formalise this concept, we need a 'non sound' just as we use a 'blank symbol' to represent 'no symbol'. Let us suppose that we can choose distinct sounds for the figures and for the 'blank' sound.

Definition 1.45 (Talkative Machine (1)[25])**.** Let $\mathbb{F}$ be a set of figures containing the blank '$\square$', let $\mathbb{A}$ be a set of auxiliaries not containing the blank '$\square$' and $Q = \{q_0, q_1, \ldots\}$ a finite set of states. Then a *talkative machine* is defined by the transition function tr:

$$tr : Q \times (\mathbb{A} \cup \{\square\}) \to Q \times (\mathbb{A} \cup \{\square\}) \times \mathbb{F} \times \{-1, 0, 1\}$$

We may refer to such a machine as a machine $T = (Q, \mathbb{A}, \mathbb{F}, tr, s_0)$ rather than just by tr.

[24] For an explanation of such 'subroutines' (or 'm-functions') refer to [28, Section 7] or the easier [13, Pp. 54 - 65].
[25] The definition of Talkative Machines will be refined in Definition 1.52.

Definition 1.46 (Talkative configurations)**.** For a Talkative Machine T, the set of all 'complete configurations' $\mathbb{C}_T$ is defined as:

$$\mathbb{C}_T = \{(q, tape, out, pos) \mid q \in Q,\ tape : \omega \to (\mathbb{A} \cup \{\square\}),\ out : \alpha \to \mathbb{F},\ pos < \omega, \alpha < \omega\}$$

Definition 1.47 (Talkative moves)**.** For a Talkative Machine T, 'moves' are the binary relation $\hookrightarrow_T$ on $\mathbb{C}_T$ defined below. (Where ';' is sequences concatenation. For notation see Appendix A.)

$$\begin{aligned}(q, t, out, p) \hookrightarrow_T (q', t', out', p') \iff \\ \wedge\ (q, t(p), q', t'(p), sound, p' - p) \in tr \\ \wedge\ t' = t\,[n \mapsto t'(p)] \\ \wedge\ \vee\ (sound = \square \wedge out = out') \\ \vee\ (sound \neq \square \wedge out' = out; sound)\end{aligned}$$

Definition 1.48 (Talkative computed sequence)**.** We say that a Talkative Machine T 'computes a sequence in n moves' if this sequence equals the 'output' sequence after n moves and the machine is started properly. (That is: starting with empty tape, empty output sequence, in start state q_0 and position 0). Formally, we say that it computes the sequence $comp_n(T)$ such that:

$$(q_0, \epsilon, \epsilon, 0) \hookrightarrow_T^n (q_i, t, comp_n(T), p) \qquad \text{(for some } q_i \in Q, t \in \mathbb{S}_{\mathbb{A}\cup\{\square\}} \text{ and } p < \omega)$$

We say that a Talkative Machine 'computes a sequence' if this sequence is the limit of the 'output' sequence for infinitely many moves. Formally, we say that it computes the sequence $comp(T)$ such that

$$(\forall\ n < \omega)\ (\exists\ m < \omega)\ (comp(T) \restriction n = comp_m(T))$$

We say that a Talkative Machine 'outputs' a sequence if it computes it.

Turing interpreted his a-machines as 'computing a sequence'. His notion of 'computed sequence' (Definition 1.40) exactly corresponds with our notion of 'talkative computed sequences'. A benefit of our definition of 'Talkative Machines' is that we can clearly state this interpretation of machines. We can also clearly compare and distinguish machines by their computed sequences, as in the following definition.

Definition 1.49 (Talkative equivalence)**.** We say that Talkative Machines T and T' are 'talkative equivalent' (or: 'sound-equivalent') if and only if they compute the same sequence. We write:

$$T \equiv_{talk} T' \iff comp(T) = comp(T')$$

We may also wonder how the definition of Talkative Machines compares to modern Turing Machines (as in Definition 1.10). The following claim is a direct result of the definitions.

Claim 1.50 (Turing Machine correspondence)**.** *If a Talkative Machine (with function tr) does not write blanks on the tape, then it is an extension of a Turing Machine (as in Definition 1.10). I.e. if*

$$\begin{aligned}&tr : Q \times (\mathbb{A} \cup \{\square\}) \to Q \times \mathbb{A} \times \mathbb{F} \times \{-1, 0, 1\} \\ &tr'(q, s) = (q', s', m) \iff tr(q, s) = (q', s', sound, m)\end{aligned}$$

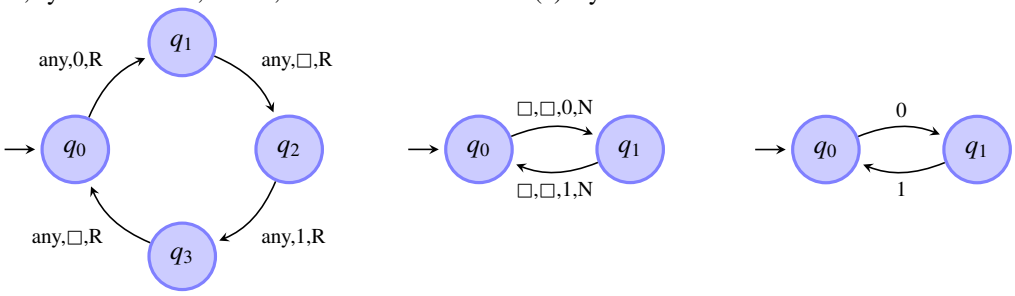
then tr' determines the Turing Machine $(Q, \mathbb{A}, tr', q_0)$.

Lemma 1.51. *For every Talkative Machine T, there is a sound-equivalent machine T' such that T' does not write blanks on the tape.*

Proof. We can construct T' from T by replacing and adding quintuples according to Table 1.4. What we do is to make sure that the T' writes ⊠'s to the tape instead of □'s, and has the same behaviour for ⊠ as for □. As a result of this translation T' does not write □'s, while it is sound-equivalent to T. □

The fact that Talkative Machines that do not write blanks are still general enough to compute any sequence that can be computed by a general Talkative Machine, justifies restricting the definition of Talkative Machines.

Figure 1.8 – Comparison of Turing Machines and automata
Three machines that compute the infinite string $(01)^\infty$. From left to right: Turing's original machine (a), a Talkative Machine (b), and a finite automaton (c). Notice that the Talkative Machine does not move on, or print to the tape. The finite automaton is understood to write on tape while always moving to the right. Transitions of (a) are labeled with the symbol read, symbol written and direction of move. For (b): symbol read, symbol written, sound, direction of move. For (c): symbol written.



Definition 1.52 (Talkative Machine (2))**.** Let $\mathbb{F}$ be a set of figures containing the blank '□', let $\mathbb{A}$ be a set of auxiliaries not containing the blank '□' and $Q = \{q_0, q_1, \ldots\}$ a finite set of states. Then a *talkative machine* is defined by the transition function tr:

$$tr : Q \times (\mathbb{A} \cup \{\square\}) \to Q \times \mathbb{A} \times \mathbb{F} \times \{-1, 0, 1\}$$

With this restriction we can apply any definition and property of Turing Machines (from Section 1.2) to Talkative Machines. Most notably, this gives us a functional interpretation and thus functional equivalence. The definition of Talkative Machines encompasses equivalence of both modern and original Turing Machines. As can be seen in the following lemma, both notions remain independent.

Lemma 1.53. *Functional equivalence and talkative equivalence are incomparable.*

Proof. Suppose we have a Turing Machine $M = (Q, \Sigma, tr, q_0)$. Now let $\mathbb{F} = \{A, B\}$ and let Talkative Machines A and B be defined by tr_A and tr_B given below. Clearly, $M \equiv_{\text{func}} M$ and therefore $A \equiv_{\text{func}} B$. However, $A \not\equiv_{talk} tr$, so we conclude $\equiv_{talk} \not\subseteq \equiv_{\text{func}}$.

$$tr_A(q_i, s) = \left(q_j, s', A, m\right) \iff tr(q_i, s) = \left(q_j, s', m\right)$$
$$tr_B(q_i, s) = \left(q_j, s', B, m\right) \iff tr(q_i, s) = \left(q_j, s', m\right)$$

Now consider the Talkative Machines T and T' determined by functions tr and tr':

$$tr(q_0, \square) = (q_0, \square, 1, N)$$
$$tr'(q_0, \square) = (q_0, \square, 1, N)$$
$$tr'(q_0, a) = (q_0, a, 1, N)$$

Clearly $T \equiv_{talk} T'$, since both computed sequences are '1^∞'. We see that T computes the identity function on any nonempty string, while the partial function computed by T' is undefined for input strings starting with 'a'. Then $T \not\equiv_{func} T'$, and therefore $\equiv_{\text{func}} \not\subseteq \equiv_{talk}$. □

Another interesting property of Talkative Machines is the resemblance with finite state machines *without a tape*. Any sequence that can be 'computed' by a finite state machine, can be computed by a Talkative Machine that does not use the tape.[26] This is exemplified in Figure 1.8.

The next order of business would be to define a 'universal Talkative Machine'. We postpone this to Section 1.5.3.

[26] We can express 'not using the tape' by requiring that $\mathbb{A} = \{\square\}$ and $tr(s, \square) = (s', \square, 0)$. For formal definitions of finite state machines or automata see [2, 4].

Table 1.1 – Translation 1

For each in M $(1 \leq i \leq n)$	For each	Put into M'
$\left(q_i, s, q_j, t, R\right)$	$\_ \in \mathbb{A}_F, m \in \mathbb{A}_M$	$\left(q_i, (s, \_, m), q_{j+n}, (t, \_, m), N\right)$
	$\_ \in \mathbb{A}_E, m \in \mathbb{A}_M$	$\left(q_{i+n}, (\_, s, m), q_{j+2n}, (\_, t, m), R\right)$
$\left(q_i, s, q_j, t, L\right)$	$\_ \in \mathbb{A}_F, m \in \mathbb{A}_M$	$\left(q_i, (s, \_, m), q_{j+3n}, (t, \_, m), L\right)$
	$\_ \in \mathbb{A}_E, m \in \mathbb{A}_M$	$\left(q_{i+n}, (\_, s, m), q_j, (\_, t, m), N\right)$
$\left(q_i, s, q_j, t, N\right)$	$\_ \in \mathbb{A}_F, m \in \mathbb{A}_M$	$\left(q_i, (s, \_, m), q_j, (t, \_, m), N\right)$
	$\_ \in \mathbb{A}_E, m \in \mathbb{A}_M$	$\left(q_{i+n}, (\_, s, m), q_{j+n}, (\_, t, m), N\right)$
	$\_ \in \mathbb{A}_T$	$\left(q_i, \_, q_{j+2n}, \_, R\right)$
	$\_ \in \mathbb{A}_T$	$\left(q_{i+3n}, \_, q_{j+n}, \_, L\right)$

Table 1.2 – Translation 2 (replace quintuples)

For each in M $(1 \leq i \leq n)$	For each	Put into M'
$\left(q_{i+n}, (s, s', *), q_j, (t, t', \_), R\right)$	$\_ \in \mathbb{A}_F$	$\left(q_{i+n}, (s, s', *), search(q_j), (t, t', \times), N\right)$
$\left(q_0, \square, q_j, t, m\right)$		$\left(q_s, \square, \text{start}(q_j, t, m), \square, N\right)$

Table 1.3 – Translation 2 (add states and quintuples)

State	Scanned square	Operations	Next state
start(S,t,m)		P[$\triangleright$],R,P[$\triangleright$],R,R,P[$(t, \boxminus, *)$],m	S
search(S)		R, R, P[$(\square, \square, *)$]	lmb (copy(S), star (S))
lmb (S, S')	$\triangleright$		lmb2 (S, S')
		L	lmb (S, S')
lmb2 (S, S')	$(\_, \_, *)$		S'
	$\square$	R	S
		R	lmb2 (S, S')
copy(S)	$(s \neq \square, \_, \_)$	L, P[s]	star (lmb (copy(S), star (S)))
		N	star (S)
star (S)	$(\_, \_, *)$		S
		R	star (S)

Table 1.4 – Talkative Machine translation (add and replace quintuples)

For each in T	Modification	Put into T'
$\left(q_i, s, q_j, \square, out, m\right)$	replace by	$\left(q_i, s, q_j, \boxtimes, out, m\right)$
$\left(q_i, \square, q_j, s, out, m\right)$	add	$\left(q_i, \boxtimes, q_j, s, out, m\right)$

1.5 Universal Turing Machines

Let us, in a very general sense, state that a 'Universal Turing Machine' is a Turing Machine that can simulate *any* other Turing Machine. Quoting Davis [25]:

> ...existence of such a universal machine is inevitable. This is because it is easy to provide an algorithm that using the table defining any given Turing Machine $\mathcal{M}$ will step-by-step perform exactly like $\mathcal{M}$. Thus, if one believes that any algorithm can be carried out by a Turing Machine, that would have to be the case for this algorithm as well.

If there is confusion about how Turing Machines are defined (see Section 1.1), or should be defined, the definitions of '*universal* Turing Machines' in standard literature are almost mystical. We refer somewhat arbitrarily to the 'standard works' on automata and Turing Machines like [2, 4, 9, 10]. These are the works referenced in the most prominent articles on the modeling of viruses (see bibliography).

In this section we will first review these 'standard works'. In doing so we will try to draw the reader's attention to some particular details. For instance, how do we describe (or 'encode') the simulated machines? And: in what form are these descriptions to be found on the tape of the universal Machine? Such issues will be important for our discussion of computer viruses in the next chapter. In particular, we would like to know (1) if 'programs' can be seen as contiguous sequences and (2) if it matters at what time during the computation we have a look at the content of the tape.

We will then shortly introduce the reader to the universal machine as defined by Turing in [28]. Lastly the section concludes by coining two other universal machines: Universal Talkative Machines in Section 1.5.3 and Universal Persistent Turing Machines in Section 1.5.4.

1.5.1 Universal Turing Machines in Literature

The first definition we review is that of Minsky [2, Section 7.2]. Minsky gives a semi-formal 'definition' of a universal machine U that can simulate the behaviour of any machine T that uses the binary alphabet $\{0, 1\}$. The author's remarks that "these restrictions are inessential". Indeed we can first translate a machines T with another alphabet to a machines T' with the binary alphabet. By simulating T', U in essence simulates T'. There is no way, however in which U can distinguish between the simulation of T and T'. In a strict sense, if the translation is not an integral part of our definition/interpretation of U, then we can not call such a machine 'universal' (since it can only simulate a very specific class of machines). The whole machine U is described in terms of: 'it does such and such and then it does such and such while it remembers something', etc; no explicit or implicit program of U is presented.

Minsky represents an arbitrary large set of states (of T) by binary strings of, presumably, fixed length, in which case the machine is definitely not universal.[27] Though a precise formulation of his machine U might resolve such representational problems, Minsky's informal definition leaves such issues unresolved.

A completely different approach to Universal Turing Machines is presented by Davis [1, Section 4.3] and Cooper [12, Section 4.3]. These authors only prove the existence of a Universal Turing Machine as a result of Kleene's 'normal form theorem'.[28] This leaves us with a Universal Turing Machine U that is essentially a black box. Notably, it is only 'universal' for machines that compute functions on natural numbers, through some encoding of the natural numbers in an alphabet.

As input for the machine we have to write the strings '1^z01^x' to the tape, where z is the 'Gödel number' of a machine Z and x is a 'Gödel list number' of n unary representations of numbers y_1 through y_n (for arbitrary n). If the universal machine halts, it should leave the unary representation of a number on the tape.[29] We say that U is universal because it writes the same number to the tape for input '1^z01^x' as machine Z does for '$1^{y_1}01^{y_2}0\ldots01^{y_n}$'.

We want to impress upon the reader that Davis' Universal Turing Machine is very particular to the format of its input on the tape, while it is completely unspecified how it's computation leads it to write the

[27] As noted in [4, Section 7.8], if we have both a restricted alphabet *and* a restricted state space, we can represent only finitely many Turing Machines.

[28] A very short introduction to this theorem is: [42].

[29] The output representation is slightly more relaxed, as the machine may output intermittent 0 symbols.

appropriate output to the tape.

A third definition of a Universal Turing Machines is due to Papadimitriou [9]. Even though Papadimitriou omits a precise description of the program of his machine, the concept of his machine is very plausible. The encoding of Turing Machines U that Papadimitriou proposes is very precise and general enough to encode any machine M, while using a fixed alphabet for U. Papadimitriou remarks that a Universal Turing Machine can be conceived to have a single tape; nevertheless his machine uses two tapes. One tape holds the (encoded) current state of the simulated machine (M), plus the (encoded) content of the tape of M. The other tape contains the encoded program of M. The distinction between using one or two tapes is an important one. For if we translate Papadimitriou's universal machine to one that uses a single tape, the encoded program and encoded tape are mashed. The effect of this interleaving on virus definitions is discussed in Chapter 2.

As an afterthought, we might wonder if we can be sure that the proposed algorithm for U can be programmed for a two-tape Turing Machine in such a way that U does not have to write on the tapes, in order to remember the intermediate result of such actions as "searching a matching string". In other words, can we be sure that at any moment of its computation, the content of the tapes of U can be interpreted as the encoded program of M or the encoded state plus encoded tape of T, respectively?

This afterthought is partially addressed by Hopcroft and Ullman in [4]. They propose a universal machine M_1 that uses three tapes: one for the (encoded) program and input; one for the (encoded) simulated tape; and one tape for the (encoded) state. The unused part of the third tape may be used as a 'scratchpad', so we can be fairly confident that first two tapes contain valid encodings at all times. Note that the translation of machines with *multiple* tapes to machines with *one* tape involves an 'explosion of symbols' [4, Theorems 7.1 & 7.2]. If we consider a one-tape universal machine, we don't have a clean separation of encoded program, encoded tape, and scratchpad.

Also note that Hopcroft and Ullman construct a machine that is constructed to be universal for Turing Machines that accept input strings, not for machines that compute functions on input strings.

1.5.2 Turing's Universal Machine

We would like to give a very short introduction to Turing's original "universal computing machine". For an overview of Turing's computing machines, see Section 1.4; we build upon the notions detailed in that section. First of all, let us note that Turing's approach has the following properties:

- By representing states and symbols as unary strings with distinguishable separators, Turing can encode the program of any machine.
- The machine uses a single tape.
- The machine has squares that are write-once (the F-squares) and that contain all the information about the simulated machine: encoded program, encoded tape content and encoded state. Other squares (the E-squares) are used solely for the computation of the universal machine itself.
- The universal machine keeps track of the entire computation of the simulated machine, not just the current tape and state at any given time.

Definition 1.54 (Turing's encoding)**.** We translate the states q_i ($0 \leq i < \omega$), symbols s_j ($0 \leq j < \omega$) and quintuples to a string over alphabet $\{A, C, D, L, R, N, ;, :, ::\}$ by the function below. (Think of the symbol 'D' as 'delimiter').

$$\begin{aligned}
&\mathit{translation}(q_i) = DC^i \\
&\mathit{translation}(s_i) = DA^i \\
&\mathit{translation}((q_a, s_b, q_c, X, s_d)) = DC^a DA^b DC^c XDA^d \qquad \text{for } X \in \{L, R, N\}
\end{aligned}$$

Figure 1.9 – Example tape content of Turing's universal machine

In this example we only show the content of the F-squares (white). We have a left-hand marker followed by a description of a machine program In the program, each coded quintuple is separated by a semicolon. The end of the program is marked with a 'double-colon'(::). After that, we see three sequences separated by colons. Each such sequence is a 'complete configuration', encoding the tape content, tape head position and state. If we leave out the E-squares we can read the example as in Table 1.5.

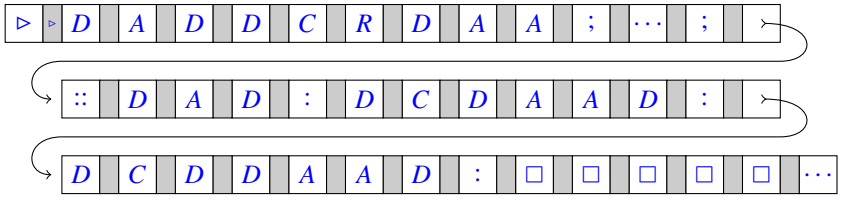


Table 1.5 – Interpreting the Universal Turing Machine tape

We interpret the example of Figure 1.9 by assuming that the simulated machine uses figures $s_0 = \square$, $s_1 = 0$, $s_2 = 1$. The left-hand side symbol is not explicitly encoded, so we start each tape with $\triangleright\triangleright$.

Type	**Symbols**	**Value**	**Interpretation**		
Program quintuples	*DADDCRDAA*	$q_1 s_0 s_1 R q_2$	$(a, \square, 0, R, b)$		
	...	...	...		
Type	**Symbols**	**Value**	**Interpretation**		
			State	*Tape*	*Position*
Complete configuration	*DAD*	$q_1 s_0$	q_1	$\triangleright\triangleright\square$	1
	DCDAAD	$s_1 q_2 s_0$	q_2	$\triangleright\triangleright 0\square$	2
	DCDDAAD	$s_1 s_0 q_2 s_0$	q_2	$\triangleright\triangleright 0\square\square$	3

Figure 1.9 is an example due to [13]. To determine the computed sequence of the simulated machine, we have to look at the machine configurations of the simulated machine that U writes on the tape. Each progressive configuration *may* contain a new figure. It is possible, but non-trivial, to filter out these printed figures and determine the entire computed sequence of the simulated machine.

A Universal Turing Machine is itself not a Turing Machine in the strictest sense. After all, the machine is started with some symbols on the tape, and not with a blank tape. However, the mechanical machine itself is unchanged.

1.5.3 Universal Talkative Machine

In Section 1.4.3 we defined 'Talkative Turing Machines', a concept that unifies the interpretation as function and interpretation as computing a sequence. We are confident that there exists a *universal* talkative machine: a talkative machine that can simulate any talkative machine. Such a machine is universal in two senses:

- Its computed sequence can be interpreted as the computed sequence of the simulated machine.
- It can be interpreted to compute the same function as the simulated machine.

Arguably, it is not trivial to find the transition function of a machine that accomplishes both tasks. It is especially tricky to find and define sensible encoding functions. We have to encode transition function, input and output using auxiliaries (i.e. words over $\mathbb{A}$), while the computed sequence is encoded in figures (i.e. infinite sequences of symbols in $\mathbb{F}$).

Constructing a universal talkative machine is a project unto itself. We encourage any reader who is interested in such puzzles to take on this project.

1.5.4 Universal Persistent Turing Machines

The concept of universality extends from Turing Machines to Persistent Turing Machines. Quoting Goldin et al. [26]: "A PTM is a universal persistent Turing Machine if it can simulate the behaviour of every PTM". Unfortunately, the formal definition of a universal PTM in [26] is flawed: it can simulate a single arbitrary PTM rather than *all* PTMs. Nevertheless, a correct notion of simulation of PTMs could be defined. In order to be able to simulate any PTM we have to properly define an encoding function; we discuss the effect of a ill-defined encoding function in Section 2.4.3.

For our purposes it suffices to note that the simulation of a machine M by a universal machine U adheres to the following. Let (for all PTMs M) η be an encoding into Σ_U (the alphabet of U), and $\langle \cdot, \cdot \rangle$ an injection into Σ_U. Then:

- U has an initializing macrostep. From then on the work tape of U contains a pair of encoded machine M and encoded work tape of M:

$$\epsilon \xrightarrow[U]{\eta(M)/\epsilon} \langle \eta(M), \epsilon \rangle$$

- U simulates each macrostep of M:

$$w \xrightarrow[M]{i/o} w' \quad \Longrightarrow \quad \langle \eta(M), \eta(w) \rangle \xrightarrow[U]{\eta(i)/\eta(o)} \langle \eta(M), \eta(w') \rangle$$

We will come back to UPTMs in Section 2.4.3.

Chapter 2

Modeling Computer Viruses

2.1 Computers

Thimbleby, Anderson and Cairns claim that conventional Turing Machine models are not suitable for modeling computer viruses. Their critique is twofold: first, Turing Machines are idealisations of 'real computers'; secondly, Turing Machines do not have the basic ingredients to define viruses. Let us quote from [21]:

> Turing Machines are infinite machines, whereas personal computers are finite. Clearly, Turing Machines are an idealisation. Personal computers have properties not shared with Turing Machines. Some real-world properties of personal computers -plausibly including the issues of trojans and viruses- are not modelled by the idealisation.

In [22] Cohen retorts by reasoning ad absurdum:

> In terms of Turing Machines being infinite and real computers being finite, of course this is essentially true. But the real question is whether the finiteness of real computers makes much of a difference. For Turing Machines, problems are 'undecidable', while for real machines, they are simply too complex to be solved by any real-world solution mechanism.
>
> The problem with a 'theoretical framework' for 'real computers' is that it is, in some sense, an oxymoron. Theoretical frameworks and models are designed to abstract the critical elements of real systems for the purpose at hand so that we can use reason and mathematics about those elements while ignoring things we consider irrelevant for our purposes. Real systems run on real computers, and real computers are subject to all sorts of physical issues, like solar flares and power failures, that might impact a lot of theoretical results if we were forced to include them. The question is how far to go in modeling the 'real world' and the answer depends on what you are trying to accomplish with your models.

We feel that Cohen misrepresents the intentions of Thimbleby et al. Even if the statement of 'real computers being finite' is slightly off topic (as none of these authors explores this further), Thimbleby et al. raise a perfectly valid question: "Which (real-world) properties of personal computers are necessary or sufficient to model computer viruses?".

Thimbleby et al. maintain that Turing Machines are not a suitable basis to model computer viruses. In fact, they feel that "closer analysis [of viruses, ed.] requires concepts not usually considered in the standard models of computation". Some key issues:

- Viruses have to 'enter' a system in order to infect. This requires modeling interaction of (computer) systems.
- Viruses occur on systems with limited (finite) memory.
- Viruses are only identifiable through their behaviour.

- To model infection or replication of programs within a computer, the model needs to identify at least two programs.
- Viruses need to be able to write (or output) programs in two senses: be able to output any program symbol, and have access to the locations where programs are stored.

Assume that Thimbleby et al. are correct in saying that we need something other than standard models of computation, in particular something other than Turing Machines. Then there are two avenues of escape: (1) use another (less common) model of computation or (2) do not use a model of computation at all. Before we choose, let us consider the following basic questions: what is computation and how is computation related to computers?

The question "what is computation" is the basis of considerable dispute. There are those who maintain that the Turing Machine is the prototypical model of 'effective calculability' and that this model (just as each equivalent model) captures all forms of computation. There are also those who claim that 'there is more' than 'computation' in the regular sense. According to this view, as expressed by Goldin, Wegner and Eberbach in [27, 31], the concept of 'computation' is often misinterpreted in the literature. They identify four fallacies that together constitute what they call the 'Turing Thesis myth'[31]:

1. All computable problems are function-based.
2. Turing Machines can compute any solvable problem.
3. Turing Machines serve as a general model for computers.
4. A Turing Machine can simulate any computer.

Goldin and Wegner do not just criticize, but contribute positively with a new paradigm: 'interactive computation'. One example of a model of interactive computation, that stays close to modern Turing Machines, is the 'Persistent Turing Machine' (PTM) [38]. Goldin et al. show that PTMs somehow capture 'more' than the common notion of computation (which is function based). It remains to be seen whether PTMs are themselves a suitable model of computers (see Section 1.3 and Section 2.4.3). They do capture one aspect of modeling viruses that modern Turing Machines do not: interaction with the environment. In principle it should be possible to model 'a virus entering a computer' using PTMs.

Goldin and Wegner dispute the fallacy that Turing Machines "serve as a general model for computers" partly by pointing out the importance of the *semantics of a model*. As we *interpret* what Turing Machines do and what 'computers' do differently, the former can never be considered a model of the latter. As we will see in Section 2.3, it is one of the key shortcomings of the work of Cohen that he fails to consider the semantics of Turing Machines. The same can be said of some of the criticasters of Goldin and Wegners work. Particularly, in [36], Cockshott c.s. reduce a persistent Turing Machine to a regular one-tape Turing Machine, without taking into account the semantics of either kind of machine. Without the semantics, such a reduction is not only trivial, but also pointless; taking the semantics into account, such a reduction is simply not possible as the one is more expressive than the other.

Let us now try to describe 'computers'. Computers have undergone a lot of change during the last 100 years; we can roughly distinguish three kinds of computers. Firstly: human beings that perform calculations. Secondly, machines that perform calculations. Thirdly and lastly, the machines that we use in every day life today, ranging from desktop computers to, for example, 'computerized' car control systems. Let us call this category the 'modern computer'. We have only to pick up a textbook on the history of computing to read up on this fascinating transformation.[1] In the absense of a taxonomy of modern computers that 'occur in the wild', we shall have to appeal to common sense when deconstructing a 'modern computer' into its essential components.

We look at Maurer's view of computers. In [34] he presents a model that could be praised for its close resemblance of the (electro)mechanical functioning of a modern computer [37]. His model is very general, because he used very simple ingredients.

[1] For instance, a short history is presented by Copeland in [13].

1. A computer has a memory. Without determining what shape this memory should be, we can represent the memory as a set M. This set could, for example represent RAM memory, CPU registers or harddisk, or any combination thereof.
2. The elements of the memory should in principle be able to have a value. Which values we may encounter is determined by the 'base set' B. The base set contains at least two elements.
3. The content of the entire memory (the value of each element) is determined by a map $S : M \to B$. A class $\mathcal{S}$ of maps S determines which combinations of values are possible. For example, if the values of the elements are independent, we get the full class of maps. Otherwise this class may be restricted.
4. A computer has a mechanism that causes the memory content to change. We call these changes (and at the same time their cause) an 'instruction'. An instruction is a map $I : \mathcal{S} \to \mathcal{S}$. A class $\mathcal{I}$ of maps I determines which instructions are allowed.

We can now define what Maurer calls a 'computer' (and what we call a 'Maurer Computer'). The reader might want to skip the two axioms, as they are of no consequence for our discussion. We have included the axioms because they are an integral part of Maurer's 'computer'. They guarantee that we can compare instructions based on their effect on the states of the computer.

Definition 2.1 (Maurer computer)**.** A structure $(M, B, \mathcal{S}, \mathcal{I})$, as outlined above, is a Maurer computer if it satisfies these two axioms:

- Axiom 1. Any 'recombination' of states is admissible:

$$\begin{aligned} &\forall\, S, S' \in \mathcal{S},\ \forall N \subset M, \forall T : M \to B \\ &\quad \text{if} \quad \wedge\, x \in N \implies T(x) = S(x) \\ &\qquad\quad \wedge\, x \notin N \implies T(x) = S'(x) \\ &\quad \text{then}\ \ T \in \mathcal{S} \end{aligned}$$

- Axiom 2. Two states only differ in a finite way:

$$\forall S, S' \in \mathcal{S}\ \{x \in M \mid S(x) \neq S'(x)\} \text{ is finite}$$

Remark 2.2. The reader may notice a certain similarity between Maurer's construction and our redefinition of Turing Machines in Section 1.2. This is not just because we were inspired by his article; there are intrinsic similarities between the two models. Let (Q, Σ, tr, q_o) be a Turing Machine, then we can interpret the Turing Machine as a Maurer computer (the reader may verify that the axioms hold):

$$\begin{aligned} M &= \{state\} \cup \omega \cup \{position\} \\ B &= Q \cup (\Sigma \cup \{\square\}) \cup \omega \\ S_{(q,t,p)} : n &\mapsto \begin{cases} q & \text{if } n = state \\ t(n) & \text{if } n \in \omega \\ p & \text{if } n = position \end{cases} \\ \mathcal{S} &= \left\{ S_{(q,t,p)} : M \to B \mid (q,t,p) \in \mathbb{C}_M \right\} \\ I_{(q,t,p)} : S &\mapsto \begin{cases} S_{(q',t',p')} & \text{if } S = S_{(q,t,p)} \text{ and } S_{(q,t,p)} \hookrightarrow_M (q',t',p') \\ S & \text{otherwise} \end{cases} \end{aligned}$$

The transition relation from Definition 1.10 can be seen as an instruction operating on a 'memory'. In that case the 'memory' would be the combination of machine state, tape and tape head position. When we say that the transition relation is a 'function' we express that it is fully determined what the Turing Machine will do, i.e. which change will occur to this combined 'memory' (of tape, state and position). We then have a special case: the class $\mathcal{I}$ is not the full class of instructions, but a class such that for any content of the memory, only one instruction applies in a non trivial way.

Note that the above 'reduction' from Turing Machine to Maurer computer is a sketch at best. Even if we define control, we have not specified at what moment we look at the memory and how we interpret its content.[2] In other words: no semantics.

Maurer introduced his model to discuss a "theory of computer instructions". Absent from the model is a unit of 'control': a mechanism to determine which instructions to apply. In a mechanical sense we can perceive the control unit to be separate from the memory; in models of computation the two are usually distinguishable as well. This is most apparent in Turing Machines: the machine proper is separate from the tape.

One of the most important contributions of Turing's 1936 article [28] is that we can 'push the control into memory'. His universal machine still has a separate control (its program), but there is a clear causal relation between the encoded Turing Machine that is written on the tape and the behaviour of the universal machine. For Maurer computers, something similar is achieved in [37, Section 10]. The concept of pushing the machine controls into the memory, is commonly referred to as the concept of 'stored programs'. According to Copeland, amongst others, the 'stored program' marks the birth of the modern computer [13].

We can specify the requirements of control in the Maurer computer framework, by extending it with two maps:[3]

5. A computer has a control unit that determines the next instruction to be applied, formalised as a map $C : \mathcal{S} \to \mathcal{I}$ from states to instructions.

Now we can choose to 'push the control into memory', by adhering to the following convention.

6. The memory contains a 'next instruction' NI subset that can hold a coded instruction. A map $DEC : \{S \restriction NI \mid S \in \mathcal{S}\} \to \mathcal{I}$ decodes the next instruction. We make sure that the control unit respects the stored instruction by specifying that $C(S) = DEC(S \restriction NI)$.

The addition of a 'control mapping' C is sufficient to express the (possibly infinite) process of sequentially applying instructions to the memory. We view the single application of an instruction as a 'move'.

Definition 2.3 (Maurer moves)**.** Let $(M, B, \mathcal{S}, \mathcal{I})$ be a Maurer computer. Maurer moves are defined as a binary relation $\hookrightarrow_{cc}$ on $\mathcal{S}$ such that

$$S \hookrightarrow_{cc} T \iff T = C(S)(S)$$

Note that a sequence of consecutive moves is only infinite under strong conditions: $DEC(X \restriction NI)$ and $I(X)$ must be defined for $S \hookrightarrow_{cc}^{n} X$ for arbitrary $n < \omega$ and $S \in \mathcal{S}$. In words: the next instruction must be intelligible, i.e. $DEC(X \restriction NI)$ must be defined, and applicable to the memory, i.e. $I(X)$ must be defined.

What is an instruction exactly? That depends upon our interpretation. We can think of elementary building blocks of a computational model, for example the elementary operations of a Turing Machine: 'move', 'change state', 'write symbol'. Thus we might interpret a sequence of instruction applications as the computation of a function. Or we can stay close to the electromechanical construction of present day computers and define one instruction for each CPU instruction. We can even create more fine-grained modeling of the internal processes of the CPU.

Now, if a program is a kind of sequence of instructions in memory, it is not yet well defined. We would be hardpressed to define it on the sequence of applied instructions. For where does the sequence start or end?

For the remainder of this section, we will interpret instructions at a 'higher level': let $\mathcal{I}$ be the class of 'effectively computable' maps on $\mathcal{S}$.[4] In this case it is more appropriate to think of an instruction, as

[2] In this case the control would be: apply the only applicable instruction. Note that this sounds simpler than it actually is: because we defined the set $\mathcal{I}$ on top of $\hookrightarrow_M$ we have hidden the computation within this class $\mathcal{I}$. In some sense, determining which instruction applies is the 'real' computation that is otherwise effected by the mechanical process in the Turing Machine.

[3] We choose a modeling of 'control' that is more generic than that of Bergstra and Middelburg in [37, Section 5]. They extend the Maurer computer model with Basic Polarized Program Algebra and call the extended model a 'Maurer machine'. We need no such algebra for our purposes.

[4] For a more in depth discusion of what we mean by such a class of maps on $\mathcal{S}$, see Chapter 3.

encoded in NI, as a program for that function. Thus 'applying instructions sequentially' comes down to executing programs sequentially. We say 'programs' in plural because each instruction application may cause the content of NI to change. If $S \hookrightarrow_{cc} T$ then $S \upharpoonright NI$ may differ from $T \upharpoonright NI$ and therefore the instruction (now 'program') $C(S)$ may differ from $C(T)$.

In general, an instruction operates on the whole memory and therefore has complete access. By considering appropriate subsets of the full class $\mathcal{I}$ of instructions, we could model restrictions on 'programs' and explicitly define whether and how programs have access to other programs.

Wrapping up, if we want to model:

- 'Stored programs', and thereby model infection of programs
- Sequential 'execution' of programs; and thereby model a virus life cycle
- Access to programs, and thereby defense against infection

then the following simple ingredients are sufficient:

- Memory that holds values
- Instructions that cause the memory to change
- Control unit that determines the instruction to be applied
- Coding of programs in memory

Should the concept of 'programs' be a part of a computer model or should we restrict such a model to deal with CPU instructions? It can be argued that the concept of 'programs' only enters the stage if we model computer 'operating systems'. For example, in [24], while arguing that universal Turing Machines are unsuitable to model computer viruses, Thimbleby et al. state: "These problems are not insurmountable, but [the universal Turing Machine, ed.] $\mathcal{U}$ somehow has to be an operating system for TMs [Turing Machines, ed.]". In [5], Cohen introduces a new model of 'computers with operating system' to be able to model viral protection schemes. We will discuss this model in Section 2.4.2.

However, as Cohen rightly states: "The question is how far to go in modeling the 'real world' and the answer depends on what you are trying to accomplish with your models.". We feel that to define computer viruses, we do not need to model operating systems in full detail. To define viruses for computers as generically as possible, we want to include only these minimal elements of operating systems: multiple 'stored programs' and 'sequential execution' of programs (as we explained above). At a later stage we may want to consider the element of 'interaction' (or external input and output) to model infection between computer systems.

2.2 Turing Machine Viruses

According to Cohen "a virus may be loosely defined as a sequence of symbols which, upon interpretation, causes other sequences of symbols to contain (possibly evolved) virus(es)"[16].

For the better part of the 20$^{\text{th}}$ century, the dominant model of computation has been the 'Turing Machine'. It therefore seems appropriate to combine the notion of computer viruses with Turing Machines. It is this elementary combination that has lead to Cohen's formal model of computer viruses. In his model, a virus is a sequence of symbols *on the tape of a Turing Machine*, with certain properties.

Are the various Turing Machine models appropriate models of modern computers? Are they an appropriate basis for modeling computer viruses?

2.2.1 Modern Turing Machines

Stored Programs

A modern Turing Machine is understood to 'compute a function'. In Definition 1.22, we expressed the interpretation of a machine as 'computing a function from strings to strings'. This function is completely determined by the machine's transition function. The symbols that are on the tape when the machine is started, are understood as 'input' and they determine to which input the function is applied. There is no prior reason to suspect that this input contains a program of whatever form or shape.

The modern version of the 'universal machine' introduced by Turing (see Section 1.5) computes a function: the so called 'universal function'. As the universal machine is itself a Turing Machine, it computes a function from strings to strings. The concept of a universal machine, however, is to take a different stance: we fix the transition function and vary a part of the input which we now call a '(stored) program'. This 'program' is the encoding of another Turing Machine. The second part of the input is what we now solely perceive to be the 'input'. The interpretation of a universal machine is that, given a program, it computes the function corresponding to the machine encoded by the program.

If we do not specify what constitutes a valid encoding of machine and input, we can view every Turing Machine as a machine that interprets part of its input as the code of another machine. The most trivial example is a machine that interprets an empty string as the 'code' of itself. A slightly less trivial example is a machine that interprets the entire input of a machine as the 'code' of a constant function. These examples stress that it depends on our interpretation which symbols of the input can be understood to be 'interpreted' by the machine. In general, when we use Turing Machines without an explicit interpretation, 'programs' are not available.

What about a universal machine? The encoding of machines as programs is usually an injective mapping, the reason being that the universal machine itself needs to be able to decode the content of the tape. As a consequence, the content of the tape of the universal machine is decodable in general; we say that program and input are 'separable'. If they are not separable, we lose the correspondence between program and computed function. We will assume that we have an implementation of the universal machine in mind, as well as the corresponding encoding. Then each time we run the universal machine, we can identify one encoded machine, or 'program', that determines the behaviour of the universal machine: it computes the function of the coded machine applied to the 'input'.

Such a universal machine, by definition, can simulate any other Turing Machine; as such there is a plethora of programs that might be on its tape. At any particular time, however, we can discern only a single program on the tapes. That single program, therefore, could never 'infect' another program.

REMARK 2.4. Notice that a sequence of symbols that we do understand to be a 'program', need not be a contiguous sequence on the tape. In the implementation of Universal Turing Machines in the literature, the encoded machine is usually interleaved with input data, output data and temporary symbols (see Section 1.1).

Sequential Execution

As stated frequently, modern Turing Machines 'compute a function'. In this interpretation, our focus is on machines that 'halt'; what happens *after halting*, is completely unspecified. This also applies to the universal machine. After a single stored program is executed, the machine halts. As such the model gives us two points to inspect the machine: before execution and after execution. This is not sufficient to define a virus lifecycle.

Execution, interpretation or simulation

Universal (Turing) machines simulate other machines; in the previous section we indicated that Turing's universal machine models 'stored programs'. How should we understand that statement? Let universal machine U simulate M by means of the 'stored program' P. Then P somehow encodes the transition function tr of M. We have seen in Remark 2.2 that tr can be understood as a set of instructions that cause the 'memory' of M to change when applied.

A single instruction of M applies to a single symbol on a single square. As U has a finite alphabet but should be able to simulate machines without any prior bound on the size of the alphabet, it is safe to assume that a single symbol (or square) of M corresponds to a set of symbols (or squares) of U. Therefore U cannot 'execute' a single instruction i of M (in P) as a single instruction of U; we may not even assume that i corresponds to a unique sequence of instructions $j_1, \ldots, j_n$ of U as it depends on the current state of U and content of U's tape how i should be simulated. So P is not a set of instructions that cause the 'memory' of U to change.

As such we should perhaps not call P a 'program' that is 'executed', but rather a high level 'program' that is 'interpreted'. It is even liberal to speak of 'interpretation', rather than 'simulation'. In Section 2.4 we will discuss the fact that the simulated machine M can only affect a part of the tape of U (the part corresponding to the simulated tape of M); in Section 2.4.3 we will discuss the fact that a simulated machine M cannot reasonably be understood to output programs (the code of other machines).

2.2.2 Turing's original machine

In Turing original machine model, the a-machine, it is apparent that we cannot model 'stored programs'. For Turing, a machine starts with an empty tape; there is no input that can contain a stored program. As such, a machine can only be understood to compute a single sequence.

You may find this at odds with the fact that it was Turing himself who introduced us to the concept of stored programs by defining a 'universal machine'. We can only remind you that in the strictest sense, this 'universal machine' is not a 'Turing Machine' itself since it starts with a tape that is not empty. The machine starts with exactly one description of another machine (exactly one stored program) on the tape. Again, a single program does not suffice. Turing's universal machine simulates a single machine at a time. As the simulated machine will compute ad infinitum, the universal machine will not get the chance to execute another program, even if it were available.

2.3 Cohen

2.3.1 Cohen Machines

Cohen can be considered the founding father of the study of computer viruses. The study of computer viruses is a direct result of his 1985 Ph.D. thesis 'Computer viruses' [5].[5] According to Cohen "a virus may be loosely defined as a sequence of symbols which, upon interpretation, causes other sequences of symbols to contain (possibly evolved) virus(es)" [16]. He goes on to make two more informal statements:

> If we consider an interpreted sequence of symbols in an information system as a program, viruses are interesting to computer systems because of their ability to attach themselves to programs and cause them to contain viruses as well.
>
> We informally define a computer virus as a program that can infect other programs by modifying them to include a, possibly evolved, copy of itself.

Cohen formalises the notion of virus using Turing Machines. We will show that Turing Machines as he defines them are unlike either modern Turing Machines or Turing's a-machines (see Section 1.2 and Section 1.4). To avoid confusion, we will call his machines 'Cohen Machines'.

Definition 2.5 (Cohen Machine)**.** A Cohen Machine is a structure $M = (Q, \Sigma, tr)$, where Q is a finite set of machine states, Σ is a finite set of symbols and tr is the total transition function defined below. The (total) functions n, o, d can be viewed separately as indicating the next state (n), the output symbol (o) and direction of movement of the tape head (d).

$$tr : Q \times \Sigma \rightarrow Q \times \Sigma \times \{-1, 0, 1\}$$
$$tr(q, s) = (n(q, s),\ o(q, s),\ d(q, s))$$

[5] If you have no access to this work, you can consult either [16] or [8, Appendix B], as both copy from his earlier thesis. You can also read [14, Section 3.2], which presents Cohen's formalisation of viruses in a similar fashion.

First of all we notice that a blank symbol 'blank' is not identified, either in Σ or outside of it. Then we notice that unlike Definition 1.10, the starting state is not uniquely defined for a Cohen Machine. We are free to choose any initial configuration $(q, t, p) \in \mathbb{C}_M$ where

$$\mathbb{C}_M = \{(q, t, p) \mid q \in Q,\ t : \omega \to \Sigma, p \in \omega\}$$

To ensure that the tape head never falls off the left end of the tape, the moves from one configuration to the next are slightly restricted (compare with Definition 1.12):

Definition 2.6 (Cohen moves)**.** Cohen moves are defined as the binary relation $\hookrightarrow_M$ on $\mathbb{C}_M$ such that $(q, t, p) \hookrightarrow_M (q', t', p')$ if

$$\begin{aligned} q' &= n(q, t(p)) \\ t' &= t[p \mapsto o(q, t(p))] \\ p' &= \begin{cases} 0 & \text{if } p + d(q, t(p)) < 0 \\ p + d(q, t(p)) & \text{otherwise} \end{cases} \end{aligned}$$

Because the transition function and n, o, d are total functions $\hookrightarrow_M$ is a total function on the domain $\mathbb{C}_M$. In other words: each configuration has a next configuration. Now Cohen defines 'histories', in effect his bookkeeping to keep track of the moves of a Turing Machine.

Definition 2.7 (Histories)**.** Let M be a Cohen Machine. A history h is an infinite connected sequence of M-configurations. The set of all histories is defined as

$$\mathbb{H}_M = \{h :\ \omega \to \mathbb{C}_M \mid (\forall\ i < \omega)\,(h(i) \hookrightarrow_M h(i + 1))\}$$

A history is an infinite sequence of configurations. This expresses the fact that Cohen Machines never halt. Nevertheless, Cohen defines that a machine is understood to 'halt' if it reaches a configuration for which the transition function is invariant, i.e. if for some history $h \in \mathbb{H}_M$ and some $i \in \omega$ we have $h(i) = h(i + 1)$. (In fact, this implies $h(i) = h(j)$ for any $j > i$.)

How does a Cohen Machine start? We noticed that a starting state is not part of the definition of a Cohen Machine. Cohen specifies that a machine has an initial configuration (the first element of a history):

$$(\forall\ h \in \mathbb{H}_M)\,(h(0) = (q_{M0} \in Q, t_{M0} : \omega \to \Sigma, p_0 \in \omega))$$

The trouble is that the value(s) of q_{M0}, t_{M0} and p_0 are unspecified. This leaves us with a few different interpretations of Cohen Machines. Unfortunately, Cohen is does not tell us how his machines should be understood. Are they modern Turing Machines or Turing's original machines. We will have to take a guess. There are three possibilities:

- All of q_{M0}, t_{M0} and p_0 are uniquely defined for a machine M. This interpretation is supported by the way Cohen defines $h(0)$. However, this implies that a machine can only perform a single 'computation', i.e. the set $\mathbb{H}_M$ is always a singleton. It therefore seems unlikely that this was Cohen's intention.

- All of q_{M0}, t_{M0} and p_0 can be varied. From the way $h(0)$ is defined, this seems the second best guess. However, under this interpretation, $h(0)$ is a very big set. In fact it is so big that it is unique for each M. Let $M = (Q_M, \Sigma_M, tr_M)$ and $N = (Q_N, \Sigma_N, tr_N)$ be Cohen Machines. Then $\mathbb{H}_M = \mathbb{H}_N \implies M = N$.

  Proof. For any $q_i \in Q_M$ and any $s \in \Sigma_M$, since tr_M is a total function, there are $q_j \in Q, t \in \Sigma_M$ and $m \in \{-1, 0, 1\}$ such that $tr_M(q_i, s) = \left(q_j, t, m\right)$. Then there is $h \in \mathbb{H}_M$ such that $h(0) = (q_i, tape, 0)$, $h(1) = \left(q_j, tape', m\right)$ and $h(0) \hookrightarrow h(1)$, where $tape(0) = s$ and $tape'(0) = t$. Since $\mathbb{H}_M = \mathbb{H}_N$ it must be that $h \in \mathbb{H}_N$. Then it must be that $q_i \in Q_N, s \in \Sigma_N$ and $tr_N(q_i, s) = \left(q_j, t, m\right)$. Then $Q_M = Q_N$, $\Sigma_M = \Sigma_N$ and $tr_M = tr_N$. □

It is unclear how a Cohen Machine should be understood, in this setting. The notion of histories does not help us here, because the equivalence deduced from it is very uninteresting.

- A third alternative would be that q_{MO} and p_0 are uniquely defined, whilst t_{M0} may be varied. Although this reading cannot be endorsed by Cohen's definition, it is supported by the examples of machines in his thesis. This alternative makes Cohen's machine almost like modern Turing Machines, because the initial state and tape head position are fixed whilst we can vary the input.

  To express the computation of functions, we have to overcome an additional problem in Cohen's definition: there are no blanks. In particular, Cohen allows for infinite input on the tape, which is not allowed in the setting of modern Turing Machines (see Section 1.1). Another way to view the situation is that the set of histories is still rather big. To define a function on finite strings using these machines, we have to be explicit about how we represent finite strings as infinite strings; i.e. we need an (injective) 'encoding' $enc : \Sigma^* \to \Sigma^\omega$. Such an encoding is blatantly absent from Cohen's definitions in [5].

The received view of Turing Machines is that they compute functions. We will therefore assume that Cohen had this interpretation of machines in mind. At the face of it, Cohen Machines are unlike modern Turing Machines. We choose to 'repair' this by using Definition 1.10 as a definition of modern Turing Machines instead.

Are Cohen Machines similar to Turing's Original Machines?

The only reference in [16] that discusses Turing Machines is Turing's original 1936 article [28].[6] We feel obliged to discuss wether Cohen Machines should be interpreted the same way as Turing's original machines (see Section 1.4. Interestingly, the nomenclature in Cohen's article (e.g. "finite state machines") and the definition of (transition) functions o, n, d seems to stem from [2, 4] rather than [5]. But there are more serious discrepancies.

Again, we have trouble interpreting Cohen's initial configuration. First of all, whether the initial tape-head is predefined or variable, a Cohen Machine does not necessarily start on the first square of the tape, as Turing's original machines do. Moreover, a Cohen Machine starts with a non-empty tape rather than with an empty tape. If the input for Cohen were finite, such problems could be overcome, as it is provable that any sequence that can be computed from a finite string, is also computable from an empty string. Cohen, however, allows infinite input, which defeats the whole purpose of Turing's article [28]. As noted in [13, page 35], every sequence is trivially computable from infinite input, by a machine that does not change the tape content.

Another discrepancy is the fact that all examples of Cohen Machines in [16] are machines that are *constructed to halt*. This seems to preclude the possibility that Cohen had Turing's original definitions and meaning in mind, as the machines that *have meaning* according to Turing, are those that are 'circular', i.e. print infinitely many figures.

Lastly, we observe that Cohen machines may write symbols unrestrictedly, without distinguishing between the two sorts of symbols, namely figures and auxiliaries. As such, we cannot apply Turing's definition of the *computed sequence* (see Definition 1.40).

If we were to 'repair' Cohen Machines so that they were like Turing's original machines, we could consider a subset of Cohen machines that is well-behaved w.r.t. Turing's conditions. As we have seen in Section 1.4.1, it is non-trivial to restrict the class of (Cohen) machines based on the transition function. Notably, the effect of such a restriction of viruses (as defined in the next section) has not been investigated by Cohen.

We cannot but conclude that Cohen Machines are unlike Turing's original machines, Cohen's reference to "On computable numbers, with an application to the Entscheidungsproblem" [28] notwithstanding.

[6] The same is true for Cohen's [5, chapter 2] and [8, Appendix B].

2.3.2 Cohen Viruses

We will now discuss Cohen's formal definition of viruses. Our notation differs slightly from that in [5]. First of all, we use Lamport's style of writing long formulas [20] (see also Appendix B). Secondly, we define VS as a class function rather than as a set of tuples. Lastly we use some shorthand notation introduced in Appendix A. The careful reader of this work and Cohen's will see that the differences are immaterial.

We would like to point out to the reader that we define viruses for Cohen Machines as defined in the previous section. That means that the start state and start position are ambiguous. This way we stay as close as possible to Cohen's definition of viruses. Later on we will show that we can apply this definition of viruses to the Turing Machines (of Definition 1.10) by redefining the set of histories.

Definition 2.8 (Cohen's viruses)**.** For a Cohen machine $M = (K, \Sigma, tr)$, let $V \subseteq \Sigma^*$. We say that V is a viral set for machine M, written as $V \in \mathrm{VS}(M)$, if and only if the formula below is true. In this formula we implicitly quantify over the elements of a history (notably these variables: q_i, t_1, p_1, t_2, p_3) by quantifying over h, n_1, n_2 and n_3. We also implicitly quantify over elements that will not occur a second time in the formula and we denote them by a underscore.

$$\forall v \in V,\ h \in \mathbb{H}_M,\ n_1 < \omega \tag{2.1}$$
$$\text{if} \quad \wedge\ h(0) = (q_i, \_, \_) \tag{2.2}$$
$$\wedge\ h(n_1) = (q_i, t_1, p_1) \tag{2.3}$$
$$\wedge\ t_1[p_1, |v|] = v \tag{2.4}$$
$$\text{then } \exists\, v' \in V,\ n_2 < \omega,\ pos < \omega \tag{2.5}$$
$$\wedge\ h(n_2) = (\_, t_2, \_) \tag{2.6}$$
$$\wedge\ t_2[pos, |v'|] = v' \tag{2.7}$$
$$\wedge \quad \vee\ pos \geq p_1 + |v| \tag{2.8}$$
$$\vee\ p_1 \geq pos + |v'| \tag{2.9}$$
$$\wedge\ \exists\, n_3 < \omega \tag{2.10}$$
$$\wedge\ n_1 < n_3 < n_2 \tag{2.11}$$
$$\wedge\ h(n_3) = (\_, \_, p_3) \tag{2.12}$$
$$\wedge\ pos \leq p_3 \leq pos + |v'| \tag{2.13}$$

We can read the condition in the definition of Cohen-viruses as: "For a Cohen machine M, the set V is a closed set of viruses for M if and only if each virus $v \in V$ has the property that:

- if the machine encounters (the first symbol of) the virus v after n_1 moves, while in the same machine state as when the machine was started then
  - after n_2 moves (since starting), another virus $v' \in V$ will be on the tape, starting at position pos such that the strings v and v' do not overlap
  - and at least one symbol of v' was written by the machine at some point after encountering v (after move n_1), but before the whole virus v' is on the tape (before move n_2)."

The central notion of Cohen's definition of viruses is the 'viral set' V. This notion gives rise to a form of 'viral equivalence', although such a notion is not present in the work of Cohen.

Definition 2.9 (Viral equivalence)**.** Let M, N be Cohen machines. Then M and N are virally equivalent, denoted $M \equiv_{vir} N$ if and only if $\mathrm{VS}(M) = \mathrm{VS}(N)$.

How does viral equivalence relate to the notions of equivalence that we discussed in Section 1.2.3? This is a question we can only ask if functional equivalence is expressible for Cohen machines. To apply the above definition of viruses to the Turing Machines of Definition 1.10, we have to redefine the set of histories as a set of configuration sequences (see Definition 1.15). The definition of 'viral equivalence' for such machines is analogous to the above equivalence for Cohen Machines.

Figure 2.1 – Machines M and N that compute the same functions but have different viruses.
The transitions are labeled with the symbol read, symbol written and movement (R=right, L=left, N=None).

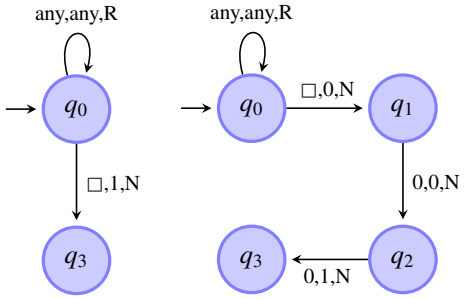
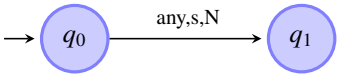


Figure 2.2 – For any distinct symbol s, this machine computes a different function, while it has not viruses.
The transition is labeled with the symbol read, symbol written and movement (N=None).



Definition 2.10 (Histories as configuration sequences)**.** Let M be a Turing Machine according to Definition 1.10. Let the set of 'histories' be defined as the set of configuration sequences for M, as in:

$$\mathbb{H}_M = \Big\{ h : \alpha \to \mathbb{C}_M \;\Big|\; \begin{aligned} &\wedge\, h(0) = (q_0, in \in \mathbb{P}_{\Sigma\cup\{\square\}}, 0) \\ &\wedge\, (\forall\, i, 0 < i < \alpha - 1)\,(h(i) \hookrightarrow_M h(i+1)) \\ &\wedge\, (\alpha = \omega \vee (0 < \alpha < \omega \wedge \nexists c \in \mathbb{C}_M\, h(\alpha - 1) \hookrightarrow_M c)) \Big\} \end{aligned}$$

Lemma 2.11 (Viral equivalence)**.** *Viral equivalence is incomparable to functional equivalence.*

Proof. We first show that $\equiv_{\text{func}} \not\subseteq \equiv_{vir}$. We define two Turing Machines that both compute $f : x \mapsto x;1$ (where ';' is string concatenation), but have different viruses. Let $Q = \{q_0, q_1, q_2, q_3\}$, $\Sigma = \{0, 1\}$. Let the transition functions of machines M and N be defined by the diagrams in Figure 2.1. Then both machines compute the function f. We notice that $\{0\} \in \text{VS}(N)$ whereas $\{0\} \notin \text{VS}(M)$.

We now show that $\equiv_{vir} \not\subseteq \equiv_{\text{func}}$. We define two machines that have the same viruses, but compute a different function. Let $\Sigma = \{a, b\}$ and the two machines A and B be defined by Figure 2.2 where $s = a$ and $s = b$ respectively. Neither machine has viruses, i.e. $\text{VS}(M) = \text{VS}(N) = \emptyset$, but A computes $x \mapsto x\,[0 \mapsto a]$ whereas B computes $x \mapsto x\,[0 \mapsto b]$.

Notice that one direction of the proof can be seen as a direct result of Claim 1.30: tape-equivalence is a refinement of functional equivalence. The notion of viral equivalence exemplifies that it does matter whether the tape is observable. □

The result of this lemma is that two machines that we consider equivalent may have different viral sets (and vice versa). The fact that Cohen's machines do not compute functions, seems to be built into his formalisation of computer viruses.

2.3.3 Interpreted Sequences

For Cohen, the first step to define viruses is to define strings, or words over the tape alphabet. Remember Cohen's informal statement that "a virus may be loosely defined as a sequence of symbols which, upon interpretation, [has certain properties, ed.]".

We have shown in Section 2.2 that it is not appropriate to call any string on the tape of a Turing Machine a 'program' that is 'interpreted'. In that section, we considered this issue for both modern Turing Machines

and Turing's original machines. We will now consider this issue specifically for Cohen's definition of viruses.

Cohen calls a (finite) string a "Turing Machine program". This in itself is misleading, since it seems to imply that the only interpretation of strings is that of 'programs'. Again, we have to face the fact that the initial configuration is defined ambiguously. First, suppose that the initial machine state and position on the tape are variable. Then indeed every string should be considered a program and potential virus. In fact, due to the quantification over all histories in Definition 2.8, the meaning of a string as a virus is independent of the position of a string on the tape, the current machine state or even the number of moves since the machine was started. In short, being viral is independent of the context. If being viral is independent of the tape content, no organisation or meaning we give to the tape content will have any bearing. We can only define such notions as 'file systems' or 'operating systems' independently of this definition of viruses.

Perhaps the more reasonable position is that we should not mind the awkward definition of Cohen Machines and apply the virus definition to modern Turing Machines. In this setting we are faced with the peculiar role that the initial state plays in the definition of viruses. In a way, it marks the beginning of a sequence that is understood as a program (or virus). The end of the sequence is not marked, but derived from the viral property because substrings of a virus are not necessarily viruses. What makes a string (or symbol) that is read when the machine is in the initial state a program? Or what makes strings (or symbols) that are read in other states not a program? The following lemma illustrates how puzzling Cohen's virus definition is.

Lemma 2.12. *Let* $\Sigma^n = \{t \in \Sigma^* \mid size(t) \leq n\}$. *Let* n *be any number* $n > 0$ *and* M *any Turing Machine (Definition 3.7) with alphabet* Σ *and initial state* q_0. *Let* f_M *be the function computed by* M. *If* M *does not return to its initial state and if:*

$$(\forall\, a \in \Sigma^n)\,(size(f_M(a)) > size(a) + 2)$$

then $\Sigma^n \in \mathrm{VS}(M)$.

Proof. In Definition 2.8, let $V = \Sigma^n$ and consider some $v \in V$. Since M does not return to its start state q_0, we only have to consider the case where $n_1 = 0$ and where v is an initial segment of the input. Then the antecedent, i.e. (2.2) through (2.4), is fulfilled by the fact that $h(0) = h(n_1) = (q_0, t_1, 0)$.

We have to make the consequent true, i.e. satisfy (2.5) through (2.13). Let $p_3 = pos = size(t_1)$. Since M writes at least a symbol at each of the two squares directly following t_1, it has to visit those squares after respectively i and j moves . Let $n_2 = j$, then $h(n_2) = (\_, t_2, \_)$ for some tape content t_2. Let v' be the last symbol to be written (before the j^{th} move) at the square inmediately juxtaposed to t_1, i.e. $v' = t_2(pos)$. Then we satisfy (2.6) through (2.9). Let $n_3 = i$, then we satisfy (2.10) through (2.13) and thereby (2.5). Then the consequent is true and v is a virus that 'evolves' to virus $v' \in V = \Sigma^n$. □

We simply do not understand in what sense the class of viral sets captures the notion of viruses. In the above lemma, is it reasonable to view each string of a certain length a virus for these machines? According to Cohen's definition they are, because they 'evolve' into a string of length one that is also a virus. We doubt that a virus that consists of a single symbol can be considered an evolved copy of a virus that consists of, say, a hundred symbols. The above also shows that if the entire input is a virus v, then v' is a single symbol concatenated to v itself. This is at odds with Cohen's informal statement that v should "infect other programs by modifying them [...]" or "attach themselves to other programs and cause them to contain viruses as well".

2.3.4 Computational Aspects

In the previous sections we have criticised the underlying machine model. This leaves us to ponder what Cohen's results in 'computational aspects' of computer viruses amount to [5, 16]. To answer this question we have to examine in what sense the his formalisation really captures computer viruses. What do the core elements of the formalisation mean?

The definition allows us to inspect whether "$v \in V \in \mathrm{VS}(M)$?", which can be understood as: "is the virus v an element of the closed set V of evolving viruses?", and "is this closed set one of the closed sets of evolving viruses for machine M?".

On the face of it, it would seem more reasonable to ask if v is a virus for M. It is doubtful whether this is appropriately answered by "$v \in V \in \mathrm{VS}(M)$". The very choice of using closed sets, however, is at the core of the definition; otherwise any combination of strings read and string written is a viral set. Is the closedness property reasonable? It excludes, for example, viruses that stop being infectious after several evolutions. Viruses that have just one possible non-infectious descendant are discounted. Treading the dangerous path of analogy with biology, this discounts such things as 'random mutation'.

On the other hand, Cohen's notion of viral sets includes too many 'viral' strings. Suppose we have a machine for which we only consider "singleton" viruses, i.e. viruses that cause an exact (non-evolved) copy of itself to be written. According to Cohen's results, $\mathrm{VS}(M)$ is closed under union. In particular $\bigcup \mathrm{VS}(M) \in \mathrm{VS}(M)$. It does not seem sensible to talk of such a union of singleton sets as a 'set of evolving viruses', as the separate viruses are completely unrelated.

In the previous section (Lemma 2.12) we saw an example of a class of machines for which all strings of a certain length are viruses. We could argue that the set of such strings is simply not interesting as a viral set and that we should look at 'more interesting' viral sets. If so, then the answer to "$v \in V \in \mathrm{VS}(M)$" is not an answer for the question if v is a ('interesting') virus for M.

Our doubts about the connection between viruses and Cohen's formalisation are strengthened by his computational results. For example in [16, Theorem 5] he proves that any string $t \in \Sigma^*$ is a virus for some machine $M = (K, \Sigma, tr)$. Cohen goes on to show that for any finite set $V \subset \Sigma^*$ there is a machine $M = (K, \Sigma, tr)$ such that $V \in \mathrm{VS}(M)$. Cohen proves this by defining a 'recognizer' Turing Machine that can recognize a set V. We would claim that this is just a restatement of the decidability of finite sets, and has nothing whatsoever to do with 'computer viruses'.

Intuitively, the 'closedness' property captures some part of the notion of viruses: replication and evolution. But it is neither necessary nor sufficient for defining computer viruses. Combined with our earlier criticism of the underlying machine model, we have to conclude that Cohen's results have no bearing on computer viruses. We throw out the most acclaimed result: the undecidability of computer virus detection. We view this result as the undecidability of the question whether any given set of strings for any Cohen machine has a special closure property. The undecidability of computer virus detection shall have to be proven using a more plausible formalisation of computer viruses.

2.4 Universal Machine Viruses

2.4.1 Universal Turing Machine

Universal Turing Machines are discussed in Section 1.5; this section discusses the merits of [18, 23]. In these articles, Mäkinen and Kauranen propose a refinement of Cohen's model of computer viruses. Their proposal is to use the Universal Turing Machine (as defined in [4]) as a model of computers. A virus would then be the encoding of a Turing Machine that copies itself somewhere on the tape or infects other Turing Machine encodings. By choosing an implementation of the Universal Turing Machine, the encoding of machines as strings is fixed, and Mäkinen et al. rightly notice that "by fixing the encoding for TMs we have fixed the set of strings which can be interpreted as viruses.". This addresses part of our criticism of Cohen's formalisation, namely that it is paramount to specify which strings can be interpreted as 'programs' (see Section 2.3.3).

In order to decide which strings *on the tape of a machine* can be interpreted as programs, it is not sufficient to determine if a string is the encoding of a machine; this is co-determined by the place it has on the tape. As discussed in Section 2.2.1, we can identify only a single program on the tape of a universal machine, at any given time. Incredibly, Mäkinen et al. seem to disagree with this basic fact. We quote their response to [21]:

> Their second argument concerns the 'other' programs. In conventional TM models there are no 'other' programs to be infected. This, of course, is not a problem in UTM based models where the tape may contain any number of programs, 'normal' programs as well [as] various malicious ones.

A clear understanding of (Universal) Turing Machines is called for; this is one of the purposes of this thesis.

As noted in [24], a Universal Turing Machine provides a kind of 'jail' from which a 'program' cannot break free. This 'jail' prevents the program from accessing the tape of the universal machine. Let universal machine U simulate machine M, then M has 'access' to a simulated tape that is encoded in some form or other onto the tape of U; M never has 'access' to the tape of U. Thimbleby et al. point out that it is of no help if M itself is a universal machine; this just gives a regress of M simulating M' and M' having 'access' to a simulated tape encoded on the simulated tape of M encoded on the tape of U.

According to Mäkinen et al., "one may think that different encodings for TMs correspond to different operating systems"; in the absence of a definition of 'operating systems', this is a dubious claim. At least, Thimbleby et al. seem to have a different view on 'operating systems'; they note that we can escape the infinite regress by changing our machine U in such a way that U "somehow has to be an operating system for TMs".

We maintain that if U somehow 'is an operating system for TMs', it will cease to be a Universal Turing Machine. As discussed in Section 2.1, an essential feature of such an 'operating system' (or computer model) is that it can execute programs sequentially. This concept is orthogonal to universal machines that simulate machines that compute a function. Thus we disagree that a 'Universal Turing Machine'-model for computer viruses can be sharpened to 'any level of fine granularity', as Mäkinen et al. would have us believe [23].

None of [18, 21, 23] refer to Cohen's original thesis work [5]; this is a pity, because Cohen already explored the idea of a 'kind of universal machine' along the lines that Thimbleby et al. suggest. This is the topic of the next section.

2.4.2 Universal Protection Machines

In [5, chapter 3], Cohen introduces the notion of 'Universal Protection Machines' as a model of a 'computer with an operating system'. We discuss it here to show the intrinsic confusion that arises from defining viruses as sequences of symbols on the tape of a Turing Machine, in particular as defined in Section 2.3.

The model is based on Cohen Machines (defined in Section 2.3). Its purpose is "to model the mutual effects of computation and protection".

Superficially, the UPM can be viewed as a kind of Universal Turing Machine: it is supposed to be a special Turing Machine that can schedule and execute 'programs' on the tape. The scheduling and execution of 'programs' for 'users' is restricted by a matrix of 'access rights'; Cohen uses these features to model the effects of user rights restrictions on the spread of viruses.

The most poignant similarity with the Universal Turing Machine is that, as a model, it is underspecified.[7] Quoting [5]:

> We now briefly summarize the operation of the UPM by description without formally specifying its operation. Perhaps the most important aspect of our description is that all operations and information stored as a result of these operations are finite, and can thus be performed in a finite number of moves of a TM. If all of these operations are possible for a TM, and if they can all be performed in finite time, then we can be certain that a D.N [description number, ed.] of a TM exists for implementing the UPM, even if we cannot easily generate it herein. The existence of a D.N for this purpose is sufficient for almost any demonstrations that an actual description would be useful for and thus we do not attempt to generate an actual description.

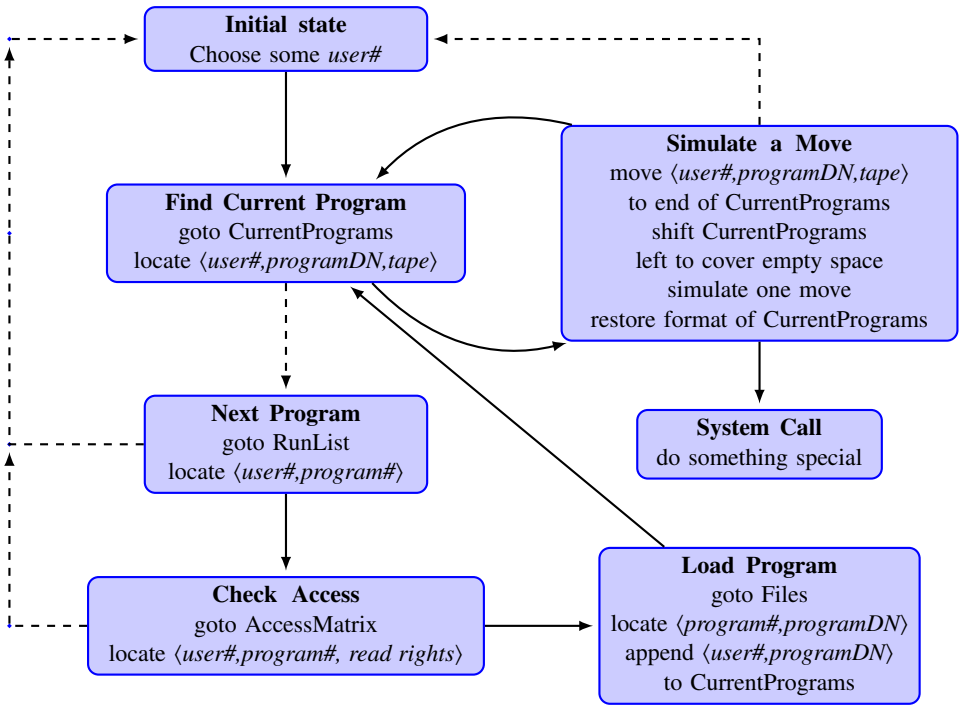
We cannot agree with the above statements. First of all, the property of 'being a viral string' for a machine M is very much dependent on the exact description of the machine M. Now suppose we have two different implementations U and U' of the UPM model. As we have seen in Section 2.3.2, we lack a suitable notion of machine equivalence that would guarantee that U and U' are virally equivalent. Therefore, without specifying the exact implementation of the UPM model as a machine U, we have no basis whatsoever to call any string a virus.

Secondly, Cohen states that the most "important aspect" of a model of a "computer with an operating system" is that its basic operations are computable; i.e. such a model could be implemented on Turing Machines in principle. This would imply that we can model properties of computer viruses in the UPM

[7] For an introduction of the definitions of the Universal Turing Machine in literature see Section 1.5.

Figure 2.3 – Sketch of UPM states and operations

In this figure we refer to marked locations on the tape of a UPM by their names: 'CurrentPrograms', 'RunList', 'AccessMatrix'. The UPM (somehow) remembers variables: *user#* (user number), *programDN* (program), *tape* (the content of a simulated tape), *program#* (program number) and *read rights*. Dashed arrows are traversed when an operation fails, black arrows are traversed when the operations succeed. The arrow to 'System Call' is only traversed under special circumstances.



model without referring to the formal definition of viruses (in the Cohen machine model). This defeats the purpose of formally defining computer viruses in the first place.

We claimed that the UPM as a model is underspecified, let us explain this claim. Cohen defines the UPM to be a Cohen Machine that performs a number of high level 'operations' on a tape that is organised in a special way. Figure 2.3 sketches the working of such a machine. We are not easily convinced that a description of these operations is sufficient to guarantee that there is an implementation that respects this organisation of the tape, and that moreover allows us to apply the definition of viral sets.

As we have seen in Section 1.5, the implementation of a Universal Turing Machine is non trivial, and the implementation of a UPM is arguably more complex. Many universal machine implementations in the literature exhibit a kind of interleaving of tapes to accomplish the simulation of another machine (see Section 1.5). As a result, programs are not to be located as contiguous strings on the tape. We are therefore not convinced, without proof or demonstration, that a UPM can be implemented in such a way that 'programs' are represented as contiguous sequences on the tape.

For Cohen's definition of viruses, the interpretation of the 'initial machine state' is particularly important, see our discussion in Section 2.3.2 and Section 2.3.3. If it is not unique, we notice that each program, program number, user number etc. is a virus, because these strings are copied around. If, on the other hand, the initial state is unique, then it has a special significance; we saw in the proof of Lemma 2.12 that only the initial segment of the tape is considered to be a program. Cohen proposes that the first part of the tape a UPM contains the 'Files' part (see also Figure 2.3); this has the effect that the first square contains a special symbol. This special symbol cannot be part of the strings that the UPM considers 'programs'. Such 'programs' can therefore never be viruses, if the UPM does not return to its start state.

We feel the UPM model returns our attention to a central question: "why define viruses for Turing Machines?". If the operations of the Universal Protection Machine were actually implemented as a transition function, the resulting machine should certainly not be understood as a modern Turing Machine (that computes functions on input). Neither should this machine be understood as a universal machine capable of simulating several Turing Machines at once. It would perhaps be a model of a computer that operates on a (tape) memory. The similarity with Turing Machines becomes a disadvantage, because the operations that it can perform on its memory (the tape) are very basic which means the model becomes very complex. Even so, as we have shown in this section, Cohen's definition of viruses does not apply to this model per se.

2.4.3 Universal Persistent Turing Machine

In [33], Hao et al. propose to model computer viruses using the concept of a Universal Persistent Turing Machine (UPTM). We introduced Persistent Turing Machines in Section 1.3 and their universal counterpart in Section 1.5.4. The following quote from [33] illustrates the authors' rationale for choosing UPTMs over (Universal) TMs:

> Modern general-purpose computers are implemented in accordance with the idea of Universal TMs. However, as the theoretical basis for computers, Universal TMs are unable to describe a continuous computational process of a computer during which unpredictable interaction events sequentially happen.

Their definition:

Definition 2.13 (UPTM viruses)**.** Let U be a UPTM with an encoding function η. (The status of function η is discussed below). Then a set $V \subseteq \Sigma_U^*$ is viral for U if

$$
\begin{aligned}
v \in V \iff & \exists \text{ PTM } M,\ \exists w, w', i, o \in \Sigma_M^* \text{ such that} \\
& \wedge\ v = \eta(M) \\
& \wedge\ (v, \eta(w)) \xrightarrow[U]{\eta(i)/o} (v, \eta(w')) \\
& \wedge\ w, w' \in reach(M) \\
& \wedge\ o \in V
\end{aligned}
$$

The above condition should be read as: "a set V is viral if each program v in V, if it runs on the universal machine U, eventually produces another program $o \in V$". In essence, this captures 'viruses' using the same closure property on programs, as we saw in Cohen's work (Section 2.3.2).

Let us discuss the encoding function η. According to Hao, Yin and Zhang, η is a one-one function that maps the transition function and strings of a machine M to the alphabet Σ_U of the universal machine. The encoding function η is defined by:

$$\begin{aligned} \eta &= \eta_{machine} \cup \eta_{tape} \\ \eta_{machine} &: M \to \Sigma_U^* \\ \eta_{tape} &: \Sigma_M^* \to \Sigma_U^* \end{aligned}$$

The above specification of η is particularly ambiguous. Such a function η could only encode M, or perhaps a very specific (finite) set of PTM machines. In other words U is not universal.[8] Moreover, if η is one-one then it is injective and no output of U while simulating M will be the encoding of a machine (because the output of M is in Σ_M^*). In particular, this definition tells us that V must be empty, i.e. viruses do not exist.

It is not unreasonable to define η as an injective function, as does Turing (see Definition 1.54). We might even feel that 'injective' is part of the definition of the word 'encoding'. Suppose, however, that we have an encoding that is not injective, in which case it would in principle be possible that $\eta^{-1}(g_p(x))$ is (the transition function of) a PTM. Now if a simulated machine M wants to output a program, it has to output a string t in Σ_M^* exactly so that $\eta^{-1}(\eta(t))$ is not uniquely defined. That means that the simulated machine needs to be aware, in some sense, of the encoding that is employed; when a different (non-injective) encoding function is employed (and possibly an altered machine U), it probably does not output the same, if any, program.

One could argue, that this models the fact that some data can be a program (or virus) for one computer, whilst not being a program (or virus) for another. This, however, is a fallacy - the machine M can be simulated by *any* machine (with appropriate encoding) that is universal. If we interpret the output of the simulation of M in a way that is particular to a specific universal machine implementation U, the interpretation of M changes with it. This is orthogonal to the concept of universal machines, where the fact that we simulate M means that there is a one-one correlation between the 'behaviour' of M when simulated and is its behaviour when running machine M itself.

We conclude that, whether the encoding function η is injective or not, we should not interpret the (encoded) output of simulated machines as programs. The proposed definition of viruses in [33] is no more plausible than that of Definition 2.8.

[8] In all fairness to Hao, Yin and Zhang, this view of the encoding function can be traced back to [26].

Chapter 3

Recursion Machines

In this chapter we will propose a new model of computers: machines which we dub 'Recursion Machines'. This will be a model based on the theory of recursive functions. We were inspired by the work of Bonfante, Kaczmarek and Marion in [30, 32]. These authors define viruses using a kind of recursion theory of enumerably infinite sets. What makes the work of Bonfante et al. appealing is that the domain of discourse is not the natural numbers as usual, but a rather more liberal domain $\mathcal{D}$. In standard recursion theory we think of data and programs as natural numbers: both are 'encoded' as numbers. If the domain would be more liberal, for example the untyped lambda calculus or a full set of strings Σ^*, the connection between a program and its encoding becomes closer; they may even be identical.

Bonfante et al. provide this connection by defining 'acceptable programming systems' [30, 32, 39]. The properties that characterise an 'acceptable programming system' seem to mirror the well-known sufficient conditions of an acceptable system of indices: enumeration and parametrization [3, 6]. As proved by Rogers [15] and subsequent authors, most results of recursion theory will hold for any acceptable system of indices; notably the second recursion theorem, fixed point theorem, etc.

We feel that parts of Bonfante cum suis' definition of 'acceptable programming systems' are unclear and that the connection between 'acceptable programming systems' and 'acceptable system of indices' is not convincing. We propose a definition that is more directly linked to the definition of 'acceptable system of indices'.

On the one hand, this is a straightforward way of lifting results in the theory of computable functions on natural numbers to a theory of computable functions on arbitrary enumerably infinite sets. On the other hand, we can view the connection between the two as a 'minimal condition'. We assume that for any reasonable programming system we have a background notion of computability or recursivity. If this background notion is not explicit, the link with acceptable systems of indices provides the minimal conditions that enable us to act as if it were explicit.

The theory of acceptable programming systems that we present can be extended to define 'Recursion Machines', we will do so by adding a notion of 'sequential execution of programs' in Section 3.2. The connection with recursive function theory also allows us to express Adleman's theory of viral functions in the setting of Recursion Machines [17]; Adleman's theory is the most convincing theory of viruses we have encountered. In Section 3.3.3 we show that the notion of 'Recursion Machine' is a fruitful addition to the theory of viral functions. The chapter is concluded by discussing the possibilities of further extensions of Recursion Machines.

3.1 Acceptable Programming Systems

We will start by providing the definition of acceptable systems of indices. On top of these we define 'acceptable programming systems'. Throughout this chapter we will refer to the standard system of indices, using Gödel numbering of Turing Machines, as the family of maps ϕ.

Definition 3.1 (Acceptable system of indices [6, II.5.1/2])**.** We call a system of indices any family ρ of maps ρ^n from ω to the set of n-ary partial recursive functions (by ρ^n_e we will indicate the partial recursive function corresponding to the index e). It is acceptable if, for every n, there are total recursive functions f and g such that:

$$\rho^n_e \simeq \phi^n_{f(e)} \quad \text{and} \quad \phi^e_n \simeq \rho^n_{g(e)}$$

A standard result in recursion theory is the fact that acceptable systems of indices are characterised by two properties: enumeration and parametrization [3, Exercise 2.10].

Proposition 3.2. *A system of indices ψ is acceptable if and only if it satisfies the following properties:*

- Enumeration [6, II.5.1]*: for every n there is a number a such that*

$$\psi^{n+1}_a(e, x_1, \ldots, x_n) \simeq \psi^n_e(x_1, \ldots, x_n)$$

- Parametrization [6, II.5.1]*: for every m, n there is a total recursive function s^m_n such that*

$$\psi^n_{s^m_n(e,x_1,\ldots,x_m)}(y_1, \ldots, y_n) \simeq \psi^{m+n}_e(x_1, \ldots, x_m, y_1, \ldots, y_n)$$

Proof. A proof can be found in [6, II.5.3]. □

We define 'acceptable programming systems' with respect to a domain $\mathcal{D}$ on the basis of 'acceptable systems of indices'. The set $\mathcal{D}$ will have to have some structure that resembles the natural numbers. This is expressed by a bijection between ω and $\mathcal{D}$ that is in some sense 'effectively computable'.

Definition 3.3 (Acceptable programming system (1))**.** A *programming system* is a family ψ of maps ψ^n from $\mathcal{D}$ to the set of n-ary functions on $\mathcal{D}$. We say that ψ is *acceptable* if there is an effectively computable bijection $h : \omega \leftrightarrow \mathcal{D}$ such that

if for all natural numbers n, e: $\quad \rho^n_e \simeq \lambda x_1 \ldots x_n \,.\, h^{-1}(\psi^n_{h(e)}(h(x_1), \ldots, h(x_n)))$

then ρ is an acceptable system of indices

In essence, the bijection h gives us a one-one correspondence between all partial recursive functions (over ω) and functions over $\mathcal{D}$ (as ψ indexed by $\mathcal{D}$). This correspondence justifies that we lift the concept of recursivity to $\mathcal{D}$.

Definition 3.4 (Partial recursive function)**.** Let ψ be an acceptable programming system. Then we call a partial (or total) function $f : \mathcal{D}^n \to \mathcal{D}$ 'partial (or total) recursive' if there is an $d \in \mathcal{D}$ such that $f \simeq \psi^n_d$.

We can also look at 'acceptable programming systems' this way: the background notion of effective computability expressed by a programming system ψ corresponds exactly with the standard notion of effective computability if ψ is acceptable.

We shall now prove two properties of acceptable programming systems that are essential to lift results from recursion theory to acceptable programming systems. From the following theorem we can prove for example Kleene's second recursion theorem and the existence of a composition function for $\mathcal{D}$ (by copying the corresponding proofs from [3] and substituting each occurrence of the natural numbers by $\mathcal{D}$).

Theorem 3.5. *Let ψ be an acceptable programming system with respect to domain $\mathcal{D}$ (Definition 3.3). Then ψ satisfies the following properties:*

- Enumeration property for $\mathcal{D}$*. For every $n < \omega$ there is an element $a \in \mathcal{D}$ such that:*

$$\psi^{n+1}_a(e, x_1, \ldots, x_n) \simeq \psi^n_e(x_1, \ldots, x_n)$$

- Parametrization property for $\mathcal{D}$*. For every $m, n < \omega$ there is a total recursive function s^m_n such that:*

$$\psi^n_{s^m_n(e,x_1,\ldots,x_m)}(y_1, \ldots, y_n) \simeq \psi^{m+n}_e(x_1, \ldots, x_m, y_1, \ldots, y_n)$$

Proof. We prove the enumeration property. Definition 3.3 guarantees that we have an acceptable system of indices ρ and bijection $h : \omega \leftrightarrow \mathcal{D}$. Let g be the inverse of h. Proposition 3.2 ensures that ρ satisfies enumeration and parametrization (for ω). Essentially, we only have to go back and forth between $\mathcal{D}$ and ω to prove the theorem:

$$\begin{aligned}\psi_e^n(x_1,\ldots,x_n) &\simeq h\left(\rho^n_{g(e)}\left(g(x_1),\ldots,g(x_n)\right)\right)\\ &\simeq h\left(\rho^{n+1}_{a}\left(g(e),g(x_1),\ldots,g(x_n)\right)\right)\\ &\simeq g\left(h\left(\psi^{n+1}_{h(a)}\left(h(g(e)),h(g(x_1)),\ldots,h(g(x_n))\right)\right)\right)\\ &\simeq \psi^{n+1}_{h(a)}(e,x_1,\ldots,x_n)\end{aligned}$$

Since $h(a) \in \mathcal{D}$, the enumeration property is satisfied by the above equation.

The proof of the parametrization property is analogous. Notice that each function s^m_n that we find for $\mathcal{D}$ is 'total recursive' by virtue of our correspondence of enumerations. □

We conclude this section with another positive result.

Theorem 3.6 (Existence of acceptable programming systems)**.** *Let $\mathcal{D}$ be any enumerably infinite set. Then there exists an acceptable programming system ψ for this domain.*

Proof. Since $\mathcal{D}$ is enumerable, there is an effectively computable bijection $h : \omega \leftrightarrow \mathcal{D}$. Let g be the inverse of h. Let ϕ be the standard system of indices. For any positive natural number n define mapping $\psi^n_e \simeq \lambda x_1 \ldots x_n \,.\, h\left(\phi^n_{g(e)}\left(g(x_1),\ldots,g(x_n)\right)\right)$. Then h is the required bijection of Definition 3.3. □

3.2 Recursion Machines

We will now extend the concept of an 'acceptable programming system' ψ. We will exploit the double interpretation of elements in $\mathcal{D}$ that ψ provides: 'data' or 'program'. In its capacity of 'data', a single element d in $\mathcal{D}$ can be understood to describe the current 'state' of some (as yet unspecified) machine R. At the same time, a 'program' p in $\mathcal{D}$ could be 'executed' by R, in the sense that if R 'executes' p then R by some means computes ψ^1_p applied to the current state of R. The question arises: how did we (or R) know to 'run' p and not some other program; and how do we know what program to run next?

As was the case for Maurer computers (see Section 2.1), we lack a unit of 'control' that uniquely determines the next program to run (based on the current state description). We postulate such a (partial) recursive control function $\tau : \mathcal{D} \to \mathcal{D}$. We can now define a 'Recursion Machine'.

Definition 3.7 (Recursion Machine)**.** Let ψ be an acceptable programming language on a domain $\mathcal{D}$, and τ be a (partial) recursive function $\tau : \mathcal{D} \to \mathcal{D}$. Then the structure $R = (\mathcal{D}, \psi, \tau)$ is a Recursion Machine (RM). The domain $\mathcal{D}$ is the set of all states of R. We define 'moves' as a (partial) recursive function $\hookrightarrow_R$ on the set of states ($\mathcal{D}$) by:

$$x \hookrightarrow_R y \iff y = \psi^1_{\tau(x)}(x)$$

Remark 3.8. The concept of a 'Recursion Machine' closely corresponds to a high-level interpretation of Maurer computers (see Section 2.1). This is best exemplified by carefully choosing $\mathcal{D}$. Let us choose $R = (\mathcal{D}, \psi, \tau)$ such that $\mathcal{D} = \mathbb{F}_B$ for some finite base set B that includes an element representing blank ($\square$). Now we define a Maurer computer $M = (M, B, \mathcal{S}, \mathcal{I})$ where

$$\begin{aligned}M &= \omega\\ \mathcal{S} &= \mathcal{D} = \mathbb{F}_B = \{t : \omega \to B \mid \exists n < \omega\ \forall m > n \ :\ t(m) = \square\}\\ \mathcal{I} &= \left\{\psi^1_p : \mathcal{S} \to \mathcal{S} \mid p \in dom(\psi)\right\}\\ C &= \left\{S \mapsto \psi^1_{\tau(S)} \mid S \in \mathcal{S}\right\}\end{aligned}$$

We notice that by choosing $\mathcal{S} = \mathbb{F}_B$ Maurer's axioms hold. The reader may verify that Maurer moves for M correspond to Recursion Machine moves for R, i.e. $\hookrightarrow_{cc} \simeq \hookrightarrow_R$.

We shall now show that we can express computation of functions on the naturals using Recursion Machines and their 'move'-function '$\hookrightarrow_R$'. Given the fact that we use an acceptable programming system, this should not be surprising.

Theorem 3.9. *For any partial recursive function $f : \omega \to \omega$ and enumerably infinite domain $\mathcal{D}$ we can construct a Recursion Machine $R = (\mathcal{D}, \psi, \tau)$ such that there is a bijection $f \leftrightarrow \hookrightarrow_R$.*

Proof. By Theorem 3.6 we have an acceptable system ψ (with respect to the standard system ϕ) and bijection $h : \omega \leftrightarrow \mathcal{D}$. Let g be the inverse of h. Let $e \in \omega$ be such that $\phi^1_e \simeq f$. Choose $\tau(x) = h(e)$. Let $f(a) = b$, then:

$$\begin{aligned}\psi^1_{\tau(h(a))}(h(a)) &\simeq \psi^1_{h(e)}(h(a))\\ &\simeq h\left(\phi^1_{g(h(e))}(g(h(a)))\right)\\ &\simeq h\left(\phi^1_e(a)\right)\\ &\simeq h(f(a)) = h(b).\end{aligned}$$

Then $h(a) \hookrightarrow_R h(b)$. Since h is a bijection, the converse holds as well. The bijection between f and $\hookrightarrow_R$ is given by $(a, b) \mapsto (h(a), h(b))$. □

Remark 3.10. If we weaken Theorem 3.9 to domain ω, then we can choose the identity function as bijection in the proof. This leads to a corollary: for domain ω and any partial recursive function $f : \omega \to \omega$ there is a Recursion Machine R such that $f \simeq \hookrightarrow_R$. Moreover this R has the standard system of indices ϕ as acceptable programming system. For any other bijection (i.e. permutation of) $\omega \to \omega$, Theorem 3.6 gives us an acceptable programming system ψ that is at the same time that is a non standard acceptable system of indices.

Remark 3.11. For any R such that τ is the identity function we have $x \hookrightarrow_R y \iff \psi^1_x(x) = y$. This is the simplest form of self reference of code, where program x has access to its own representation (but nothing more).

The intention of defining a control function τ was to execute programs sequentially. After each execution we get a machine state in $\mathcal{D}$, which gives rise to a sequence of states.

Definition 3.12 (Recursion machine state sequences)**.** The set of all state sequences of a Recursion Machine R is the set $\mathbb{P}_R$ of all $s : \alpha \to \mathcal{D}$ such that

$$\begin{aligned}&\wedge \quad 0 < \alpha \leq \omega && (s \text{ is a non empty sequence, of finite or infinite length})\\ &\wedge \quad (\forall\, i < \alpha - 1)\,(s(i) \hookrightarrow_R s(i+1)) && (\text{subsequent elements are } \hookrightarrow_R \text{ connected})\\ &\wedge \quad (\nexists d \in \mathcal{D})\,(s(\alpha - 1) \hookrightarrow_R d) && (s \text{ has maximal length})\end{aligned}$$

3.2.1 Modeling an Operating System

We may wonder if we can run programs ad infinitum, given a particular starting state $q \in \mathcal{D}$. This amounts to whether the unique sequence $s \in \mathbb{P}_R$ such that $s(0) = q$ has an infinite domain. This depends, in part, on the control function τ. It is a necessary condition that τ be defined on the range of s, i.e:

$$s : \omega \to \mathcal{D} \implies \tau \text{ is defined on the range of } s$$

We can satisfy this condition by defining τ as a total function on $\mathcal{D}$. To do that, we first need to define 'lists' and 'list encodings'.

Definition 3.13 (List encoding)**.** A 'list encoding' is a total recursive injective function $[\cdot]$ such that:

$$[\cdot] : \left(\bigcup_{0<n<\omega} \mathcal{D}^n\right) \to \mathcal{D}$$

A 'one-one list encoding' is a surjective 'list encoding'. Let the corresponding projections be defined as partial recursive functions π_n such that for all $1 \leq n, m < \omega$

$$\pi_n : \mathcal{D} \to \mathcal{D}$$
$$\pi_n([x_1, \ldots, x_m]) = \begin{cases} x_n & \text{if } 1 \leq n \leq m \\ \uparrow & \text{otherwise} \end{cases}$$

In general, if an element $p = [x_1, \ldots, x_n]$ (for some positive number n), we can call p a 'list' (of $x_1, \ldots, x_n$). If $[\cdot]$ is one-one, than every element of $\mathcal{D}$ encodes a list.

Remark 3.14. The functions $[\cdot]$ and π_n can only be said to be recursive with respect to an acceptable programming system ψ. In a strict sense, different acceptable programming systems give rise to different functions $[\cdot]$ and π_n.

Notice that if $[\cdot]$ is one-one, the projection to the first element of a list always exists (and is a total recursive function). In this case $\tau = \pi_1$ is a total function on $\mathcal{D}$.

From now on we will assume that the control function for any $\mathcal{D}$ is defined as $\tau = \pi_1$. This can be seen as coding the control in the domain. Instead of choosing a function τ we can choose a program as the first element of any list.

Definition 3.15 (Recursion Machine with Operating System)**.** For any Recursion Machine $R = (\mathcal{D}, \psi, \pi_1)$, list encoding $[\cdot]$ and program $\text{os} \in \mathcal{D}$, a *Recursion Machine with Operating System (OS)* is defined by $O = (\mathcal{D}, \psi, \text{os})$. The set of states of O is $\mathcal{D}$. The moves of O are the moves of R. We restrict the set of state sequences of O to those sequences s such that $s(0) = [\text{os}, \ldots]$. I.e.:

$$\hookrightarrow_O \simeq \hookrightarrow_R$$
$$\mathbb{P}_O = \{s \in \mathbb{P}_R \mid \pi_1(s(0)) = \text{os}\}$$

Remark 3.16. There is a strong parallel between fixing the control function τ to respect a program 'os' in this framework, and fixing the (Maurer) control function C to respect the decoding of instruction in $NI \subset M$ for the Maurer computer framework. The next program to run may be located in a fixed 'location' of the current state. Suppose we have a Recursion Machine with OS $O = (\mathcal{D}, \psi, \text{os})$, then we can adapt the Maurer computer in Remark 3.8 by defining:

$$C(S) = DEC(S \restriction NI)$$
$$DEC = \left\{\left(S \restriction NI, \psi^1_{\text{os}}\right) \mid S \restriction NI = head(\text{os}),\ S \in \mathcal{S}\right\}$$

The reader may verify that in this setting, Maurer moves still correspond to Recursion Machine moves.

3.2.2 Memory access

We can now model memory access. We can think of the state of a Recursion Machine (with OS) as the memory of a 'computer' (in a broad sense). To that end we model the distinguishability of separate programs (and data in general) in memory.

For example, we can view the memory as a list of data $[x_1, \ldots, x_n]$. On top of that we can, for example, model an operating system that employs a program counter t (without specifying how t maps to t') that gives programs full access to the memory (except for the operating system and the program counter). Let $O = (\mathcal{D}, \psi, \text{os})$ be a Recursion Machine with OS such that:

$$\psi^1_{\text{os}}([\text{os}, t, x_1, \ldots, x_n]) = [\text{os}, t']; \psi^1_{x_t}([x_1, \ldots, x_n])$$

We can see that $\psi_{\text{os}}([X, t, x_1, \ldots, x_n])$ is only well defined if $0 < t \leq n$. Now if the operating system 'executes' a program x_t, this program has access to the full list of programs $[x_1, \ldots, x_n]$ and it may return a shorter or longer list, or not return a list at all. (Note that if $[\cdot]$ is one-one we can at least be sure that $\psi_{x_t}(y)$ is a list). The program x_t can change the next scheduled program $x_{t'}$ or make it irretrievable, in which case the

systems breaks down (no next RM state). We could try to restrict the list of programs $[\mathrm{os}, t, x_1, \ldots, x_n]$ we start out with: ensure that each x_i is well behaved, in a way that each future program p that is executed (and created by a program that was executed earlier) is also well behaved. How do we determine if a program is well behaved in such a far-reaching way? Can we even express this without taking into account other (possible) data in memory?

We are already close to the issue of viruses. We would call a program 'malicious', for example, if it is not well behaved in this sense.

3.3 Recursion Machine viruses

3.3.1 Viruses as programs

We would like to briefly mention the work by Bonfante et al. [30, 32]. According to these authors "viruses may be defined by their ability of self-reproduction".

> The propagation mechanism describes how a virus infects and replicates with possible mutations. The propagation may end when a virus decides to do something else. We represent the propagation mechanism as a computable function $\mathcal{B}$, which is a vector that infects, carries and transmits a viral code. For this, assume that v is a program of a virus and that p is a program, or a datum.
>
> We are thinking of $\mathcal{B}(v, p)$ as a program which is the infected form of p by v. So, the propagation function transforms a program p into another program. When one runs the program $\mathcal{B}(v, p)$, the virus v is executed and it possibly copies itself by selecting some targets.

This is then formalised as:

Definition 3.17 (Computer virus)**.** Assume that $\mathcal{B} : \mathcal{D} \times \mathcal{D} \to \mathcal{D}$ is an effectively computable function, called the 'propagation function'. Assume that $\langle \cdot, \cdot \rangle : \mathcal{D} \times \mathcal{D} \to \mathcal{D}$ is a pairing function. Let ψ be an acceptable programming system. A virus with respect to $\mathcal{B}$ is a program v such that for each $p, x \in \mathcal{D}$

$$\psi^1_v(\langle p, x \rangle) = \psi^1_{\mathcal{B}(v,p)}(x)$$

This characterisation of viruses is appealing at first sight; Ockham's razor applies here. Furthermore it is intuitive to think of viruses as programs. We feel however, that it is not sufficiently clear from [30, 32] in which way this definition truly captures the notion of viruses. If further research unequivocally demonstrates this fact, then this characterisation can be expressed within the framework of acceptable programming systems (as developed in Section 3.1).

3.3.2 Viruses as functions

We will shortly introduce the reader to the work of Adleman [17].

According to Adleman, we should think of a virus as a map from programs to infected programs. Suppose p' is the infected form of p. An infected program p' can do one of three things:

1. *Injure*: Compute a function $\psi_{p'}$ that is different from the function ψ_p computed by the uninfected program.
2. *Infect*: First compute the original function ψ_p (on some input) and then (if this halts) infect programs.
3. *Imitate*: Neither injure not infect, i.e. $\psi_{p'} \simeq \psi_p$.

We will provide Adleman's formalisation, as translated to our framework of acceptable programming systems. We slightly simplify the definitions of viral function, by discarding the differentiation between infectable data (programs) and non-infectable data (just data) that is present in Adleman's definitions. The simplification is justified by the fact that the differentiation is not used at all [17]. First we need to introduce some notation.

- We can pairwise compare the elements of two lists. If all pairs are equal or h-related, and at least one pair is not equal, then we call the list h-similar. For all $n < \omega$, we say that $[x_1, \ldots, x_n] \overset{h}{\sim} [y_1, \ldots, y_m]$ if $n = m$ and:

$$\wedge\ (\forall\ i, 1 \leq i \leq n)\,(x_i = y_i\ \vee\ h(x_i) \downarrow = y_i)$$
$$\wedge\ (\exists\ i, 1 \leq i \leq n)\,(x_i \neq y_i)$$

- For all $n < \omega$ and $f, g : \mathcal{D} \to \mathcal{D}$ we say that $f(x) \overset{h}{\cong} g(x)$ if

$$\vee\ f(x) \overset{h}{\sim} g(x)$$
$$\vee\ f(x) \simeq g(x)$$

Definition 3.18 (Virus (Adleman))**.** Let ψ be an acceptable programming system. A total recursive function $\mathcal{A} : \mathcal{D} \to \mathcal{D}$ is a *virus* (w.r.t ψ) if one of the following conditions holds. We also refer to viruses as a 'viral functions'.

- $\mathcal{A}$ *injures*: $(\forall\ p, q, x \in \mathcal{D})\left(\psi_{\mathcal{A}(p)}(x) \simeq \psi_{\mathcal{A}(q)}(x)\right)$
- $\mathcal{A}$ *infects or imitates*: $(\forall\ p, x \in \mathcal{D})\left(\psi_p(x) \overset{\mathcal{A}}{\cong} \psi_{\mathcal{A}(p)}(x)\right)$

Based on this definition of viruses, we can express some properties of infected programs. Throughout this section we will silently assume we have chosen an acceptable programming system ψ.

Definition 3.19 (Viral properties (Adleman))**.** Let $\mathcal{A}$ be a viral function. Then for any program $p \in \mathcal{D}$ we call $\mathcal{A}(p)$ a virus. A virus can have several properties. We say that program $\mathcal{A}(p)$ is

- *pathogenic* w.r.t. $\mathcal{A}$ if there is a q such that $\psi_p(q) \overset{\mathcal{A}}{\ncong} \psi_{\mathcal{A}(p)}(q)$
- *contagious* w.r.t $\mathcal{A}$ if there is a q such that $\psi_p(q) \overset{\mathcal{A}}{\sim} \psi_{\mathcal{A}(p)}(q)$

By the above properties of programs infected by a virus, Adleman distinguishes between several kinds of viral functions. Most prominently

- A virus $\mathcal{A}$ is *benign* iff no infected program is pathogenic or contagious (w.r.t. $\mathcal{A}$).
- A virus $\mathcal{A}$ is *malicious* iff there is a contagious and a pathogenic infected program (w.r.t. $\mathcal{A}$).

Adleman then proves two theorems about the detection of viruses. Further definitions and results along this line, for example about 'polymorphic' viruses can be found in [29].

- The set of all viral functions is $\Pi_2$complete.
- There is a malicious virus such that its range is $\Sigma_1$complete.

These are strong undecidability results. However, we should be careful not to jump to conclusions: these results do not show that the question whether a given program is pathogenic and/or infectious w.r.t. a virus is undecidable. The undecidable of virus detection in this sense is not yet proven.

It seems beyond dispute that Adleman's definition of viruses captures the notion of computer virus to a large degree. However, it has has several drawbacks:

- Infected programs can only add or delete programs in a list if the corresponding uninfected program does the same. (This might be remedied by adjusting the definition of $\overset{h}{\sim}$.)
- In practice, a virus is encountered in the form of an infected program and not as the corresponding viral function. Moreover, a single program may be infected w.r.t. more than one viral function.
- We would be tempted to identify a viral function with an infected program that is pathogenic and/or contagious (w.r.t. a viral function). An 'infected program' that is neither pathogenic nor contagious can hardly be called 'infected', as we cannot deduce from it the (presumably malicious) viral function that infected it. (While it may be uncomfortable that a program is padded to twice the original size, as long as it behaves properly and is not infectious, we cannot really be bothered.)

3.3.3 Recursion Machine Viruses

We use Adleman's theory of viruses in our framework of acceptable programming systems, but in a reverse way. Instead of defining a specific viral function, we will define the behaviour of one or more programs that we consider as 'viral'. The (or rather a) corresponding viral function should be deduced from these infected programs.

We now give several examples:

- *Overwriting virus [32].* Let $\psi_v^1([x_1, \ldots, x_n]) = [v, \ldots, v]$ (such that this list of v's has n elements). The program v is pathogenic w.r.t. the viral function $\mathcal{A} : x \mapsto v$.

- *Companion virus [32].* A program that is infected by a 'companion virus' will first execute the viral code and afterwards its original code. Let $\psi_{\text{comp}(x,y)} \simeq \psi_y \circ \psi_x$ (i.e. $\text{comp}(x, y)$ is a function that computes a code of the composition of the functions of x and y). Now let

  $$\psi_v([x_1, \ldots, x_n]) = [\text{comp}(v, x_1), \ldots, \text{comp}(v, x_n)]$$

  Such a program exists as a direct result of the second recursion theorem (which holds because ψ is an acceptable programming system). The program v is pathogenic w.r.t. $\mathcal{A} : x \mapsto \text{comp}(v, a)$. Note that we can easily express such a viral function because we start out by specifying a program that exhibits the behaviour that characterises the viral function.

- *Memory resident virus.* Suppose we have an operating system using a program counter and list of programs. Suppose that in this case, the operating system grants 'full memory access' to the programs that it executes. Such an operating system can be modeled as a Recursion Machine with Operating System defined by $R = (\mathcal{D}, \psi, os)$ where (compare with the example in Section 3.2.2):

  $$\psi_{\text{os}}([\text{os}, t, x_1, \ldots, x_n]) = \psi_{x_t}([\text{os}, t', x_1, \ldots, x_n])(\text{for any positive number } n)$$

  We can now model a form of 'memory resident virus' v w.r.t $\mathcal{A}$ by (for any $0 < n < \omega$):

  $$\psi_v([x_1, \ldots, x_{n+2}]) = ([\text{osv}, x_2, \ldots, x_{n+2}]) \qquad \text{such that}$$
  $$\psi_{\text{osv}}([\text{osv}, t, x_1, \ldots, x_n]) = \psi_{\mathcal{A}(x_t)}([\text{osv}, t', x_1, \ldots, x_n])$$

  The program 'osv' is similar to program 'os' in all respects, except that it executes the infected program $\mathcal{A}(x_t)$ instead of the uninfected program x_t. Whether these infected programs $\mathcal{A}(x_t)$ are themselves pathogenic or contagious, depends on further conditions on $\mathcal{A}$. Notice that the programs x_t are unaltered in the state (memory) of the Recursion Machine. The infected version of x_t will be executed by virtue of the sole infection of the first program in the list (the operating system). Given programs os, osv $\in \mathcal{D}$, a solution to the above equations (such that $\mathcal{A}$ is indeed viral) is:

  $$\mathcal{A} : x \mapsto \begin{cases} \text{osv} & \text{if } x = \text{os} \\ \text{x} & \text{otherwise} \end{cases}$$

The last example shows that we sensibly define classes of viral functions by the partial definition of the (functional) behaviour of an infected program. On the other hand, the more specific we are about the behaviour of the infected program, the less solutions there we will be for the viral function $\mathcal{A}$. There will either be no solutions (due to a contradiction), or infinitely many solutions (due to the padding lemma).

The last examples also shows that in the setting of Recursion Machines with Operating System we can model the impact of viruses on the computer, depending on their implementation. If we consider the same infected program v in the setting of the operating system defined in Section 3.2.2, it will infect the first program x_1 instead of the operating system 'os'.

3.4 Discussion

We have proposed to define computer viruses in the context of Recursion Machines. We have achieved a modeling of computers and computer viruses with the following properties.

- The computer modeling is based on a clear definition of acceptable programming systems. As such we can apply most results from the theory of recursive functions. (Section 3.1).
- We model sequential execution of programs (Section 3.2.1).
- Programs are language (domain) dependent but not machine dependent, as opposed to [5, 18, 33].
- We can apply the theory of viral functions [17, 29].
- We can model interaction of computer implementations and viruses (Section 3.3).
- We can characterise viruses by defining infected programs (Section 3.3).

It must be noted that one desirable feature of the modeling of computer viruses, which we identified in Section 2.1, is still absent: interaction with the environment. Recursion Machines can not yet cope with an infinite stream of inputs. As we have seen in Section 1.3, Persistent Turing Machines are well equipped to deal with infinite input streams. This is due, firstly, to the fact that the work tape of PTMs is a kind of persistent memory. Secondly, the *interpretation* of PTMs is crafted to deal with inputs and output.

Recursion Machines already exhibit a persistent memory, captured by the intermediate machine state after each Recursion Machine 'move'. That leaves us to deal with identifying input and output. A first suggestion is to define Interactive Recursion Machines along the following lines.

Definition 3.20 (Interactive Recursion Machines (IRM) moves)**.** Let $R = (\mathcal{D}, \psi, \tau)$ be a Recursion Machine. Interactive moves are a relation $\hookrightarrow_{IRM}$ on the set of states $\mathcal{D}$ defined by:

$$x \xrightarrow[R]{in/out} y \iff \psi^2_{\tau([x,in])}([x, in]) = [y, out]$$

Remember that a PTM is really a 3-tape Turing Machine interpreted differently. At each move, an external mechanism sets the first tape's content to the input and erases the content of the third (output) tape. If instead of a 3-tape machine, we were to use a 1-tape machine then at each move parts of the content of the tape would be set or erased. We can view this external mechanism as a function on tape contents.

In the context of Recursion Machines, we might therefore view inputs as functions on $\mathcal{D}$:

$$x \xrightarrow[R]{in/out} y \iff \psi_{\tau(in(x))}(in(x)) = [y, out]$$

or even as the programs in $\mathcal{D}$ for these functions:

$$x \xrightarrow[R]{in/out} y \iff \psi_{in}(x) = z \wedge \psi_{\tau(z))}(z) = [y, out]$$

The latter definition lends itself to being interpreted in the following manner: input is modeled by allowing the machine and the environment to alternate moves, where the environment makes an 'input-move' and the machine makes an 'output-move'.

Whether such definitions of interaction have a concrete benefit for the definition and classification of viruses is an open question. In the end such extensions can only be justified by modeling something new: the interaction with a computer user (the use may activate infections or activate defenses) or the interaction with other computers (computers me be infected from outside).

Summary

The purpose of this thesis is to examine the following question: "How can we adequately model computers and computer viruses?". We discuss existing work that uses Turing Machines [1, 2, 4, 28] and propose our own model. The thesis is divided into three clearly separated chapters.

Chapter 1 lays the foundation for our discussing Turing Machines and viruses. We first review the literature on this topic; thereafter we step by step construct a definition of Turing Machines. Reviewing Turing's work [28] brought to light the fact that there is a considerable discrepancy between modern versions of the Turing Machine and Turing's original machine. A thorough investigation of his definitions leads us to an interesting collateral result: we propose a new model of computation called 'Talkative Machines'. This model can unify the original and modern interpretation of Turing Machines.

In Chapter 2 we argue that Turing Machines are not a suitable model of computers if we want to express properties of programs. We particularly discourage the modeling of computer viruses using Turing Machines as a basis. A model of computers that represents 'sequential execution of multiple programs' is called for.

We review Cohen's formalisation of computer viruses in detail [5]. His PhD thesis is given an in-depth review because it was the seminal work that opened up the new research area of 'computer viruses'. We show that Cohen's definition of viruses is based on a non standard definition of Turing Machines. In the setting of standard Turing Machines, his virus definition leads to an equivalence of machines that is incomparable to the standard functional equivalence of machines.

In this thesis we also scrutinize some related work, where viruses are defined using Universal or Persistent Turing Machines [18, 23, 33]. We show that these attempts are susceptible to the same conceptual criticism: these models do not plausibly model modern computers and computer programs.

Chapter 3 is devoted to constructing a new model of computers: Recursion Machines. We will show that this model meets most of the requirements of a computer model that we identified in Chapter 2. The Recursion Machine is defined in the setting or recursion theory. As such, the model is independent of the implementation of computers. Moreover, we can apply the theory of viral functions developed by Adleman [17].

Although the Recursion Machines themselves are implementation-independent, we can use them to model some aspects of computers with their operating systems, for example restricting memory access to programs. Our theory than adds to that of Adleman, as we can express new kinds of viral behaviour that are dependent on the operating system implementation.

This thesis contributes two new models, one of computation: 'Talkative Machines'; and one of computers: 'Recursion Machines'. Their properties and applications can be further researched.

Appendix A

Notational Conventions

In this thesis we will borrow most notation from Devlin [7].

Number, ordinals and tuples

We prefer to use ordinals rather than, say, the naturals ($\mathbb{N}$). Loosely, you can think of $\alpha < \omega$ as $\alpha \in \mathbb{N}$ and of $\alpha = \omega$ as $\alpha =$ size of $\mathbb{N}$. Ordinals are defined as well-ordered sets α such that $\alpha = \{\beta \mid \beta < \alpha\}$. We indiscrimintately write $x \in \alpha$ or $x < \alpha$.

We use ordinals to define three function ranges:

1. A function $f : X \to \alpha$ (with $\alpha < \omega$) has an finite range and maps to finite 'numbers' (ordinals).
2. A function $f : X \to \omega$ has infinite range but maps to finite 'numbers' (ordinals).
3. A function $f : X \to \omega + 1$ has infinite range and maps to either finite 'numbers' or the infinite ordinal ω. This is particularly usefull to define a function that gives the size of a sequence, where a sequence is either finite or enumarably infinite.

For some $n < \omega$ and sets X_i ($i \in n$) we define an n-tuple $(x_0, \ldots, x_n) \in \Pi_{i \in n} X_i$ as:

$$(x_0, \ldots, x_n) = \quad \{\{x_0\}, \{x_0, x_1\}, \ldots, \{x_0, \ldots, x_n\}\} \quad \text{such that } (\forall\, i \leq n)\,(x_i \in X_i)$$

Thus $(x_0, \ldots, x_n)$ is ordered by set inclusion.

Strings and sequences

The set Σ^* is commonly seen as "the set of all finite words over Σ". We read in [4], for example, that a finite string is a 'juxtaposition' of symbols (where 'juxtaposition' means that we write all the symbols right next to each other). Such a characterisation of Σ^* is not precise enough for our purposes.

We choose to view strings a sequences of symbols. Following [7, page 24], "a sequence is a function whose domain is an ordinal". (We will assume throughout this thesis that sequences are total functions.) So a finite sequence t has $dom(t) = \alpha < \omega$. When can then formulate precisely how w interpret Σ^*:

$$t \in \Sigma^* \iff t : \alpha \to \Sigma \text{ such that } \alpha < \omega \text{ and } t \text{ is a total function on } \alpha$$

When we want to express properties of strings (or sequences) is often desirable to be able to 'manipulate' them. Let $t : \alpha \to \Sigma$ and $n < \alpha$, $x \in \Sigma$. We can:

- change (update) a single symbol at a position n :

$$t\,[n \mapsto x] = (t \setminus \{(pos, t(n))\}) \cup \{(n, x)\}$$

- change a symbol x by y throughout a string t:

$$t[x\backslash y] = t - \{(i, x) \mid i < \alpha, t(i) = x\} \cup \{(i, y) \mid i < \alpha, t(i) = x\}$$

- define finite subsequences as (for $t : \alpha \to X$):

$$\begin{aligned} t[start, length] = \quad & t' : length \to X \\ & \text{such that } (\forall\, i < length)\,(t'(i) = t(start + i)) \\ t[start \ldots end] = \quad & t' : (end - start + 1) \to X \\ & \text{such that } (\forall\, i < end - start + 1)\,(t'(i) = t(start + i)) \end{aligned}$$

- express the length of a string. The 'length' of a sequence is expressed by it's domain. As a shorthand, we define $|t| = dom(t)$.

- concatenate sequences using the semicolon ';' as operator. Let $\alpha \in \omega$, and let $x : \alpha \to A$ and $y : \beta \to B$ be sequences. Then the concatenation $x; y$ is a sequence z such that:

$$z : (\alpha + \beta) \to (A \cup B)$$
$$z : i \mapsto \begin{cases} x(i) & \text{if } i < \alpha \\ y(i) & \text{otherwise} \end{cases}$$

- concatenate a sequence with a single symbol, overloading the operator ';'. Let $\alpha \in \omega$, and let $x : \alpha \to A$ be a sequence and $s \in B$ be a symbol. Then the concatenation $x; s$ is a sequence z such that:

$$z : (\alpha + 1) \to (A \cup B)$$
$$z : i \mapsto \begin{cases} s & \text{if } i = \alpha \\ x(i) & \text{otherwise} \end{cases}$$

Appendix B

Long Formulas

Throughout the thesis we write (long) formulas using the style proposed by Lamport in [20].[1] He suggests that we improve readability of formulas by writing conjuncts and disjuncts as lists and eliminating parentheses by indentation. To write a bunch of conjuncts or disjuncts as a list, each element is prepended by the operator $\wedge$ (or $\vee$). For example:

Some property of $A \wedge$ (some property of $B \vee$ something $\vee$ something else)

should be written as:

$\wedge$ Some property of A
$\wedge \vee$some property of B
$\vee$ something
$\vee$ something else

The same style of listing (and prepending the operator) can be applied to elements connected by any associative operator. Lamport notices that this style does not work for non-associatives operators, especially the implication. We propose to extend Lamport's style by writing implication as if ... then .... For instance:

Some property of $A \wedge$ (some property of $B \vee$ something) $\Longrightarrow$ a serious result

would become:

if $\wedge$ Some property of A
$\wedge \vee$ some property of B
$\vee$ something
then a serious result

[1] Lamport is also the creator of LaTeX. We feel obliged to acknowledge that Lamport has already made a sizeble contribution to the styling of mathematical texts.

Bibliography

Books

In chronological order

[1] M Davis. *Computability and unsolvability*. Ed. by J Nash. McGraw-Hill series in information processing and computers. McGraw-Hill, 1958.
See pages: 2, 3, 10, 23, 55.

[2] ML Minsky. *Computation: finite and infinite machines*. Prentice-Hall, 1967.
See pages: 4, 5, 10, 21, 23, 35, 55.

[3] H Rogers Jr. *Theory of recursive functions and effective computability*. New York, USA: McGraw-Hill, 1967.
See pages: 10, 45, 46.

[4] JE Hopcroft and JD Ullman. *Introduction to automata theory, languages, and computation*. Addison Wesley, 1979.
See pages: 3–5, 21, 23, 24, 35, 39, 55, 57.

[5] FB Cohen. "Computer viruses". PhD Thesis. University of Southern California, 1985.
See pages: vi, 31, 33, 35, 36, 38, 40, 53, 55.

[6] P Odifreddi. *Classical recursion theory*. Vol. 125. Studies in logic and the foundations of mathematics. Elsevier North-Holland, 1989.
See pages: 10, 45, 46.

[7] KJ Devlin. *The joy of sets: fundamentals of contemporary set theory*. Springer, 1993.
See pages: 57.

[8] FB Cohen. *A short course on computer viruses*. Wiley, 1994.
See pages: 33, 35.

[9] CH Papadimitriou. *Computational complexity*. Addison-Wesley, 1994.
See pages: 4, 5, 7, 12, 23, 24.

[10] M Sipser. *Introduction to the theory of computation*. International Thomson Publishing, 1996.
See pages: 4, 7, 23.

[11] P Blackburn et al. *Modal logic*. Cambridge University Press, 2001.
See pages: 5.

[12] SB Cooper. *Computability theory*. CRC Press, Inc., 2002.
See pages: 4, 10, 23.

[13] BJ Copeland. *The essential Turing: seminal writings in computing, logic, philosophy, artificial intelligence, and artificial life plus the secrets of enigma*. Oxford University Press, 2004.
See pages: 13, 19, 25, 28, 30, 35.

[14] E Filiol. *Computer viruses: from theory to applications*. Birkhäuser, 2005.
See pages: 33.

Articles

In chronological order

[15] H Rogers Jr. “Gödel Numberings of partial recursive functions”. In: *The Journal of Symbolic Logic* 23.3 (Sept. 1958). Pp. 331–341.
See pages: 45.

[16] FB Cohen. “Computational aspects of computer viruses”. In: *Computers and Security* 8.4 (June 1989). Pp. 325–344.
See pages: vi, 31, 33, 35, 38, 39.

[17] LM Adleman. “An abstract theory of computer viruses”. In: *Advances in Cryptology — CRYPTO' 88*. Vol. 403. Lecture Notes in Computer Science. Springer Berlin/Heidelberg, 1990. Pp. 354–374.
See pages: vi, 45, 50, 53, 55.

[18] K Kauranen and E Mäkinen. “A note on Cohen's formal model for computer viruses”. In: *SIGSAC Review* 8.2 (1990). Pp. 40–43.
See pages: vi, 39, 40, 53, 55.

[19] P van Emde Boas. “Machine models and simulations”. In: *Algorithms and complexity. Handbook of theoretical computer science, volume A*. Ed. by J van Leeuwen. Elsevier Science Publishers, 1990. Pp. 1–66.
See pages: 11.

[20] L Lamport. “How to write a long formula”. In: *Formal Aspects of Computing* 6.5 (1994). Pp. 580–584.
See pages: 36, 59.

[21] H Thimbleby, S Anderson and P Cairns. “A framework for modelling trojans and computer virus infection”. In: *Computer Journal* 41.7 (1998). Pp. 444–458.
See pages: vi, 27, 39, 40.

[22] FB Cohen. “Reply to ‘Comment on “A framework for modelling trojans and computer virus infection”’ by E. Makinen”. In: *The Computer Journal* 44.4 (2001). Pp. 326–327.
See pages: 27.

[23] E Mäkinen. “Comment on ‘A framework for modelling trojans and computer virus infection’”. In: *The Computer Journal* 44.4 (2001). Pp. 321–323.
See pages: 39, 40, 55.

[24] H Thimbleby et al. “Reply to ‘Comment on “A framework for modelling trojans and computer virus infection”’ by E. Makinen”. In: *The Computer Journal* 44.4 (2001). Pp. 324–325.
See pages: 28, 31, 40.

[25] M Davis. “The myth of hypercomputation”. In: *Alan Turing: Life and legacy of a great thinker*. Ed. by C Teuscher. Springer-Verlag, 2004. Pp. 195–211.
See pages: 23.

[26] DQ Goldin et al. “Turing machines, transition systems, and interaction”. In: *Information and Computation* 194.2 (2004). Pp. 101–128.
See pages: 7, 12, 13, 26, 43.

[27] EED Goldin and P Wegner. “Turing's ideas and models of computation”. In: *Alan Turing: Life and legacy of a great thinker*. Ed. by C Teuscher. Springer-Verlag, 2004. Pp. 159–194.
See pages: 28.

[28] A Turing. “On computable numbers, with an application to the Entscheidungsproblem”. In: *The essential Turing*. Ed. by BJ Copeland. This is a faithful reproduction of the original 1936 article. Oxford University Press, 2004. Pp. 58–90.
See pages: v, 1, 2, 5, 7, 13–16, 18, 19, 23, 30, 35, 55.

[29] Z Zuo and M Zhou. “Some further theoretical results about computer viruses”. In: *The Computer Journal* 47.6 (2004). Pp. 627–633.
See pages: 51, 53.

[30] G Bonfante, M Kaczmarek and JY Marion. “Toward an abstract computer virology”. In: *Theoretical Aspects of Computing – ICTAC 2005*. Vol. 3722. Lecture Notes in Computer Science. Springer Berlin/Heidelberg, 2005. Pp. 579–593.
See pages: 45, 50.

[31] PW Dina Goldin. “The Church-Turing thesis: breaking the myth”. In: *New Computational Paradigms*. Vol. 3526. Lecture Notes in Computer Science. Springer Berlin/Heidelberg, 2005. Pp. 152–168.
See pages: 28.

[32] G Bonfante, M Kaczmarek and JY Marion. “On abstract computer virology from a recursion theoretic perspective”. In: *Journal in Computer Virology* 1.3 (2006). Pp. 45–54.
See pages: vi, 45, 50, 52.

[33] J Hao, J Yin and B Zhang. “Modeling viral agents and their dynamics with persistent Turing Machines and cellular automata”. In: *Agent Computing and Multi-Agent Systems*. Vol. 4088. Lecture Notes in Computer Science. Springer Berlin/Heidelberg, 2006. Pp. 690–695.
See pages: vi, 12, 42, 43, 53, 55.

[34] WD Maurer. “A theory of computer instructions”. In: *Science of Computer Programming* 60.3 (2006). Pp. 244–273.
See pages: 5, 28.

[35] JA Bergstra and CA Middelburg. “Maurer computers with single-thread control”. In: *Fundamenta Informaticae* 80.4 (2007). Pp. 333–362.
See pages: 5.

[36] P Cockshott and G Michaelson. “Are there new models of computation? Reply to Wegner and Eberbach”. In: *The Computer Journal* 50.2 (Mar. 2007). P. 232.
See pages: 12, 13, 28.

[37] JA Bergstra and CA Middelburg. “Simulating Turing machines on Maurer machines”. In: *Journal of Applied Logic* 6.1 (2008). Pp. 1–23.
See pages: 28, 30.

[38] DQ Goldin and P Wegner. “The interactive nature of computing: refuting the strong Church–Turing thesis”. In: *Minds and Machines* 18 (Mar. 2008). Pp. 17–38.
See pages: 12, 28.

[39] N Jones. “Computer implementation and applications of Kleene’s Smn and recursive theorems”. In: *Logic from Computer Science: Proceedings of a Workshop Held November 13-17, 1989*. Ed. by Y Moschovakis. Springer-Verlag, 1992.
See pages: 45.

Online Resources

[40] *Eise Eisinga*. Wikipedia, the free encyclopedia.
http://en.wikipedia.org/w/index.php?title=Eise_Eisinga&oldid=209206380
(visited on 05/06/2008).
See pages: 1

[41] *Journal in Computer Virology*.
http://www.springer.com/computer/journal/11416
(visited on 10/06/2008).
See pages: i

[42] *Kleene's T predicate*. Wikipedia, the free encyclopedia.
http://en.wikipedia.org/w/index.php?title=Kleene%27s_T_predicate&oldid=175554885
(visited on 11/03/2008).
See pages: 23

[43] *Press release: mobile phone security market to reach nearly $5bn by 2011, driven by rising security threat to smarter phones*.
http://www.juniperresearch.com/shop/viewpressrelease.php?pr=40
(visited on 10/06/2008).
See pages: i

[44] *Third international workshop on the Theory of Computer Viruses (TCV'08)*.
http://tcv.loria.fr/
(visited on 05/06/2008).
See pages: i, v

[45] *Virus Prevalence - March 2008*. Virus Bulletin.
http://www.virusbtn.com/resources/malwareDirectory/prevalence/index.xml?200803
(visited on 29/05/2008).
See pages: v