The Computational Complexity of Evolving Systems

De Berekeningscomplexiteit van Evoluerende Systemen

(met een samenvatting in het Nederlands)

Proefschrift ter verkrijging van de graad van doctor aan de Universiteit Utrecht op gezag van de Rector Magnificus, Prof. dr. W. H. Gispen, ingevolge het besluit van het College voor Promoties in het openbaar te verdedigen op woensdag 1 februari 2006 des ochtends te 10.30 uur

door

Peter Rudolf Alexander Verbaan

geboren op 21 september 1976, te Tegelen

promotoren: Prof. dr. Jan van Leeuwen
Faculty of Science
Utrecht University

Prof. dr. Jiří Wiedermann
Institute of Computer Science
Academy of Sciences of the Czech Republic

The work in this thesis has been carried out under the auspices of the research school IPA (Institute for Programming research and Algorithmics).

Printed by Ridderprint Offsetdrukkerij B.V., Ridderkerk.

ISBN 90-393-4155-9

Contents

CHAPTER 1

Introduction

Models for computational devices are crucial to many fields of computer science. Without a suitable model, there is no foundation to base one's theories on. Choosing a proper model for a computational phenomenon is not an easy task. If the model does not adequately reflect the phenomenon that one is trying to study, then the results obtained from studying the model may have no counterpart for the actual phenomenon itself. If the model is too detailed, then studying the model may become impossible, because unimportant details get in the way.

In the last century, the Turing machine[42] and its variants like the random access machine (RAM, see Odifreddi[35]) became the standard model for computational devices. The Church–Turing thesis[24], which states that every process that can naturally be called an effective procedure or algorithm can be performed by a Turing machine, was generally accepted. In recent years however, as computers became increasingly more powerful, scientists from different corners of the world came to doubt the validity of the Church–Turing thesis, at least for the networks of computing systems emerging in the real world. In particular, contrary to e.g. Minsky's[33] observation for Turing machines, machines can no longer be viewed as "closed" systems. In a stimulating paper, Wegner[45] argued that the interaction of machines with their environment leads to a wider notion of computation, not covered in his view by the notion of algorithm. Van Leeuwen and Wiedermann[27] made this more precise and argued that on the assumption that modern computing systems are always on, interact, learn and evolve, a more powerful theory of computation than Turing's can indeed result. This shows that the Church–Turing thesis in itself can remain valid, but that new systems have come into play that are not covered by it. Stepney et al.[40] identified six elements of classical machine models that one should modify in view of the modern developments, among them the algorithmical paradigm, which states that systems do not adapt and that all computations are deterministic and finite. Several models are known to possess

super-recursive power, e.g. in the theory of real computation and in hybrid neural networks. In this thesis we focus on the computational aspects of system evolution.

Evolving Interactive Systems

In response to the realization that Turing machines were not an adequate model for contemporary computing systems, several new models have been proposed that capture the computational properties of these systems more accurately. Among those were the evolving interactive systems.

Interactive systems are systems that interact with their environment. While the aspect of interactivity warrants an entire study in itself, we view interactivity only from the viewpoint of the system. The environment behaves in an unpredictable and uncontrollable fashion, and the system merely responds to its environment. In this sense, perhaps it would be better to speak of reactive systems. The aspect of interactivity is modeled by using input (and output) ports. The idea is that time is divided into discrete intervals or time-frames. Every time-frame, a symbol appears at the input port. These symbols originate from the environment that the system is in. A special symbol that carries no information is introduced for time-frames during which there is no action from the environment. This way, the system can respond to a symbol from the environment in the next time-frame. In theory, the environment can adapt its interactions with the system based on the actions of the system in the previous time-frames.

An interesting consequence of allowing computations to be interactive is that it does not make sense to speak about termination of computations, since a computation can be continued by the environment at any point. Thus, the models need to cope with inputs that can be extended to infinite lengths. We use sequences of inputs to deal with these extensible computations.

Evolving systems are systems that change over time. In the context of this thesis, it is assumed that we have no control over these changes, i.e., the changes are governed by outside interferences. While a system can be anything, we deal mainly with computational devices. Examples of evolving systems are seen in many places, from computers that get component upgrades over the course of their existence and networked environment where nodes get added and removed dynamically, to DNA strings or communities of organisms interacting with their environments. This evolving aspect is modeled in two ways. A rather direct approach involves using *sequences* of systems to model an evolving system. All systems in the sequence are static, i.e., they do not evolve. Each system in the sequence then represents a "snapshot" of the evolving system, taken at a specific time-frame. The sequence consists of all these snapshots, ordered by time-frame. Since we have no way to control the changes of an evolving system, we have to deal with sequences that have no uniform way to describe the elements in the sequence other than listing them all in order. Such a sequence is a so-called *non-uniform* sequence. A more traditional approach uses *advice functions* to model evolving systems. This is a concept that was introduced by Karp and Lipton[22]. An advice function can be used to graft an element of evolution onto an otherwise static system. The idea behind an advice function is that for a computational device to perform its computations, it might require extra information from the "outside" that depends only on the lengths of

the inputs it receives. Thus, given an input, the device can ask for advice related to the input length. Since the advice can be different for every input length, the aggregation of device and advice function can be seen as an evolving system. The advice functions that we use are generally non-uniform, i.e., there is no algorithm that generates the advices, so the only way to describe an advice function may be to have a full table mapping each input length to a string of symbols. Note that Karp and Lipton only dealt with classical computations, while we apply the concept to interactive computing systems.

Evolving interactive systems perform computations, i.e., they process input strings and produce output strings. Functions that map inputs to outputs are called *translations*. We develop the theory of evolving interactive systems and the translations they compute. We will show that the fundamental results of theories based on classical models have their counterparts in the theory of evolving interactive systems.

Advice functions and sequences are actually two sides of the same coin. It can be useful to distinguish between the two view-points. This way, we can easily switch view-points and use different characterizations of the same problem to gain insights. On the other hand, it can be useful to acknowledge the similarities between the two approaches and unify the theories into one framework. Thus, we explore several implementations of evolving interactive systems in detail as well as design a more abstract framework for any kind of evolving system.

An important facet of computations is the use of resources, i.e., the time needed to execute algorithms, or the memory used to store temporary data. The study of resource-bounded computations is the basis of (computational) complexity theory. In this thesis we study the complexity of evolving interactive systems. In addition to the classical resources, the nature of the changes of evolving systems imposes extra bounds on the computations. This has already been studied to some extent by Karp and Lipton[22], although in different contexts. We place the theory in the context of evolving systems and establish a whole new complexity theory for evolving interactive systems.

The translations that are computed by evolving interactive systems are divided into classes based upon the resources that the systems use to compute them. We show that these resource bounds impose a complexity hierarchy on the classes of translation, i.e., for pairs of functions f and g, with f smaller than g in a suitable sense, there are translations that can be computed by evolving interactive systems using an amount of resources bounded by g, but not by evolving interactive systems using an amount of resources bounded by f. The functions f and g depend on the resource. For the time resource, the result holds if $f \log f \in o(g)$, for the memory resource, the result holds if $f \in o(g)$, for the evolutionary resource, the result holds if $f(n) < g(n)$ for infinitely many integers n. In this thesis, several other facets of complexity theory will be explored.

Prior Work

Non-uniform complexity, especially the theory of advice functions in relation to concrete computational models, has been studied previously, among others, by Karp and Lipton[22, 23], Ibarra and Ravikumar[21], Hermo and Mayordomo[18]

and Damm and Holzer[10]. The aspect of infinity has been studied in the field of ω-computing by e.g. Cohen and Gold[8], Landweber[26], Staiger[39] and Thomas[41]. The aspect of interactivity, at least our approach to it, has been studied, among others, by Minsky[33], Goldin[13], Goldin and Wegner[14], Kosub[25], Milner[32], Van Leeuwen and Wiedermann[27, 29], Prasse and Rittge[38] and Wegner[45, 46, 47]. Wegner[47] argues that the notion of algorithm no longer adequately describes the behavior of (interactive) computing systems. Burgin[5] gives an overview of ways in which people have tried to extend computing beyond the notion of algorithm. Cardelli[6] argues that global (or web) computing does not naturally correspond to the classical computation models. Examples of theoretical computing systems capable of breaking the Turing barrier are given by Blum et al.[3] for real computing, Calude and Pavlov[7] in the realm of quantum computing, Etesi and Németi[12] using the event-horizon of a black-hole and Orponen[36] in the field of neural networks. Several new models for evolving interactive systems, including interactive Turing machines and lineages of automata were designed by Van Leeuwen and Wiedermann[28, 30] and Wiedermann and Van Leeuwen[49, 50] and will be reviewed in this thesis as well.

1.1 Overview of the Thesis

The main contributions of this thesis can be divided into three parts. In the first part, we contribute several general results that complement the known non-uniform complexity theory involving classical models of computation, especially with respect to the complexity hierarchy results. Hermo and Mayordomo[18] showed that for pairs of functions f and g, such that $f \in o(g)$ and $g \in O(2^n)$, there are problems that can be solved by machines using an advice of size g, but not by machines using advice of at most size f. Although in general this result cannot be improved, we show that if we restrict the advice alphabet sizes to a fixed size, then the result is even valid if we let $f(n) < g(n)$ for infinitely many positive integers n. In the second part, we develop a new theory for evolving systems and their computations, based on the ideas behind the new models that have previously been introduced by several authors (e.g. Wegner[45], Van Leeuwen en Wiedermann[27] and Stepney et al.[40]). As it is already known that the models possess non-recursive computational powers, we focus especially on limiting this power by introducing new complexity measures, and we develop a whole new complexity theory of evolving systems. In the last part, we develop a new framework that allows us to deal with many different implementations of evolving systems. The framework is inspired by notions from topology and optimization theory, especially local search algorithms and genetic algorithms. The framework can be applied in different contexts where sequences are used to model evolving systems.

The layout of the Chapters follows this division. The first few Chapters serve as the basis for the theory that is developed in the later Chapters and deal with classical non-uniform complexity theory. In Chapter 3, several techniques, based on diagonalizing arguments and counting arguments, are combined to achieve several improvements over the results set forth by Karp and Lipton[22] and Hermo and Mayordomo[18]. These techniques are used in Chapter 5 to achieve similar results

for models of evolving systems. The relationship between advice functions and sequences is studied in Chapter 4. This links Chapter 5 with Chapters 6 and 7. Chapters 5, 6 and 7 form the core of the thesis. In these Chapters, several fundamental models for evolving systems are discussed and compared to each other as well as to classical machine models ("breaking the Church–Turing barrier"). The focus of the Chapters lies on the challenge of developing a new complexity theory, more suitable for evolving systems. Finally, in Chapter 8, all the theory that has been developed is cast into a new all-encompassing framework. This framework allows many of the hierarchy results of the previous Chapters to be restated and proved in a unifying way, which extends also to fields outside the realm of evolving systems.

1.1.1 Outline and Contributions of the Chapters

In Chapter 2, we introduce the concepts and definitions that we use throughout this thesis. The material in this Chapter is assumed to be known by the reader. While most of the material in this Chapter should be known to graduate students, it is included to ensure that the details of the definitions of the reader match the details of ours.

The basics of non-uniform complexity theory are contained within Chapter 3. A fundamental concept of (non-uniform) complexity theory is the complexity hierarchy, i.e., the fact that computational devices of higher complexity can perform tasks that devices of lower complexity cannot. An original result of this type, proved by Hermo and Mayordomo[18], is examined. In this Chapter, an improvement to this result is proved.

Non-uniform complexity, as defined by Karp and Lipton[22], is based on an advice mechanism that allows computational devices to extract extra information (a string of symbols) from an oracle-like advice, based solely on the input length. The non-uniform complexity of a device is then measured by the length of the advice as a function on the input length. The computational devices are usually Turing machines.

Let f and g be two integer-valued functions. Hermo and Mayordomo proved that when $f \in o(g)$ and $g \in O(2^n)$, there are Turing machines with an advice of size g that can decide languages that cannot be decided by Turing machines with an advice of size f (see Theorem 3.16 for a more accurate statement of their result). In their original paper, Karp and Lipton[22] stated a stronger version without proof, namely that the result already holds when f is smaller than g infinitely often. It turns out that this statement is untrue when the size of the alphabet used for the advice is unbounded. However, we can prove that the statement is true when we allow only advices using alphabets of a fixed size (see Theorem 3.22).

In Chapter 4, the focus lies on sequences of machines. The relationship between sequences of machines on the one hand and machines with an advice mechanism is explored. A general equivalence result is set up, comparing the concepts of sequences and advice. While this equivalence result is already known for particular cases (e.g. the equivalence between $P/poly$ and sequences of polynomially sized Boolean circuits proved by Karp and Lipton[22]), a general result has never been stated.

The advantage of a general equivalence result is of course that needless duplication of efforts in proving equivalences between different machine models can be avoided. Furthermore, it eases comparing different machine models. With this equivalence result in place, the computational power of sequences of finite automata, as well as that of sequences of pushdown automata are examined. Several different characterizations of these computational devices are given, both in terms of sequences of machines as in terms of machines with advice capabilities (see Theorems 4.25 and 4.42).

Interactive Turing machines, as defined by Van Leeuwen and Wiedermann[28, 30], are introduced in Chapter 5. An Interactive Turing machine (ITM) typically translates a stream of input symbols into a stream of output symbols. The function that maps input streams to output streams is then a *translation*. While an ITM is just an extension of a Turing machine that accepts infinite inputs in an on-line fashion, it is the basis of an interesting theory nonetheless. Questions such as the extensibility of computations form the basis of an interesting complexity theory.

Not until an advice mechanism is incorporated into the ITM does it become a model for evolving interactive systems. The ITM with advice (ITM/A), as defined by Van Leeuwen and Wiedermann[28, 30], is the first model for evolving interactive systems that is treated in this thesis. Just as the original Turing machine model is commonly held as the machine model to which all other machine models are compared, the ITM/A can be seen as the standard for all models for evolving interactive systems. The existence of a complexity hierarchy for translations computed by ITM/A's is proved, using the theory from Chapter 3.

The next model for evolving interactive systems that is introduced is the lineage of automata. In Chapter 6, lineages of automata are defined. A lineage of automata is a sequence of finite automata. Each automaton in the sequence is seen as an instantiation of an evolving system. This can be compared to e.g. the lineage of man, where Homo Erectus is just an instantiation of the evolving organism man. Perhaps a more fitting example is that of a computer that gets upgraded with new hard- and software during the course of its existence. Due to the direct correspondence between automata in a sequence and instantiations of the evolving system that the sequence models, lineages are very suitable to model all kinds of problems related to evolving systems.

Unlike in the theory of genetic algorithms, we view the evolution of a system as something we have no influence on. Rather, we focus on the complexity of instances of lineages, which is defined in Chapter 7. The complexity of a given lineage is measured by counting the number of states in every automaton in the sequence. Using this complexity measure, a complexity hierarchy is established. The efficiency of lineages is then measured against that of ITM/A's.

Several techniques that were employed in this thesis, such as diagonalizing arguments and counting arguments, proved to be useful over and over again. It turns out that the models in the Chapters share important common properties. By taking advantage of these properties, the techniques could be used repeatedly, though in slightly different forms. In recognition to this fact, the concepts of the preceding Chapters are reexamined in Chapter 8 and placed in a new framework, which encompasses the common properties. The techniques can then be applied to instances of the framework without effort.

The framework builds on sequence-based models. In a sequence of objects, all objects originate from a universe. Distance measures on this universe allow us to impose bounds on the possible evolutionary changes within a sequence of these objects. Fitness functions defined on subsets of objects indicate which objects are most likely to appear next in a sequence of objects. Models of evolving systems can be seen as instances of this framework by choosing a suitable universe and distance and fitness functions.

Where previous work focused on descriptions of the objects in a sequence, we propose investigating the effects of changes or differences between objects appearing consecutively in a sequence. Using this framework, the burden of theorem-proving then shifts from reinventing techniques over and over again to finding a suitable instance of the framework. It even becomes possible to apply the framework to problems that at first glance have nothing to do with the theory of evolving systems. We illustrate the use of the framework by applying it to the models of lineages of automata as proposed in Chapter 6, the Turing machines with advice functions and recursive languages.

CHAPTER 2

Preliminaries

In this Chapter, we briefly describe some concepts that we assume to be familiar to the reader. First, we introduce some notations. Often, we resort to arguments involving counting quantities. Thus, we need integers. Whenever we use the term integer, it is implied that we mean the term non-negative integer.

Functions are also used heavily throughout this thesis. Given two integer-valued functions f and g, we say that f is *bounded* by g if $f(n) \leq g(n)$ for every n. Similarly, a function f that maps integers to strings is bounded by a function g if the length of the string $f(n)$ is at most $g(n)$ for every n.

Sequences are an important concept in this thesis. Formally, a sequence is a function f from an ordered index set I to a set of objects U. We assume that the index set I is always a countable set and usually take intervals of integers $[m, n]$ or the set of all positive integers for I. We use two common ways to denote sequences, either

$$(x_i)_{i \in I} \ , \tag{2.1}$$

or

$$\begin{aligned} &x_m, \ldots, x_n \ , \\ &x_1, x_2, \ldots \ , \end{aligned} \tag{2.2}$$

for finite and infinite sequences respectively, where the items x_i in the sequence are defined by the function f, i.e., $x_i = f(i)$ for every $i \in I$. The set U can be anything, from sets of integers to sets of machines.

Tuples are fairly similar to sequences. Although usually defined in a different way, tuples can also be viewed as functions from an index set to a set of objects U. Given a tuple $x = (x_1, x_2, \ldots, x_n)$, the i-th component of x is usually denoted by $\pi_i(x)$. The function π_i is called the i-th *projection* of x.

Machines operate on strings of symbols. A string of symbols is similar in essence to a sequence. In other literature, the symbols are called letters and strings are words consisting of letters. The symbols are taken from a finite alphabet. We only

deal with countable strings. The length of a string x is the number of symbols it contains, denoted by $|x|$. If x is an infinite string, then we let $|x| = \infty$. A string x is usually denoted by

$$\begin{array}{l} x_1 \dots x_{|x|} \ , \\ x_1 x_2 \dots \ , \end{array} \tag{2.3}$$

for finite and infinite strings respectively, where x_1, x_2, ... are the consecutive symbols of x. The n-th symbol of a string x is denoted by x_n or by $(x)_n$ for clarity. If we don't know the length of a string, we usually denote it as if it was an infinite string. Note that there is no symbol x_∞. In order to work comfortably with lengths of strings, we assume some arithmetical rules on dealing with ∞, i.e., for any integer n, we let

$$\begin{array}{lll} n + \infty & = \infty & , \\ n & \leq \infty & , \\ n & \neq \infty & . \end{array} \tag{2.4}$$

Note that $\infty + n$ is not defined for any integer n.

We allow a few operations on strings. Given a finite string u and a string x, the *concatenation* of u and x is the string ux, where ux is the string

$$u_1 \dots u_{|u|} x_1 x_2 \dots \ . \tag{2.5}$$

Observe that $|ux| = |u| + |x|$. Given a string x and an integer $n \leq |x|$, a *prefix* of length n of x is a string u of length n such that $u_i = x_i$ for all $1 \leq i \leq n$. Note that a prefix is by definition a finite string. We often denote a prefix of length n of a string x by $x_{[1:n]}$.

If Σ is a finite alphabet of symbols, then we let Σ^n be the set of strings of symbols of Σ of length n, $\Sigma^\star$ the set of finite strings of symbols of Σ and Σ^ω the set of infinite strings of symbols of Σ. Furthermore, we let

$$\Sigma^\infty = \Sigma^\star \cup \Sigma^\omega \ . \tag{2.6}$$

If x is a string of symbols from an alphabet Σ, we say that x is a string over the alphabet Σ. Sets of strings are called *languages*. The theory of finding algorithms to decide if a given string is part of a given language is essential in this thesis. If a machine M implements an algorithm to decide if an input string belongs to a language L, we say that the machine M *decides* L. In general, if a machine M decides a language, this language is denoted by $L(M)$.

2.1 Machine Models

Many machine models fall into one of two classes. In one class, machines have a finite control, i.e., a set of states which the machine can be in. A transition function governs which state the machine is in. Machine models that fall in the other class are best described as flow-networks, i.e., a set of connected gates that produce output at designated gates based on input at a predefined subset of the gates. The gates are ordered, and the outputs of the gates are produced in this order.

We introduce some general characteristics of machines. This introduction is very informal. However, the characteristics are implemented rigorously in the machine models that we consider later on.

An important aspect of a machine is its *program*. This program describes how the machine uses its resources to process the inputs. For transition-based machines, the program is the transition function. For flow-based machines, the program consists of the flow network that describes the machine.

Machines typically contain some form of immutable data (e.g. firmware). Most models have this immutable data embedded in states of a finite control or in gates of a flow-network. The *size* of a machine is a measure of the amount of immutable data it possesses. This is e.g. the number of states in its finite control, or the number of gates in the flow-network.

Apart from having immutable data, most machines have the ability to store additional data. This data can be roughly divided into data that was given to the machine as input and data that was generated by the machine itself. In many models, the additional data can be discarded or overwritten at suitable times (i.e., it is mutable). Transition-based models use tapes to store this data, with a fixed number of input tapes, while some flow-based models have a fixed number of input gates to receive input. Random-access devices store mutable data in their registers. The *input length* for a machine is the maximum amount of mutable data that is given to the machine as input. This is usually measured as the length of the used part of the input tapes. For flow-based models with input gates, the input length is the number of input gates. It follows that these models only accept inputs of one fixed length. The *space* a machine uses is a measure of the maximum amount of mutable data that is generated by the machine, as a function of the input length. This function is often called the *space usage* or *space complexity* of a machine. This is usually measured as the length of the used part of the tapes and can be determined by examining the program of the machine in question.

Machines typically need some time to process the input before they can give sensible results. The *time* a machine needs is the number of basic operations the machine performs before it produces its end-result as a function of the input length. This function is often called the *running time* or *time complexity* of a machine. For transition-based machines, this is measured by the number of transitions. For flow-based machines, this is measured by the longest path from an input gate to the output gate. The time function is determined by the program of the machine.

2.1.1 Models of Machines

We will use several different models for machines. In this section, we will recall the definitions of some classical models. Although this is all assumed to be well-known, there are many variations on the exact definitions, so in order to reduce ambiguity, we give the definitions, plus short descriptions. For more information, see e.g. Balcázar et al.[1] or Hopcroft et al.[19].

Finite Automata

A two-way deterministic finite automaton (DFA) consists of a read-only tape and a finite set of states. A transition function determines the behavior of the DFA.

Formally, a DFA is a tuple $(\Sigma, Q, q_{\text{in}}, F, \delta)$, where Σ is the alphabet for the tape, Q is the finite set of states, $q_{\text{in}} \in Q$ is the initial state, $F \subseteq Q$ is the set of accepting states and $\delta : Q \times \Sigma \to Q \times \{\text{left}, \text{none}, \text{right}\}$ is the transition function. The DFA starts in the initial state, with its tape head on the leftmost symbol of the tape. It operates by reading a symbol from the tape. This symbol and the state it is in determine the next state the DFA changes to, and in which direction the tape head moves. A computation halts when the transition function is undefined for the state it is in and the symbol it has read. An input is accepted if the DFA halts in an accepting state.

The program of a DFA consist of its transition function, the initial state and the accepting states. Any immutable data accessible to a DFA is stored in its states. The size of a DFA is measured by its number of states. A basic DFA does not generate any data, so the only mutable data of a DFA is the input on the tape. Thus, a DFA has no concept of space usage. Its running time is the number of transitions it makes before halting.

There are multiple extensions of this basic model, including one-way models, models with multiple tape heads, non-deterministic models, models with output mechanisms (called transducers) and combinations thereof. In terms of computational power, all these extensions are equivalent, the languages they decide are the *regular* languages.

Pushdown Automata

A two-way deterministic pushdown automaton (PDA) is basically a finite automaton equipped with a pushdown store. It is described by a set of states and a transition function. Formally, a PDA is a tuple $(\Sigma, \Gamma, Q, q_{\text{in}}, F, \delta)$, where Σ is the alphabet for the tape, Γ is the alphabet for the pushdown store, Q is the finite set of states, $q_{\text{in}} \in Q$ is the initial state, $F \subseteq Q$ is the set of accepting states and $\delta : Q \times \Sigma \times \Gamma \to Q \times \{\text{left}, \text{none}, \text{right}\} \times \{\text{pop}, \text{none}, \text{push}\} \times \Gamma$ is the transition function. The PDA operates by reading a symbol from the input tape and the top of the pushdown store. It uses these symbols and the state it is in to determine the next state, the head move and what to do with the pushdown store, i.e., pop the top-most symbol from the store, do nothing, or push a new symbol onto the store. In the first two cases, the fourth component of the output is ignored. The PDA accepts when it halts in an accepting state.

The program of a PDA consists of its transition function, the initial state and the accepting states. Any immutable data accessible to a PDA is stored in its states. The size of a PDA is measured by its number of states. The mutable data of a PDA consists of the input on the tape and the symbols that get stored in the pushdown store. Thus, the space usage of a PDA is the maximum size of the used part of the pushdown store. Its running time is the number of transitions it makes before halting. In terms of computational power, they are more powerful than finite automata, the languages they decide are the *context-free* languages.

Turing Machines

A Turing machine (TM) is similar to a finite automaton in that it is also based on a finite set of states and a read-only input tape. However, in addition to the

input tape, a Turing machine can have one or more writable work tapes. Formally, a Turing machine with k work tapes is a tuple $(\Sigma, \Gamma, Q, q_{\text{in}}, F, \delta)$, where Σ is the alphabet for the input tape, Γ is the alphabet for the work tapes, Q is the finite set of states, $q_{\text{in}} \in Q$ is the initial state, $F \subseteq Q$ is the set of accepting states and $\delta : Q \times \Sigma \times \Gamma^k \to Q \times \{\text{left}, \text{none}, \text{right}\}^{k+1} \times \Gamma^k$ is the transition function. The TM operates by reading a symbol from the tapes, using these symbols and the state it is in to determine the next state, what to write to the work-tapes and in which direction to move the tape heads. The TM accepts when it halts in an accepting state.

The program of a TM consists of its transition function, the initial state and the accepting states. Any immutable data accessible to a TM is stored in its states. The size of a TM is measured by its number of states. The mutable data of a TM consists of the input on the tape and the data it writes on the work tapes. Thus, the space usage of a TM is the maximum size of the used part of the work tapes. Its running time is the number of transitions it makes before halting.

There are many variations of Turing machines, including non-deterministic models, models with just one writable tape, one-way models, models with multiple tape heads per tape, models with output mechanisms and combinations thereof. In terms of computational power, all these extensions are equivalent, the languages they decide are the *recursive* languages. Turing machines are more powerful than pushdown automata.

Boolean Circuits

A Boolean circuit for binary inputs of length n is a directed acyclic graph of Boolean gates. In its topologically sorted form, a circuit is described by a sequence of gates, together with a description of each gate. The i-th gate in the sequence takes a finite number k of inputs from the results of the $i-1$ previous gates and the n input bits. It combines these k values to a result, using a Boolean function that takes k bits. An input is accepted if the result of the last gate in the sequence is 1. The size of a Boolean circuit is the number of gates, the depth is the length of the longest path from the input symbols to the last gate. Boolean circuits can be generalized in a straight-forward manner to accept strings over alphabets of size c.

The program of a Boolean circuit consists of its graph, as well as the Boolean functions at each gate. Any immutable data accessible to a Boolean circuit is stored in its gates (using constant functions). The size of a Boolean circuit is measured by its number of gates. A Boolean circuit does not generate any data, so the only mutable data of a circuit is the input. Thus, a Boolean circuit has no concept of space usage. Its running time is the longest path in the graph ending in the last gate (also referred to as its depth).

2.1.2 Machine Isomorphisms

Sometimes two different machines compute the same things. It can be possible that they implement two different algorithms to perform the same tasks. However, it is also possible that the machines are essentially the same, just labeled differently.

If this is the case, we call the machines *isomorphic*. More formally, two machines A and B of the same model are isomorphic if there is a bijection that maps the defining sets of A to the defining sets of B such that the relations on the sets of A hold iff the relations hold on the images of the sets of A in the sets of B. An example will clarify this.

Example 2.1. An automaton is described by the sets Σ and Q and the relations q_{in}, F and δ. Let A be the automaton $(\Sigma, Q, q_{\text{in}}, F, \delta)$ and B the automaton $(\Sigma', Q', q'_{\text{in}}, F', \delta')$. Then A and B are isomorphic iff there is a bijection ϕ from Σ to Σ' and a bijection ψ from Q to Q' such that

$$\begin{aligned} \psi(q_{\text{in}}) &= q'_{\text{in}} \ , \\ \psi(F) &= F' \ , \\ \psi(\pi_1(\delta(a,q))) &= \pi_1(\delta'(\phi(a), \psi(q))) \ , \end{aligned} \tag{2.7}$$

for every $a \in \Sigma$ and $q \in Q$.

Intuitively, an isomorphism relabels the states from the first automaton to match the labels from the second, making sure that the structure of the automaton is not affected.

2.2 Encoding Mechanisms

An encoding is a way to encapsulate information, with the goal to describe the information concisely but completely in a given framework. The informational aspect of encoding mechanisms is important enough to us that we also refer to encoded information as a description. More formally, an encoding mechanism can be seen as an injective function f from a set U to a set of strings. The set U contains all the information that can be encoded and the set of strings contains all the possible descriptions. Thus, given some object $u \in U$, the encoding f produces a description $f(u)$ of u. A description of an object u is denoted by $\langle u \rangle$. Although any injective function can be an encoding mechanism, we are usually interested in encoding mechanisms that can be implemented by machines. Applying the Church–Turing thesis[24], we therefore restrict ourselves to recursive functions for encoding mechanisms.

The efficiency of an encoding mechanism is measured by the length of the descriptions of the objects one is interested in. We assume that any object in U can be described with a finite number of words, so the lengths of the descriptions are finite. A good encoding mechanism maps the interesting (or often occurring) objects to small descriptions. A related subject is the study of compression algorithms. See Li and Vitányi[31] for more information about encoding mechanisms.

2.2.1 Encoding Strings

Often, we wish to encode strings in an efficient way. A function that maps strings to strings is also called a *translation*. A standard result from Kolmogorov complexity theory states that for any chosen encoding mechanism there are strings of length

n that need a string of length $\Omega(n)$ to be described (see Li and Vitányi[31]). On the other hand, there are encoding mechanisms to encode strings of length n to strings of length $O(n)$ (see e.g. Proposition 3.9), except for pathological cases such as encoding arbitrary strings to unary strings. Thus, we assume that strings of length n can be described by strings of length $O(n)$ in the relevant frameworks we encounter.

2.2.2 Encoding Tuples of Strings

Of particular interest is the concept of encoding multiple strings. The encoding of two strings x and y is denoted by a pairing function $\langle x, y\rangle$. Let x and y be finite strings over an alphabet Σ of size 2 or more. Assume Σ contains the symbols 0 and 1. Then x and y can be encoded by the string

$$\langle x, y\rangle = x_1 x_1 \dots x_{|x|} x_{|x|} 01 y_1 y_1 \dots y_{|y|} y_{|y|} \quad . \tag{2.8}$$

If y is a string over a different alphabet, then y can of course first be encoded by a string over Σ. Thus, any two strings x and y can be described with a string of length $O(|x|+|y|)$. This can of course be extended to tuples with more components. In particular, m strings of length l can be encoded with a string of length $O(m \cdot l)$.

2.2.3 Encoding Machines

Any machine can be completely described by giving a description of its program and its immutable data. The *description size* of a machine is the length of a string that describes the machine. A description of a machine must include a description of its program. Usually, there are approximately $O(m^m)$ different programs for machines of size m. It follows that there must be descriptions of length $\Omega(m \log m)$.

Example 2.2. Consider a one-way finite automaton with a set of m states Q that works with an alphabet Σ of size c. The transition function maps $\Sigma \times Q$ to Q. Therefore, the domain of the transition function has size cm and the range has size $m+1$ (m states or an undefined transition). This gives $(m+1)^{cm}$ possibilities, leading to a description length of at least $\Omega(cm \log m)$ for some machines.

Of course, there are always pathological examples of machine models that allow far less different programs, which implies that these models can be described with shorter strings. We shall assume however, that the following Property holds for all machine models that we are interested in.

Property 2.3. For every encoding mechanism there are machines of size m that can only be described with strings of length at least $\Omega(m \log m)$.

Fortunately, for many machine models, there are encoding mechanisms that produce descriptions of length $O(m \log m)$ for machines of size m. For transition-based machines, this is done by listing all the states of its finite control, listing for each input the output of the transition function and listing the additional conditions that make up the program. The resulting description is called a *transition-list*.

Example 2.4. A deterministic finite automaton of size m over a fixed alphabet can be described by a string of length $O(m \log m)$. The description consists of a list of states represented by the numbers 1,..., m in binary (using a string of length $O(m \log m)$), a list of tuples of states and symbols combined with the outputs of the transition function for these tuples, (using a string of length $O(m \log m)$), a list containing the initial state (a string of length at most $O(\log m)$) and a list of the accepting states (using a string of length at most $O(m \log m)$).

For flow-based machines, this is done by listing the gates of the graph, listing for each gate in the graph its adjacent gates and listing the additional conditions that make up the program. The resulting description is called a *flow-list*.

Example 2.5. A Boolean circuit with m gates and a fan-in of 2 can be described with a string of length $O(m \log m)$. The description consists of a list of gates, represented by numbers, combined with the gates that are adjacent to the numbered gates (using a string of length $O(m \log m)$), a list of the gates combined with a description of the Boolean functions (using a string of length $O(m \log m)$) and a list containing the last gate (using a string of length at most $O(\log m)$).

2.3 Topology

Throughout the thesis, we use some concepts from topology. For completeness sake, we give a short introduction to this theory. See Munkres[34] for more on this subject.

Definition 2.6. *Let U be a set (called the universe). A topology for U is a collection $\mathcal{T}$ of sets called* open *sets, such that*

- *the empty set is open,*
- *U is open,*
- *the union of any collection of open sets is open,*
- *the intersection of any finite collection of open sets is open.*

The complement (in U) of an open set is a closed *set.*

For the rest of this section, we let U be a universe with a topology $\mathcal{T}$. Topologies are often defined by describing a basis for the topology.

Definition 2.7. *A* basis *for a topology on U is a collection of sets called* basis *sets, such that for every $x \in U$:*

- *there is a basis set that contains x,*
- *if x belongs to the intersection of two basis sets B_1 and B_2, then there is a basis set B_3 that contains x and is a subset of $B_1 \cap B_2$.*

Given a basis, a subset S of U is open if it is a union of basis sets.

Definition 2.8. *A point x is a* limit point *of a set S if every open set that contains x also contains a point of $S - \{x\}$.*

Definition 2.9. *A sequence x_1, x_2, ... converges to a* limit *x if every open set that contains x also contains all but finitely many points of the sequence.*

Definition 2.10. *A subset S of U is* compact *if every collection of open sets that covers S contains a finite collection of open sets that also covers S.*

Note that every finite set is compact.

2.3.1 Examples of Topologies

Next, we will give a few useful examples of topologies on sets (cf. Munkres[34]).

Trivial Topology

The *trivial topology* consists only of the subsets $\emptyset$ and U. In this topology, every subset is compact.

Discrete Topology

For any set U, the set of all subsets (the power-set) is a topology for U known as the *discrete topology*. In this topology, the compact sets are precisely the finite subsets.

Topology on a Finite Set

If U is a finite set, all its subsets are finite and therefore compact, regardless of the choice of topology.

Concentric-Sphere Topology

Suppose we are given a metric on U and a given fixed object $x \in U$. Then the set of ϵ-spheres centered around x is a topology. In this topology, the non-empty compact sets are precisely those subsets that contain an object with maximum distance to x.

Metric Topology

Given a metric on U, a basis for the *metric topology* consists of ϵ-spheres (where the center is no longer fixed). In a metric topology, a subset is compact iff every infinite subset of it has a limit point.

Let U be the class of languages over an alphabet Σ, i.e., U is the power-set of $\Sigma^\star$. Then, a metric δ on U is defined as follows: for any two languages L and L', we let $\delta(L, L') = 0$ if L and L' are equal. Otherwise, a smallest integer n and a string w of length n exist such that $w \in (L - L') \cup (L' - L)$. In this case, $\delta(L, L') = \frac{1}{n}$ if $n > 0$ and $\delta(L, L') = 2$ if $n = 0$.

Order Topology

Suppose U is a linear order, with the least upper bound property[1]. A basis for the *order topology* consists of the open intervals

$$(a, b) = \{\ x \in U \quad | \quad a < x \ \wedge \ x < b\ \}\ , \tag{2.9}$$

with $a, b \in U$ or $a = -\infty$ or $b = \infty$. In this topology, the compact sets are the closed and bounded sets.

Prefix Topology

Let Σ be an alphabet and let $U = \Sigma^\infty$. A basis for the *prefix topology* consist of the sets

$$\mathbb{B}(u) = \{\ x \in \Sigma^\infty \quad | \quad u \text{ is a prefix of } x\ \}\ , \tag{2.10}$$

for all finite strings $u \in \Sigma^\star$.

Product Topology

Let I be a set and suppose we have a collection of sets U_i for $i \in I$. Let the universe U be given by:

$$U = \prod_{i \in I} U_i\ . \tag{2.11}$$

If U_i has its own topology for every $i \in I$, then the *product topology* for U is given by the open sets

$$O = \prod_{i \in I} O_i\ , \tag{2.12}$$

such that O_i is open in U_i for every $i \in I$ and $O_i = U_i$ for all but finitely many $i \in I$. A set S is compact in U iff its projection to each of its components is compact. This fact is known as Tychonov's theorem (see Munkres[34]).

[1] Every non-empty subset of U that is bounded from above has a least upper bound.

CHAPTER 3

Non-uniform Complexity Theory

In the theory of computing devices and computation, many different models have been distinguished, such as Turing machines, Boolean circuits, finite automata and neural networks. These models can divided into two classes: models that only take inputs of a fixed length (i.e., with a possibly different device or program for every input length) and models that take inputs of any arbitrary length. Both classes lead to a rich complexity theory (cf. Balcázar et al.[1]). We are especially interested in the broad class of models of the former type.

We can model devices in this class by considering sequences of machines, one machine for each input length. Sequences of Boolean circuits are a prime instance of this class, but so are the various types of machines using advice (i.e., with a possibly different advice value for every input length). Since the sequences can be arbitrary, there is in general no way to generate a sequence recursively, that is, we cannot give a *uniform* description of the machines of the sequence. Therefore, we call these sequences of computing devices *non-uniform models* and the theory based on these models *non-uniform complexity theory*.

The concept of advice functions, which was introduced by Karp and Lipton[22], is thoroughly connected with sequences. We make heavy use of sequences throughout this thesis, and advice functions prove to be a useful tool. In particular, we are interested in the way different advice functions can lead to different complexities of computation, when added to familiar machine models.

It is known that for every pair of exponentially bounded, integer-valued functions f and g, there are languages that can be decided by a Turing machine with an advice of size g, but not by any Turing machine using an advice of size f, when $f \in o(g)$. In this Chapter, we give a detailed proof of this fact. Examination of this proof allows us to improve the result when we bound the size of the advice alphabets. In this case, there are languages that can be decided by a Turing machine with an advice of size g, but not by any Turing machine using an advice of size f if $f(n) < g(n)$ for infinitely many integers n. Thus, we obtain a finer separation of

the non-uniform complexity classes. This settles a question that was left open by Karp and Lipton[23].

The structure of the Chapter is as follows. First, the concept of advice functions is introduced. Then, we give an example using characteristic strings. Advice functions may be encoded using advice alphabets of different sizes. We give an efficient method to convert advice functions using an advice alphabet to another advice functions using another advice alphabet, such that the advice functions still contain the same information. Next, we introduce non-uniform complexity classes and prove several new hierarchy results.

3.1 Advice Functions

Many sequences are of such a nature that there is no uniform way to describe them. This is because there are uncountably many sequences and only countable many uniform descriptions. Thus, we may have to describe sequences by listing each item in the sequence separately. It is convenient to make a distinction between a common uniform part, and the part which makes it impossible to give a short description: the non-uniform part.

Karp and Lipton[22] introduced the concept of *advice* functions to formalize this distinction. An advice function is a function from the positive integers to the set of strings. Given a sequence, the advice function maps each integer n to the description of the n-th element in a sequence (which is a string). To obtain the sequence, we also need a program to turn the description into the element. Thus, we have split the sequence into a uniform part (the program) and a non-uniform part (the advice function). Just as the original sequence, there may be no way to describe the advice function besides listing each of its values.

By moving the uniform part into the program, one can try to make the non-uniform part as small as possible. The size of the advice function can be seen as a measure of the amount of non-uniformity of a sequence. The size functions induce a partial order on the class of sequences. We say that one sequence is more non-uniform than another if the size function of the first sequence is larger than the size function of the second, a concept that is related to the amount of randomness of a string, see Li and Vitányi[31] for details.

3.1.1 Non-uniformly Defined Languages

In the context of the publication by Karp and Lipton[22], a language corresponds to a sequence: the n-th item in the sequence consists of the strings of length n belonging to the language. A program uses its advice function to decide for strings of length n whether to accept or reject them. The following notation is introduced by Karp and Lipton[22].

Definition 3.1. *Let $L \subseteq \Sigma^\star$ be a language and $\alpha : \mathbb{N} \to \Omega^\star$ an advice function. Define the set*

$$L{:}\alpha = \{\ x \in \Sigma^\star \quad | \quad \langle x, \alpha(|x|)\rangle \in L\ \}\ . \tag{3.1}$$

Let $\mathcal{C}$ be a class of languages and $\mathcal{F}$ a class of integer-valued functions. Define the non-uniform class

$$\mathcal{C}/\mathcal{F} = \{\ L{:}\alpha \quad | \quad L \in \mathcal{C},\ |\alpha| \in \mathcal{F}\ \}\ . \tag{3.2}$$

We write $\mathcal{C}/g$ for the class $\mathcal{C}/\{\ h \quad | \quad h$ is bounded by $g\ \}$.

If L is decided by a machine M, then we say that M decides $L{:}\alpha$ with advice function α. If α is bounded by an integer-valued function g, then we say that M uses an advice of size g. We call Σ the input alphabet for M and Ω the advice alphabet. We assume that M uses the tape alphabet Σ. Thus, Ω is usually a subset of Σ. The idea is that M has a list of advices, one for each length. If M gets an input x, then M determines the length of x. Now M can use the extra information encoded into the advice string for this length to compute the correct output.

Remark 3.2. To decide if a string x belongs to the language $L{:}\alpha$, a Turing machine that decides L takes the tuple $\langle x, \alpha(|x|)\rangle$ on its input tape. It is clear that the input for the Turing machine consists of both x and $\alpha(|x|)$. As a consequence, the time and space measures of a machine with an advice mechanism, which are both defined as a function of the input length, are both dependent on the length of the advice.

3.1.2 Characteristic Strings

Next, we use characteristic strings as an advice function. Using such an advice function, any language can be decided with the correct advice. Characteristic strings also proves useful to establish a hierarchy of non-uniform complexity classes.

Definition 3.3. *Let Σ be an alphabet of size c and n an integer. Let $x_1, \ldots, x_{c^n}$ be an enumeration of all strings in Σ^n. Let L be a subset of Σ^n. A* characteristic string *for L is a binary string w such that $w_i = 1$ iff $x_i \in L$.*

Remark 3.4. Note that the definition says nothing about the length of characteristic strings. In some literature, *the* characteristic string of L is a characteristic string of length c^n. We denote it as the standard characteristic string.

Let l be the largest integer such that $x_l \in L$. Then the length of a characteristic string for L has length at least l. Furthermore, $w_i = 0$ for all $i > l$. The shortest characteristic string for L has length l.

Definition 3.5. *We define the Characteristic String (CS) problem for tuples. Consider a tuple $\langle x, w\rangle$, with x a string in Σ^n and w a binary string. Let X be the subset of Σ^n for which w is a characteristic string. Then $\langle x, w\rangle \in CS_\Sigma$ iff $x \in X$.*

Let L be a language over an alphabet Σ. Define the advice function α by letting $\alpha(n)$ be a characteristic string of $L \cap \Sigma^n$. It follows that

$$L = CS_\Sigma{:}\alpha\ . \tag{3.3}$$

Proposition 3.6. *Any language can be decided with an advice of exponential size.*

Proof. Let L be a language over an alphabet of size c. Definition 3.5 and the remarks following it show that L can be decided with an advice function containing characteristic strings for the sets $L \cap \Sigma^n$, for all n. The length of the characteristic strings is at most c^n.

□

Remark 3.7. The size of the advice function depends on the distribution of the strings in the language: the n-th advice value need only be as large as the shortest characteristic string of $L \cap \Sigma^n$. This is an optimization from using the standard characteristic string. However, if L contains the last element of the enumeration, this optimization still yields exponential advice length.

It turns out that for every optimization scheme there are languages that cannot be decided with advice of less than exponential size. This follows directly from Theorem 3.16.

We use the set CS_Σ to establish some complexity results. These results can be applied to all decidable classes that contain CS_Σ. Thus, it is useful to know which classes contain CS_Σ. The following Proposition shows us which complexity classes contain the set.

Proposition 3.8. *The language CS_Σ can be recognized in linear time.*

Proof. We will construct a Turing machine M that recognizes CS_Σ. Let $\langle x, w\rangle$ be an input to M. Let $|x| = n$ and $|w| = m$. The machine uses two work tapes to store x and w plus an extra work tape. If w contains a symbol other than 0 or 1, the input is rejected. Similarly, x must be in Σ^n. After separating the input tuple, we ignore the actual input tape and call the tape containing x the "input tape" and the tape containing w the "advice tape".

Now, M will try to find the position of x on the advice tape to determine the outcome. If w is not long enough, then M will reject the tuple.

Let c be the size of Σ. For $0 \leq i \leq n$, let $k_i = 1 + \sum_{j=1}^{i} x_j c^{j-1}$. We need to determine the value of w_{k_n} to decide whether $\langle x, w\rangle \in CS_\Sigma$. The machine works in stages. At the beginning of stage $i \geq 1$, the following invariants hold:

- The head of the input tape is on cell i;
- the head of the advice tape is on cell k_{i-1};
- the work tape is of length $i - 1$.

In stage i, the machine reads the symbol x_i and moves the head of the advice tape $x_i \cdot c^{i-1}$ steps to the right, using the work tape of length $i - 1$. This can be done in $O(c^i)$ steps by generating all strings of length $i - 1$ on the work tape, and moving the advice head x_i steps for every generated string. If the advice head tries to move past the end of w, the input is rejected. After this, the advice head will be at position k_i. Then, M moves the input head one step to the right and writes a symbol to the work tape to increase the tape length by one. It follows that the invariants hold at the beginning of each stage.

When the input head reaches the blank after x, then the advice head is at position k_n, which is the position of x on w. If the symbol under the advice head is 1, then M accepts, otherwise M rejects the input.

We need $O(n+m)$ steps to extract x and w from the input. The number of steps per stage is about the number of moves to the right on the advice tape (but at least one). This means that the total number of moves to the right is not more than m. Hence the total time needed is $O(n+m)$, so M works in linear time.

□

3.1.3 Conversion between Different Alphabets

When machines are allowed to use advices over alphabets of different sizes, the advice functions can be converted from one alphabet to the other. A string over an alphabet of size b can be considered as a number in base b. Converting this number to a different base d will result in a string over an alphabet of size d. The original string can be recovered by converting the number in base d back to the number in base b. This can be done in quadratic time, as Proposition 3.9 shows.

Proposition 3.9. *Let Ω be an alphabet of size $d > 1$ and Θ an alphabet of size $b > 1$. Let w be an Ω-string. Then w can be turned into a Θ-string v such that $\sum_{i=1}^{|w|} w_i d^{i-1} = \sum_{j=1}^{|v|} v_j b^{j-1}$, in quadratic time, using linear space. The length of v is at most $\lceil (\log d / \log b)|w| \rceil$.*

Proof. First, we copy the input w to a work tape. We give the algorithm in pseudo code, and leave the construction of a Turing machine to the reader. See Algorithm 3.1. Let n be the size of w, let $s_0 = \sum_{i=1}^{n} w_i d^{i-1}$. In the j-th execution of the while loop, the algorithm calculates the unique s_j and v_j such that $s_{j-1} = b \cdot s_j + v_j$ and $0 \leq v_j < b$, and stores s_j in base d in the array that previously held w. The algorithm continues until s_j becomes 0, which happens after m steps. Since $v_j b^{j-1}$ equals $s_{j-1} b^{j-1} - s_j b^j$ and $s_m = 0$, it follows that

$$\sum_{j=1}^{m} v_j b^{j-1} = s_0 b^0 - s_m b^m = s_0 \ . \tag{3.4}$$

The left-hand side of 3.4 can become as large as $b^m - 1$, while $\sum_{i=1}^{n} w_i d^{i-1} \leq d^n - 1$. The smallest value of m for which $b^m \geq d^n$ is $(\log d / \log b)n$. Thus $m \leq \lceil (\log d / \log b)n \rceil$.

Each execution of the while loop costs $O(n)$ time, since the tests and the arithmetical operations can all be implemented in constant time, using the transition function of the resulting Turing machine. Since m is in $O(n)$, the conversion can be done in quadratic time. The only space that is needed is the tape to store the values of s_j, thus linear space is sufficient.

□

Suppose $d = a^k$ and $b = a^l$ for an integer a. Then a Turing machine can read l digits of w and convert them into k digits of v. In this way, the machine can convert any d-ary number to base b in linear time.

We define a generalized version of CS.

Definition 3.10. *Consider a tuple $\langle x, w \rangle$, with $x \in \Sigma^n$ and w a string in Ω. Let v be the binary string such that $\sum_{i=1}^{|v|} v_i 2^{i-1} = \sum_{j=1}^{|w|} w_j |\Omega|^{j-1}$. Then $\langle x, w \rangle \in CS_{\Sigma,\Omega}$ iff $\langle x, v \rangle \in CS_\Sigma$.*

```
/*******************************************************\
* input:  w[1,...,n], an array with d-ary digits.       *
* output: v[1,...,m], an array with b-ary digits,       *
*                     such that the number stored in w  *
*                     equals the number stored in v.    *
\*******************************************************/

// j indicates the head of the output tape.
int j <- 1
v[j] <- 0
while ( true ) do
    // Exit the loop when the number N_w stored in w is 0.
    Boolean zero <- true
    for ( i <- 1 to n ) do
        if ( w[i] > 0 ) then
            zero <- false
        endif
    endfor
    if ( zero ) then
        exit while loop
    endif

    // Divide N_w by b,
    int wi <- 0
    int remains <- 0
    for ( i <- n downto 1 ) do
        wi <- w[i] + remains*d
        w[i]   <- floor( wi / b )
        remains <- wi - w[i]*b
    endfor

    // and store the remainder in v[j]
    v[j] <-remains
    j    <- j + 1
endwhile
return v
```

Algorithm 3.1. An algorithm to convert d-ary numbers to base b.

Note that $CS_{\Sigma} = CS_{\Sigma,\{0,1\}}$. Observe that $CS_{\Sigma,\Omega}$ can be decided in quadratic time. If the size of Ω is a power of two, then $CS_{\Sigma,\Omega}$ can be decided in linear time.

Consider the following Property of classes of languages.

Property 3.11. Let $\mathcal{C}$ be a class of languages and L a language. Suppose $L = L' : \alpha$ for a language $L' \in \mathcal{C}$ and an advice function α over an alphabet of size d. Let β be an advice function over an alphabet of size b, such that for every n the string $\beta(n)$ is obtained by converting $\alpha(n)$ to base b. Then, there is a language $L'' \in \mathcal{C}$ such that $L = L'' : \beta$.

For example, the class P satisfies this Property.

3.2 Non-uniform Complexity Classes

It is intuitive to assume that as machines have access to larger advice functions, they are able to decide languages that were previously undecidable. In accordance with this intuition, Karp and Lipton[22] claimed that $P/f \subset P/g$ when $f(n) < g(n)$ holds infinitely often. In a later publication[23], the statement was weakened

to $P/f \subseteq P/g$ (which is true by definition). Hermo and Mayordomo[18] proved that the inclusion is proper when f in $o(g)$. They used notions of Kolmogorov Complexity to arrive at this conclusion. We will give a different proof, which will give us more insight into the sizes of the involved classes. This allows us to improve the result when we restrict the size of the advice alphabet.

3.2.1 A Technical Result

In this and the following subsections, we will give some results which share the same idea. To minimize the number of repetitions, we will state the idea in a separate, technical Lemma.

Let Σ be an input alphabet and Ω an advice alphabet. Let g be an integer-valued function. Our goal is a statement of the form: For suitable functions f and integers n, a machine using advice of size g and an advice alphabet Ω can decide more different subsets of Σ^n than any machine using advice of size f. When this is true, we can use a diagonalizing argument to construct a language that cannot be decided with advice f (see Lemma 3.14).

Proposition 3.12. *Let Σ be an alphabet of size c and Ω an alphabet of size $d \geq 2$. Let g be an integer-valued function such that $g(n) \leq c^n / \log d$ for all n. Then, a Turing machine for $CS_{\Sigma,\Omega}$ can decide $d^{g(n)}$ different subsets of Σ^n with advices of size g.*

Proof. Since there are no more than $2^{(c^n)}$ subsets of Σ^n, a characteristic string corresponds to a unique subset iff it has length c^n or less.

Any string w of length $g(n)$ over the alphabet Ω can be converted into a binary string v of length $\lceil (\log d) g(n) \rceil$ (see Proposition 3.9). Thus, v is a characteristic string of length at most c^n. It follows that v corresponds to a unique subset of Σ^n. The string w corresponds to the same subset.

□

Proposition 3.13. *Let Σ be an alphabet of size c and Θ an alphabet of size b. Let h be an integer-valued function. Then, an arbitrary Turing machine can decide at most $\sum_{i=0}^{h(n)} b^i$ different subsets of Σ^n with advices of size at most h over the alphabet Θ.*

Proof. Since there are only b^i different advice values of length i over the alphabet Θ for every $0 \leq i \leq h(n)$, the result follows immediately.

□

Let M be an arbitrary Turing machine using an advice alphabet of size b. Consider the following inequality.

$$\sum_{i=0}^{h(n)} b^i < d^{g(n)} \quad . \tag{3.5}$$

When it holds, there is a subset of Σ^n that can be decided by a Turing machine for $CS_{\Sigma,\Omega}$ with an advice of size g over an alphabet Ω of size d, but not by M with an

advice of size at most h. Inequality (3.5) is the basis for the next results. Basically, for every combination of Turing machine and advice size, a suitable integer n for which (3.5) holds has to be found.

With these facts in place, we can give the technical Lemma. It can be viewed as a recipe for the actual Theorems which are given later.

Lemma 3.14. *Suppose the following conditions all hold.*

- *Let Σ be an input alphabet of size c.*
- *Let Ω be an advice alphabet of size $d \geq 2$.*
- *Let $\mathcal{B}$ be a class of allowed advice alphabet sizes.*
- *Let g be an integer-valued function.*
- *Let $\mathcal{F}$ be a class of integer-valued functions.*
- *Let $\mathcal{H}$ be a countable class of integer-valued functions.*
- *Let N be an integer.*
- *Finally, let L be a language that can be decided by a Turing machine for $CS_{\Sigma,\Omega}$ with an advice of size g.*

Suppose that $g(n) \leq c^n / \log d$ for every n. Suppose that for every $f \in \mathcal{F}$ and every $b \in \mathcal{B}$ there is a function $h \in \mathcal{H}$ such that $f(n) \leq h(n)$ for all but finitely many integers n and (3.5) holds for h and b, for infinitely many integers n. Then, there is a language L' that can be decided by a Turing machine for $CS_{\Sigma,\Omega}$ with an advice of size g over the advice alphabet Ω, but not by any Turing machine with an advice of size bounded by a function $f \in \mathcal{F}$ over an alphabet in $\mathcal{B}$. Furthermore, $L' \cap \Sigma^n = L \cap \Sigma^n$ for all $n \leq N$.

Proof. Let α be the g-bounded advice used to decide L. Consider the class of all tuples of the form (h, b, m, M) for functions $h \in \mathcal{H}$, integers b and m and Turing machines M, such that (3.5) holds for h and b, for infinitely many integers n. Observe that this class is countable, so there is an enumeration of all its tuples. To each tuple (h, b, m, M), an integer $n > m$ is assigned such that (3.5) holds for h and b and this n. The integer n is chosen such that n is larger than N and larger than the integers that were assigned to previous tuples in the enumeration. This is possible since there are infinitely many of such integers n for every h and b.

By combining (3.5) and Propositions 3.12 and 3.13, it follows that there is an advice value $\alpha'(n)$ of size $g(n)$ that helps a Turing machine for $CS_{\Sigma,\Omega}$ to decide a subset L_n of Σ^n that cannot be decided by M with any advice of size bounded by h over an alphabet of size b.

The advice function α' is constructed in this way, with $\alpha'(n) = \alpha(n)$ if n is not assigned to a tuple. It follows that α' is of size g. Let L' be the language that a Turing machine for $CS_{\Sigma,\Omega}$ decides with advice α'. Note that $L' \cap \Sigma^n = L_n$ if n is assigned to a tuple and $L' \cap \Sigma^n = L \cap \Sigma^n$ otherwise. In particular, $L' \cap \Sigma^n = L \cap \Sigma^n$ for all $n \leq N$.

Suppose that L' can be decided by a Turing machine M with an advice β of size bounded by a function $f \in \mathcal{F}$ over an alphabet of size $b \in \mathcal{B}$. Let $h \in \mathcal{H}$ be a function and m an integer such that $f(n) \leq h(n)$ for all $n > m$ and (3.5) holds for h and b, for infinitely many integers. Let n be the integer assigned to the tuple (h, b, m, M). The Turing machine M uses the advice string $\beta(n)$ to recognize a subset $S = L' \cap \Sigma^n$. Since $|\beta(n)| \leq f(n)$ and $n > m$, it follows that $|\beta(n)| \leq h(n)$.

Since (3.5) holds for h and b and this n, the set L_n is different from S. But by construction, $L' \cap \Sigma^n$ equals L_n. This yields a contradiction. So L' cannot be decided by any Turing machine using an advice of size bounded by a function $f \in \mathcal{F}$ over an alphabet of size $b \in \mathcal{B}$.

□

The existence of a language L' in Lemma 3.14 was proved by constructing a single language. Here, we will show that there are in fact many of such languages. Recall the distance function on the class of languages from Chapter 2. A subset of languages is dense in this class if for every language L and every $\epsilon > 0$, we can find a language in the subset with distance less than ϵ to L.

Proposition 3.15. *Suppose the following conditions all hold.*

- *Let Σ be an input alphabet of size c.*
- *Let Ω be an advice alphabet of size $d \geq 2$.*
- *Let $\mathcal{B}$ be a class of allowed advice alphabet sizes.*
- *Let g be an integer-valued function.*
- *Let $\mathcal{F}$ be a class of integer-valued functions.*
- *Let $\mathcal{H}$ be a countable class of integer-valued functions.*

Suppose that $g(n) \leq c^n / \log d$ for every n. Suppose that for every $f \in \mathcal{F}$ and every $b \in \mathcal{B}$ there is a function $h \in \mathcal{H}$ such that $f(n) \leq h(n)$ for all but finitely many integers n and (3.5) holds for h and b, for infinitely many integers n. Let $\mathcal{G}$ be the class of integer-valued functions such that $g' \in \mathcal{G}$ iff $g'(n) \leq g(n)$ holds for all but finitely many n. Let $\mathcal{L}$ be the class of languages such that $L' \in \mathcal{L}$ iff L' can be decided by a Turing machine for $CS_{\Sigma,\Omega}$ with an advice of size $g' \in \mathcal{G}$ over the advice alphabet Ω, but not by any Turing machine with an advice of size bounded by a function $f \in \mathcal{F}$ over an alphabet in $\mathcal{B}$. Then $\mathcal{L}$ is dense in the class of languages over the alphabet Σ.

Proof. Let S be an arbitrary language over the alphabet Σ and let $\epsilon > 0$. Define $m = \lceil 1/\epsilon \rceil$. We let L be the finite set containing the strings in X of length m or less. Observe that L can be decided by a Turing machine for $CS_{\Sigma,\Omega}$ with an advice function that is bounded by a function in $\mathcal{G}$. Apply Lemma 3.14 with $N \geq m$. Then, we obtain a language $L' \in \mathcal{L}$. By construction of L, it follows that $L' \cap \Sigma^n = L \cap \Sigma^n = X \cap \Sigma^n$ for all $n \leq m$. Thus, L' is a language in $\mathcal{L}$ with a distance of less than ϵ to S.

□

It follows that $\mathcal{L}$ is dense in any subclass of subsets of $\Sigma^\star$. As a result, the non-uniform complexity classes are evenly distributed in the class of languages, i.e., it is not possible to find a small open set of languages that contains no languages of a given non-uniform complexity. Another conclusion is that there are infinitely many languages to prove the results of the next sections.

3.2.2 Advice Alphabets of Unbounded Size

Turing machines may have arbitrarily large tape alphabets. Thus, the sizes of advice alphabets that machines can use can grow equally large. This implies that

arbitrary amounts of information can be encoded into advice functions by using large enough advice alphabet sizes. Re-examining (3.5), it becomes clear that the inequality holds when $b^{h(n)+1} - 1 < d^{g(n)}$. Therefore, $h(n)$ should be at most $(\log d / \log b)g(n) - 1$. Since the advice alphabet size b depends on the machine in the n-th tuple of the enumeration, it follows that b is a function of n. Thus, it becomes clear that f must be in $o(g)$. Theorem 3.16, originally observed by Hermo and Mayordomo[18] with an argument from Kolmogorov complexity theory, follows from this result.

Theorem 3.16 (Hermo–Mayordomo). *Let $\mathcal{D}$ be any recursive class containing $DTIME(n)$ and let $\mathcal{C}$ be the class of all recursive languages. Let $\mathcal{F}$ and $\mathcal{G}$ be classes of integer-valued functions such that there is a function $g \in \mathcal{G}$ with $g \in o(2^n)$ and $f \in o(g)$ for every $f \in \mathcal{F}$. Then $\mathcal{D}/\mathcal{G}$ is not included in $\mathcal{C}/\mathcal{F}$.*

Proof. Let Σ and Ω be binary alphabets. Let $\mathcal{B}$ be the set of positive integers. Consider the countable class of integer-valued functions

$$\mathcal{H} = \left\{ \left\lfloor (\log b)^{-1} \right\rfloor \cdot g(n) - 1 \quad | \quad b \geq 2 \right\} \quad . \tag{3.6}$$

Let $N = 0$ and let $L = \emptyset$. Observe that L can be decided with an advice of size 0. Note also that $g(n) \leq 2^n / \log 2$ for every n.

Let f be a function in $\mathcal{F}$ and b an integer in $\mathcal{B}$. Let $h \in \mathcal{H}$ be the function defined by $h(n) = \left\lfloor (\log b)^{-1} \right\rfloor \cdot g(n) - 1$ (or $h(n) = g(n) - 1$ if $b = 1$). Observe that (3.5) holds for h and b, for all integers n. Since $f \in o(g)$, it follows that $f(n) \leq h(n)$ for all but finitely many integers n.

Thus, we may apply Lemma 3.14 to obtain a language L' that is decided by CS_Σ with a binary advice of size g, but not by any Turing machine with an advice of size bounded by a function $f \in \mathcal{F}$ over an alphabet of any size. Since $CS_\Sigma \in DTIME(n)$, it follows that L' is in $\mathcal{D}/\mathcal{G}$, but not in $\mathcal{C}/\mathcal{F}$.

□

Theorem 3.17. *Let Σ be an input alphabet of size c. Let g be a function such that $g(n) \leq c^n$. Let $\mathcal{C}$ be a class of decidable languages. Let $\mathcal{F}$ be a countable class of integer-valued functions such that $f \notin \Omega(g)$ for every function $f \in \mathcal{F}$. Then $\{CS_\Sigma\}/g - \mathcal{C}/\mathcal{F} \neq \emptyset$.*

Proof. Let $\mathcal{B}$ be the set of positive integers. Let $\mathcal{H}$ be the class $\mathcal{F}$. Let $N = 0$ and $L = \emptyset$. Observe that $g(n) \leq c^n / \log 2$.

Let f be a function in $\mathcal{F}$ and b an integer in $\mathcal{B}$. Since $f \notin \Omega(g)$, there are infinitely many integers n such that $f(n) \leq \left\lfloor (\log b)^{-1} \right\rfloor g(n) - 1$. It follows that (3.5) holds for f and b, for infinitely many integers n. Since $\mathcal{H} = \mathcal{F}$, there is a function $h \in \mathcal{H}$, i.e., the function $h = f$, such that $f(n) \leq h(n)$ holds for all but finitely many integers n,

Thus, we can apply Lemma 3.14. It follows that there is a language L' that is in $\{CS_\Sigma\}/g$, but not in $\mathcal{C}/\mathcal{F}$.

□

Corollary 3.18. *Let f and g be integer-valued functions such that $f \notin \Omega(g)$. Then P/f is a proper subset of P/g.*

Proof. Follows from Theorem 3.17.

□

Corollary 3.19. *Let f and g be integer-valued functions such that $f \notin \Omega(g)$. Then $P/g - P/f$ is dense in P/g.*

Proof. Let $\mathcal{G}$ be the class of integer-valued functions such that $g' \in \mathcal{G}$ iff $g'(n) \leq g(n)$ for all but finitely many integers n. It follows from Proposition 3.15 that $P/\mathcal{G} - P/f$ is dense in P/g. Since an advice function bounded by a function g' in $\mathcal{G}$ can encode only a finite amount of extra information compared to an advice function of size g, this extra information can be stored in the finite control of a Turing machine instead. Thus, $P/\mathcal{G}$ is a subclass of P/g. Hence, $P/g - P/f$ is dense in P/g.

□

Karp and Lipton[22] originally claimed that P/f is a proper subset of P/g if $f(n) < g(n)$ holds for infinitely many integers n. Corollary 3.18 is somewhat weaker than this claim. However, for the class P, the separation cannot be improved, as the following Proposition shows.

Proposition 3.20. *Let $\mathcal{C}$ be a class of languages satisfying Property 3.11. Let g be an integer-valued function and $\mathcal{F}$ be a class of integer-valued functions with a function $f \in \mathcal{F}$ such that $f \in \Omega(g)$. Then $\mathcal{C}/g \subseteq \mathcal{C}/\mathcal{F}$.*

Proof. Let $f \in \mathcal{F}$ be a function such that $f \in \Omega(g)$. Let N and m be integers such that $f(n) \geq m^{-1} \cdot g(n)$ for all $n \geq N$. Suppose L is a language in $\mathcal{C}/g$. Let M be a Turing machine that decides L with an advice α of size bounded by g. Let b be the size of the advice alphabet that M uses. If $b > 1$, then we can encode the advice α using an alphabet of size b^m. This way, we can decrease the length of the advice by a factor of m (see Proposition 3.9). If $b = 1$, then we can encode the advice with an alphabet of size $2m$, which decreases the length of the advice logarithmically. In both cases, the encoded advice is bounded by f, which means that L can be decided by a Turing machine using an advice bounded by f that first decodes the advice and then simulates M on the tuple of input and advice value.

□

Observe that the decoding used in the proof can be done in linear time. This Proposition shows that Theorem 3.17 cannot be improved for countable classes of functions without further restrictions on the allowed advice alphabet sizes.

Corollary 3.21. *Let f and g be integer-valued functions such that $f \in \Omega(g)$. Then P/g is a subset of P/f.*

Proof. Follows from Proposition 3.20 and the fact that P satisfies Property 3.11.

□

3.2.3 Advice Alphabets of Bounded Size

In the statement of Theorem 3.16, we made no restriction on the allowed advice alphabet size, which implied that there were no bounds on the sizes. If we restrict the possible sizes, then we can improve the result of the Theorem. Since hardware implementations impose practical bounds on the sizes of the alphabets used by machines, this assumption is not unreasonable.

Theorem 3.22. *Let Σ be an input alphabet of size c and Ω an advice alphabet of size $d \geq 2$. Let g be an integer-valued function such that $g(n) \leq c^n / \log d$. Let $\mathcal{F}$ be a countable class of integer-valued functions such that for every function $f \in \mathcal{F}$, the inequality $f(n) \leq g(n) - 1$ holds infinitely often. Then, there is a language that can be decided by a Turing machine with an advice of size g over the alphabet Ω, but not by any Turing machine using an advice bounded by f over an advice alphabet of size d or less.*

Proof. Let $\mathcal{B} = \{1, \ldots, d\}$. Let $\mathcal{H}$ be the class $\mathcal{F}$. Let $N = 0$ and $L = \emptyset$.

Let f be a function in $\mathcal{F}$ and b an integer in $\mathcal{B}$. Since $f(n) + 1 \leq g(n)$ holds infinitely often and $b \leq d$, it follows that (3.5) holds for f and b, for infinitely many integers n. Note that $f \in \mathcal{H}$.

Thus, we can apply Lemma 3.14, to obtain a language L' that can be decided by a Turing machine for $CS_{\Sigma,\Omega}$ with an advice of size g over the alphabet Ω, but not by any Turing machine using an advice bounded by a function f in $\mathcal{F}$ over an alphabet of size d or less.

□

Thus, the original statement made by Karp and Lipton[22] holds if the advice alphabets are bounded in size. For instance, when using only binary advices, P/f is a proper subset of P/g whenever $f(n) < g(n)$ holds infinitely often and g is exponentially bounded. The statement remains true if e.g. a 10, 16 or 26 letter alphabet are used.

Corollary 3.23. *Suppose Turing machines may only use advice functions with advice alphabets of size bounded by an integer $d \geq 2$. Let c be an integer and let f and g be integer-valued functions such that $f(n) < g(n)$ holds infinitely often and $g(n) \leq c^n / \log d$ for all n. Then P/f is a proper subset of P/g. Furthermore, $P/g - P/f$ is dense in P/g.*

Proof. This follows from Theorem 3.22. The last statement follows from Proposition 3.15 (see also the proof of Corollary 3.19).

□

This non-uniform separation cannot be improved further, since any finite amount of extra information in the advice function could also be coded into the machine itself.

3.3 Conclusions

Simple counting arguments, the possibility to enumerate all possible interesting combinations of machine and advice and diagonalizing arguments are the ingredients that were essential to the central theorems of the Chapter. These three techniques are very useful and will be used multiple times in this thesis. To enumerate all the interesting cases, the number of interesting cases should be countable. This can in some cases be a restriction, but we can often deploy a countable subset of interesting cases which covers all interesting cases in some way. An example of this is seen in the use of the classes $\mathcal{F}$ and $\mathcal{H}$ in Theorem 3.16. Here, $\mathcal{F}$ was a possibly uncountable class of integer-valued functions used as advice bounds. The class $\mathcal{H}$ was constructed such that every interesting combination of advice bound $f \in \mathcal{F}$ with a machine was covered by the combinations of all functions from $\mathcal{H}$ with the same machine.

Hermo and Mayordomo[18] gave a proof for Theorem 3.16 that is based on Kolmogorov complexity theory. We should note here that the foundations of Kolmogorov complexity theory rely on the same kind of counting arguments. The role of the advice alphabet sizes was obvious in (3.5), while the use of Kolmogorov complexity theory did not make clear that the results could be improved by restricting the allowed sizes of the advice alphabets. This illustrates that while abstract theories can greatly reduce the complexity of proofs, one should not be afraid to *get one's hands dirty* in order to obtain improvements over the results that an abstract framework so easily provides.

For advice sizes, the picture is now complete. Hermo and Mayordomo[18] proved that $P/\mathcal{F}$ is properly contained in P/g if $f \in o(g)$ for every $f \in \mathcal{F}$. We showed that P/g is contained in $P/\mathcal{F}$ if there is a function $f \in \mathcal{F}$ such that $f \in \Omega(g)$. Karp and Lipton[22] originally claimed that P/f was properly contained in P/g if $f(n) < g(n)$ for infinitely many integers n. While we showed that this is impossible for unbounded advice alphabet sizes, we also showed that the claim holds if the allowed advice alphabets are bounded by a constant. On the other hand, if $f(n) \geq g(n)$ for all but finitely many integers n, then the advices for the finitely many integers n for which $f(n) < g(n)$ can be encoded into the finite control of a Turing machine. Such a machine can then use an advice of length f to decide languages that need an advice of length g. So in this case, P/g is contained in P/f.

The careful reader may have noticed that the advices of size g were all over an alphabet with more than one letter. For the unbounded case, this is crucial, since a one-letter advice of size g can be encoded into a binary string of length $O(\log g)$. However, if machines may only use advices over one-letter alphabets, then Theorem 3.22 remains valid. In this case, the counting argument can be simplified by observing that there are $f(n)$ different advices bounded in length by f and $g(n)$ different advices bounded in length by g.

The class $\mathcal{F}$ in Theorem 3.22 is a countable class. This is necessary to enumerate all the interesting cases. While the result can be changed to include classes of functions that are bounded by a countable class of functions, it remains an open question if the result holds for all classes of functions $\mathcal{F}$ such that $f(n) < g(n)$ for infinitely many integers n for every $f \in \mathcal{F}$.

CHAPTER 4

Sequences

A natural way to model evolving systems is to use sequences. Sequences play an important role in this thesis. In this Chapter, we will take a closer look at sequences of machines and their relation to advice functions. The machines that we consider can be of any fixed type, ranging from finite automata to Turing machines or other models of computation.

Let $\mathcal{M}$ be a class of machines with an input alphabet Σ. Let M_1, M_2, ... be a sequence of machines in $\mathcal{M}$. We view such a sequence of machines as one single computing entity which has the ability to decide subsets of $\Sigma^\star$ by utilizing the components of the sequence. A set L is said to be decided by the sequence iff machine M_n decides $L \cap \Sigma^n$ for every n.

We classify sequences of machines based on the description sizes of the machines that make up the sequence. Using this complexity measure, we define complexity classes for sequences of machines. Our first goal is to state a theorem which relates the complexity of a sequence of machines to the complexity of a single machine with an advice mechanism of a suitable complexity. An example of such a theorem was first given by Karp and Lipton[22], who showed that the class of languages decided by (sequences of) polynomially sized Boolean circuits equals the class of languages decided by a Turing machine with polynomial advice. We will state a more general version of the theorem, i.e., we don't fix the machine models to be used in the characterization in advance. The theorem given by Karp and Lipton is then a consequence of this general version. We will give several more instances of the theorem, e.g. for sequences of resource-bounded Turing machines.

In setting up the framework for the result, a conflict of interests arises. On the one hand, we want a theorem that is as general as possible, which means that we can only use aspects of machines which are shared between most (if not all) models of computation. On the other hand, in order to achieve this, we need formal definitions of these aspects in order to actually apply the results. We compromise by defining *meta-properties* of models of machines, which we use as formal definitions.

When we want to apply the equivalence result to actual machine models, we just have to translate these meta-properties into actual properties of the machines, using the definitions of the class of models to which they belong. As an example, we introduce the concept of a machine calling another machine, which is implemented by Turing machines by integrating the transition function of the second into the transition function of the first. These meta-properties build on the concepts that were introduced in Chapter 2.

The Chapter begins with a brief introduction to the fundamental properties of machine models. Using these properties, the general theorem is given. The theorem states the relationship between sequences of machines and machines with an advice mechanism. After this, the theorem is applied to several well-known and often used machine models. Next, we explore the possibilities of using different ways of describing machines as advice functions. By using different descriptions, we can generate more efficient or more powerful advice functions, depending on the results we want to achieve. This idea is applied to show several more equivalences where we specifically aim at sequence-based characterizations of the classes $LOGSPACE/poly$ and $P/poly$, within our general framework. Then, one of the fundamental properties is explored in more detail, and a more concrete property is defined which implies this fundamental property. Finally, some conclusions are given for the theory of sequences in the context of modeling evolving systems.

4.1 General Aspects of Machines

Let $\mathcal{M}_1$ and $\mathcal{M}_2$ be machine models. Given a sequence of machines of type $\mathcal{M}_1$, we will construct a machine M of type $\mathcal{M}_2$ and an advice function, such that M uses the advice function to decide the same language as the sequence of machines. This implies that M should be able to handle multiple input lengths. We assume that $\mathcal{M}_1$ and $\mathcal{M}_2$ satisfy certain conditions, listed as Properties 4.1 and 4.2.

Property 4.1. Given a sequence of machines of type $\mathcal{M}_1$, there is an encoding mechanism e for the machines in the sequence and a machine M of type $\mathcal{M}_2$ that takes as input any string x and as advice any string w, as long as w is the description of a machine in the sequence, such that M simulates the operation of the machine encoded by w on the input x. Furthermore, there are functions $T, S : \mathbb{N}^5 \to \mathbb{N}$ such that if the machine encoded by w is of size m' and runs in time T' and space S', then M uses $O(T(|x|, |w|, S', T', m'))$ time and $O(S(|x|, |w|, S', T', m'))$ space.

Property 4.2. For every machine M_2 of type $\mathcal{M}_2$, every integer n and every string w, there is a machine M_1 of type $\mathcal{M}_1$, such that for every string x of length n the machine M_1 simulates the operation of M_2 on the input x using advice w, if M_1 has the strings x and w stored as data. Furthermore, there are functions $m, T, S : \mathbb{N}^5 \to \mathbb{N}$, such that if M_2 is of size m' and uses time T' and space S', then M_1 has a size of $O\left(m(n, |w|, S', T', m')\right)$ and runs in time $O(T(n, |w|, S', T', m'))$ and space $O(S(n, |w|, S', T', m'))$.

Other important properties of machine models we need are:

Property 4.3. Machines of type $\mathcal{M}$ can handle inputs of all lengths.

Property 4.4. Machines of type $\mathcal{M}$ can store any string of length n as mutable data using at most $O(n)$ space.

Property 4.5. Every string of length n can be stored by a machine of type $\mathcal{M}$ as immutable data using at most $O(n)$ size.

Note that storing information in the immutable data of a machine essentially changes the machine itself. Thus, when a string that is stored in the immutable data is altered, the machine that stores the string is altered as well.

Property 4.6. Any machine M of type $\mathcal{M}$ can be called by any other machine M' of type $\mathcal{M}$. The machine M performs the actions on (part of) the contents of the data of the calling machine M'. Suppose M has size m and runs in time T and space S. If M' calls M, then the size of M' is increased by $O(m)$, and M' uses an additional $O(T)$ time and $O(S)$ space.

Definition 4.7. *Let M be a machine that has among its data a description of an object o. We say that M has a* pointer *to o if M is in a configuration which contains o.*

For example, the machine M may have o on a tape, and a tape head positioned at the beginning of o. Or, M may have o stored using states, and be in a state that holds o. As machine models get more exotic, more examples could be dreamed up.

4.1.1 Sequences of Machines

An important goal of this Chapter is to state a general version of an equivalence result that links sequences of machines to machines with advice. The terminology for machines with advice mechanisms has already been explored in Chapter 3. Here, we define some terminology for sequences of machines.

Definition 4.8. *Let Σ be an alphabet. A sequence of machines is said to* decide *a language $L \subseteq \Sigma^\star$ if the n-th machine in the sequence decides the subset $L \cap \Sigma^n$ as a subset of Σ^n.*

Definition 4.9. *A sequence of machines is said to be of* size *m for an integer-valued function m, if the n-th machine is of size $m(n)$.*

Definition 4.10. *Given an encoding mechanism e, a sequence of machines is said to have* descriptions of size *d for an integer-valued function d, if the n-th machine can be encoded using e within a string of size $d(n)$.*

Definition 4.11. *A sequence of machines is said to be of* time *(or* space*)* complexity *T for an integer-valued function T, if the n-th machine uses at most time (or space) $T(n)$ for its inputs of length n.*

Alternatively, we say that a sequence of machines has a running time (or a space usage) of T, or that a sequence of machines runs in time (or space) T.

4.2 A General Equivalence Result

With the terminology of the previous section in place, we can give the general version of the results that links sequences of machines to machines with advice.

Theorem 4.12. *Let $\mathcal{M}_1$ and $\mathcal{M}_2$ be two machine models. Let M_1, M_2,... be a sequence of machines of type $\mathcal{M}_1$ with descriptions of size f, using an encoding mechanism e. Suppose that Property 4.1 holds. Let L be the language decided by the sequence of machines. Then, L can be decided by a machine M of type $\mathcal{M}_2$ with an advice of size f. Furthermore, if machine M_n runs in time T_n and space S_n, then M runs in time $O(T(n, f(n), S_n, T_n))$ and space $O(S(n, f(n), S_n, T_n))$.*

Proof. Let w_n be the description of M_n. By Property 4.1, there is a machine M of type $\mathcal{M}_2$ running in the desired time and space that takes any string x as input and $w_{|x|}$ as advice and simulates $M_{|x|}$ on x. Thus, M accepts x with advice $w_{|x|}$ iff $M_{|x|}$ accepts x iff $x \in L$. Hence, L is decided by M using the advice function $n \mapsto w_n$ of size f.

□

Theorem 4.13. *Let $\mathcal{M}_1$ and $\mathcal{M}_2$ be two machine models satisfying Property 4.2. Suppose that $\mathcal{M}_1$ satisfies Property 4.5 and 4.6. Let L be a language and f an integer-valued function. If L can be decided by a machine M of type $\mathcal{M}_2$, of size m' running in time T' and space S', with an advice of size f, then L can be decided by a sequence of machines of type $\mathcal{M}_1$ of size $O(f(n)+m(n, f(n), S', T', m'))$. Furthermore, the sequence of machines runs in time $O(T(n, f(n), S', T'))$ and space $O(S(n, f(n), S', T'))$.*

Proof. Let M be a machine of type $\mathcal{M}_2$ that decides L with an f-bounded advice function α. Suppose M is of size m' and runs in time T' and space S' Let w be a string. By Property 4.2, there is a machine $M_{n,w}$ of type $\mathcal{M}_1$ such that for every string x of length n the machine $M_{n,w}$ simulates the operation of M on input x using advice w, if $M_{n,w}$ has the strings x and w stored as data.

Construct the machine M_n of type $\mathcal{M}_1$ as follows: M_n has the string $w = \alpha(n)$ stored in its immutable data (using $O(f)$ size by Property 4.5). Given an input x of length n, the machine M_n calls $M_{n,w}$. By Property 4.6, this can be done with an additional size of $O(m(n, f(n), S', T', m'))$, in the desired time and space. Since the call on $M_{n,w}$ has x and w stored as data, M_n accepts x iff M accepts x using advice w. Summing up, the total size needed for M_n is $O(f(n) + m)$. Observing that M_n decides $L \cap \Sigma^n$ finishes the proof.

□

Due to the abstract nature of the results, they cannot be applied straightaway. One must first express the abstract properties of a machine model using the concrete definitions of the model. Doing this differs from one model to the other.

As an example we show how Theorems 4.12 and 4.13 imply Karp and Lipton's result for sequences of Boolean circuits[22]. The result follows by taking the Boolean circuit model for $\mathcal{M}_1$ and the Turing machine model for $\mathcal{M}_2$ and determining the correct functions used in Properties 4.1 and 4.2.

Corollary 4.14 (Karp and Lipton). *A language L can be decided by a sequence of polynomially sized Boolean circuits iff L is in $P/poly$.*

Proof. We use the standard encoding for Boolean circuits. Then, a circuit of size m can be described using a string of length$O(m \log m)$. The Circuit Value Problem (CVP, as defined in Balcázar et al.[1]) takes a binary string x and a description of a Boolean circuit w and determines if x is accepted by the circuit described by w. Thus, a Turing machine that solves the CVP satisfies Property 4.1. The CVP can be solved by a Turing machine in quadratic time using linear space (see Balcázar et al.[1]). It follows from Theorem 4.12 that a sequence of polynomially sized circuits can be simulated by a Turing machine in polynomial time using advice of polynomial size.

Conversely, observe that Boolean circuits have no implementation of mutable data, so the space complexity of the sequence can be ignored. Note that Boolean circuits satisfy Property 4.5 and 4.6. It is well-known that a Turing machine running in time T' can be simulated by a Boolean circuit of size $m \in O((T')^2)$, with a depth $T \in O(T')$ (see Balcázar et al.[1]). Thus Property 4.2 is satisfied and Theorem 4.13 can be applied.

However, a Turing machine incorporates the advice in its input, whereas the Boolean circuit has to embed the advice using constant gates. Therefore, a Turing machine running in time T' with an advice of size f can be simulated by a sequence of Boolean circuits of size $O(f(n)+(T'(n+f(n)))^2)$. Since T' and f are polynomial by the assumptions of the Corollary, the sequence is polynomially sized.

□

4.2.1 Sequences of Resource-Bounded Turing Machines

An interesting example of the equivalence result is the case when $\mathcal{M}_1$ and $\mathcal{M}_2$ are both taken to be the Turing machine models with suitable complexity measures. First, sequences of uniformly time-bounded Turing machines are considered.

Proposition 4.15. *Let f be a integer-valued function. Let e be an encoding mechanism. If a language L is decided by a sequence of Turing machines running in polynomial time (or logarithmic space), with descriptions of size f using the encoding e, then L can be decided by a Turing machine running in polynomial time (or logarithmic space), with an advice of size f.*

Proof. We can use a universal Turing machine M that takes descriptions of Turing machines using the encoding e to satisfy Property 4.1. The machine M uses the descriptions of the machines in the sequence as an advice function, so M uses an advice of size f. Suppose the n-th machine uses time T_n and space S_n. Then M can simulate the n-th machine of the sequence in time $T \in O(T_n^2 f(n))$ and space $S \in O(S_n)$.

Observe that M has both the input and the advice on its input tape. Let $n' = n + f(n)$. Then T is polynomially bounded as a function of n' if T_n is polynomial in n. Similarly, S is logarithmically bounded in n' if S_n is logarithmic in n. Thus, by Theorem 4.12, a sequence of Turing machines running in polynomial time (or logarithmic space), with descriptions of size f, can be simulated by a

Turing machine running in polynomial time (or logarithmic space), with an advice of size f.

□

Corollary 4.16. *Let m be an integer-valued function and k an integer. If a language L is decided by a sequence of k-tape Turing machines of size f running in polynomial time (or logarithmic space), then L can be decided by a Turing machine running in polynomial time (or logarithmic space) with an advice of size $O(f \log f)$.*

Proof. Since k is a constant and the alphabet is fixed, a k-tape Turing machine of size f can be described with a string of length $O(f \log f)$. Then the result follows from Proposition 4.15.

□

To prove a converse result and simulate Turing machines with advice by sequences of Turing machines, we first prove that Property 4.2 holds.

Proposition 4.17. *Let $\mathcal{M}_1$ and $\mathcal{M}_2$ be the class of Turing machines. Then, Property 4.2 holds, with the functions $m = m'$, $T = T'$ and $S = S'$.*

Proof. Consider an arbitrary Turing machine M_2 of type $\mathcal{M}_2$, an integer n and a string w. Since M_2 is also of type $\mathcal{M}_1$, the same machine can be used to simulate M_2. It follows that the machine M_1 is of the same size and runs in the same time and space.

□

Proposition 4.18. *Let f be an integer-valued function. If a language L is decided by a Turing machine running in polynomial time (or logarithmic space) with an advice of size f, then L can be decided by a sequence of one-tape Turing machines of size $O(f)$ running in polynomial time (or logarithmic space).*

Proof. Let M be a Turing machine machine of size m' that decides L in time T' and space S' with an advice of size f. It follows from Proposition 4.17 that Property 4.2 is satisfied with $m = m'$, $T = T'$ and $S = S'$. Turing machines satisfy Property 4.5 and 4.6, so by Theorem 4.13, the language L can be decided by a sequence of Turing machines of size $O(m' + f)$, running in time $O(T)$ and space $O(S)$. These machines can be converted to Turing machines with one tape, using time $O(T^2)$ and space $O(S)$ (see Hartmanis and Stearns[16]).

However, the machine M has both the input and the advice on its input tape, so T' is a function of $n+f(n)$, while T is a function of n. Thus, the running time is $O((T'(n+f(n)))^2)$. Since T' and f are polynomial, it follows that the running time of the sequence of machines is polynomial too. Similarly, the space usage of the sequence of machines is $O(S'(n + f(n)))$, which is logarithmic if S' is logarithmic and f is polynomial. Furthermore, M is a single machine, so m' is a constant. This implies that the sequence consists of Turing machines that are of size $O(f)$ and have a fixed number of tapes.

□

Corollary 4.16 and Proposition 4.18 enable us to give simple characterizations of the computational power of sequences of time- or space- bounded Turing machines in terms of advice classes.

Corollary 4.19. *Let k be an arbitrary positive integer. A language L belongs to $P/poly$ iff L can be decided by a sequence of polynomially sized k-tape Turing machines running in polynomial time.*

Corollary 4.20. *Let k be an arbitrary positive integer. A language L belongs to $LOGSPACE/poly$ iff L can be decided by a sequence of polynomially sized k-tape Turing machines running in logarithmic space.*

Proof (for Corollary 4.19 and 4.20). Observe that if an integer-valued function m is a polynomial, then any function in $O(m \log m)$ is polynomially bounded. The results follow from Corollary 4.16 and Proposition 4.18.

□

4.3 Sequences of Machines with Bounded Description Sizes

If we compare Proposition 4.18 to Corollary 4.16, we see that the advice size blows up logarithmically. This is a consequence of Property 2.3, which states that an arbitrary machine of size m needs a description length of at least $O(m \log m)$. However, the machines that we constructed in the proof of Theorem 4.13 are not arbitrary machines. We can use this fact to our advantage to find an encoding such that these particular machines can be described more efficiently.

Proposition 4.21. *There is an encoding mechanism e such that the sequence of machines of Theorem 4.13 have descriptions of size $O(f + m \log m)$.*

Proof. We need to find an encoding such that M_n can be described using a string of length $O(f + m \log m)$, for every n. Observe that M_n consists basically of two parts, a part which stores $w = \alpha(n)$ and a part which calls $M_{n,w}$ on w and the input. We encode $M_{n,w}$ with our original encoding. This requires a string of length $O(m \log m)$. To complete the description, we need a description of a machine that stores w and calls $M_{n,w}$. This can be done with a string of length $O(|w|)$. Since $|w| = f(n)$, the machine M_n can be described with a string of length $O(f + m \log m)$.

A general machine of type $\mathcal{M}_2$ is encoded using e as follows: if the machine consists of storing data and calling another machine, then we use the above description. Otherwise, we use the original encoding.

□

As an example, we can apply Proposition 4.21 to the case of Turing machine models with suitable complexity measures.

Proposition 4.22. *Let f be an integer-valued function. If a language L is decided by a Turing machine M running in polynomial time (or logarithmic space) with an advice of size f, then L can be decided by a sequence of Turing machines running in polynomial time (or logarithmic space), with descriptions of size $O(f)$.*

Proof. Let M have size m'. By Proposition 4.18, the sequence of Turing machines runs in polynomial time (or logarithmic space). By Proposition 4.21, the machines in the sequence have descriptions of size $O(f + m \log m)$. It follows from 4.17 that $m = m'$. As m' is a constant, we conclude that the machines in the sequence have descriptions of size $O(f)$.

□

Corollary 4.23. *Let f be an integer-valued function. A language L belongs to $P/O(f)$ iff L can be decided by a sequence of Turing machines running in polynomial time, with descriptions of size $O(f)$.*

Corollary 4.24. *Let f be an integer-valued function. A language L belongs to $LOGSPACE/O(f)$ iff L can be decided by a sequence of Turing machines running in logarithmic space, with descriptions of size $O(f)$.*

Proof (of Corollary 4.23 and 4.24). The results follow from Proposition 4.15 and 4.22.

□

4.4 Sequences of Multi-Head Finite Automata

Multi-head finite automata are a common, powerful system model. We will show how the uniform framework for the equivalence result can be used to characterize the computational power of sequences of multi-head finite automata of various kinds. We will show the following result, based on familiar techniques from automata theory brought together in one framework.

Theorem 4.25. *Let k and k' be positive integers. Then, the following statements about a language L are equivalent:*

(i) L is in $LOGSPACE/poly$,
(ii) L can be decided by a sequence of polynomially sized logarithmic space k-tape Turing machines.
(iii) L can be decided by a sequence of polynomially sized deterministic finite automata with k' heads,
(iv) L can be decided by a finite automaton with one input head and an advice of polynomial size.

The equivalence between *(i)* and *(ii)* has already been shown in Corollary 4.20. Next, we will complete the proof by showing the equivalence between *(i)* and *(iii)* and the implications "*(i)* $\Rightarrow$ *(iv)*" and "*(iv)* $\Rightarrow$ *(iii)*".

4.4.1 Turing Machines with Advice

Let $\mathcal{M}_1$ be the model of finite automata and $\mathcal{M}_2$ the model of Turing machines. To show that "*(i)* $\Rightarrow$ *(iii)*" of Theorem 4.25 holds, we use a common technique to simulate Turing machines on inputs of a fixed length with finite automata: Turing machines execute their program in a discrete fashion, thus the operation of such a

machine can be described by a sequence of configurations the machine is in from the initial stage to the moment the machine halts. See Balcázar et al.[1] for details on configurations. The simulation starts in the initial configuration and proceeds by moving from configuration to configuration, accepting iff the final configuration is an accepting configuration. This technique relies on the fact that the space usage of the Turing machine is bounded, so there are only finitely many configurations.

Proposition 4.26. *If a language L is decided by a Turing machine running in logarithmic space, with an advice of polynomial size, then L can be decided by a sequence of polynomially sized two-way finite automata with one head.*

Proof. A configuration of a Turing machine consists of the state the machine is in, the contents of the work-tapes and the position of the tape heads. Additionally, if the input head is on a symbol of the advice, the corresponding configuration also contains this symbol.

Let M be a Turing machine of size m' with k work tapes that decides L in space S', using an advice of size f. Let c be the size of the tape-alphabet that M uses. In this case, the number of configurations for inputs of length n is $O(m'c^{kS'(n+f(n))}\left(S'(n+f(n))\right)^k(n+f(n)))$. Since the advice is fixed for inputs of length n, storing parts of the advice in the configurations does not increase the number of configurations. Thus, M can be simulated on inputs of length n by a finite automaton M_n of size $m(n) \in O(m'c^{kS(n+f(n))}\left(S(n+f(n))\right)^k(n+f(n)))$. Since S' is logarithmic, m is polynomial.

Let x be an input of length n and w the advice for x. If the input head is on a symbol of x for a configuration, then M_n reads the corresponding symbol from its input tape. This is automatically the case in the initial configuration, the transition function ensures that this is always the case. The transition function of M_n mimics the transition function of the sequence of configurations (again, see Balcázar et al.[1] for details). There is a small catch, namely, the Turing machine has $\langle x, w\rangle$ on its input tape, whereas the finite automaton has only x. The transition function works around this by using the symbol of w that is stored in the configuration when the configuration tries to access a tape-square beyond the end of the tape. This completes the proof.

□

The next Proposition shows that "*(iii) ⇒ (i)*" of Theorem 4.25 holds.

Proposition 4.27. *If a language L is decided by a sequence of polynomially sized two-way deterministic finite automata with k' heads, then L can be decided by a Turing machine running in logarithmic space, with an advice of polynomial size.*

Proof. A finite automaton can be viewed as a Turing machine without work-tapes. So, if we use the standard encoding for Turing machines to describe finite automata, then a universal Turing machine M satisfies Property 4.1. For a polynomially sized Turing machine, the description has polynomial size, thus M uses an advice of polynomial size. Let $m'(n)$ be the size of the n-th automaton of the sequence. For a Turing machine without work-tapes, only the state and the positions of the k' heads need to be stored, so M uses space $S \in O(\log m'(n) + k' \log n)$. Since m' is polynomial and k' is fixed, S is logarithmic in the input.

□

An interesting consequence is that the number of tape heads used in a sequence of polynomially sized finite automata has little impact on the computational power of the sequence.

Corollary 4.28. *For any integer k, sequences of polynomially sized two-way deterministic finite automata with k heads can be simulated by sequences of polynomially sized two-way deterministic finite automata with one head.*

Proof. This follows directly from Propositions 4.27 and 4.26. □

Let L be a language over an alphabet Σ. Suppose a Turing machine decides L using linear space and an exponentially bounded advice. It follows from the proof of Proposition 4.26 that L can be decided by a sequence of exponentially sized finite automata. However, in such a case, it may be more efficient to decide the finite set $L \cap \Sigma^n$ directly, without simulating the Turing machine. This can be done by a one-way finite automaton with $|\Sigma|^n$ states even for $|\Sigma| = 1$. In fact, this can be done with any language.

Theorem 4.29. *Any language can be decided by a sequence of exponentially sized one-way deterministic finite automata.*

Note that for sparse languages, polynomially sized automata will do.

4.4.2 Multi-Head Finite Automata with Advice

When we have the freedom to choose the descriptions, we can add all sorts of helpful information in the description to help with the simulation of the described machines. This allows us to simulate sequences machines with less powerful machines that were unable to simulate the machines of the sequence using the standard encoding. We illustrate this with finite automata with advice.

Using finite automata in a sequence of machines is very straight-forward. Using a finite automaton to simulate a sequence of machines on the other hand, is somewhat more complicated. A finite automaton with advice is a two-way finite automaton with two heads for its input tape. Alternatively, one may view a finite automaton with advice as an automaton with two tapes, one containing the input, the other containing the advice for the length of the input. The two heads are necessary, since the automaton cannot remember the position of a head on the tape, so it cannot move from input to advice and back using just one head. More generally, on a multi-head finite automaton with advice, one of the heads is designated as the *advice head*, the other heads are regular *input heads*. We assume that the advice is stored on the same tape as the input, as it simplifies the constructions somewhat.

Given a description of an automaton, a simulating automaton needs to move from state to state on this description. The automaton can only use its finite control to determine if it has found the correct state on the description. When the number of states of the simulated machine is unbounded, the simulating machine will run out of states to test this. Thus, in general, a finite automaton with advice cannot simulate a sequence of automata.

The argument suggests that if there is a description for automata such that the possible destination states of any transition can be located with a finite amount of information, then these automata can be simulated with a finite automaton. We will show that this is indeed the case.

Property 4.30 captures the conditions a sequence of finite automata needs to satisfy in order to be simulated by a finite automaton with advice.

Property 4.30. Consider a sequence of finite automata with k heads, for some integer k. There is an encoding mechanism e for the automata in the sequence and finite automata M_{init} and M_{step} with $k+1$ heads such that for an automaton M_n in the sequence, the description of M_n using e has the following properties:

- The description is based on a transition-list,
- the initial state of M_n can be found by M_{init} if its input tape contains the description of M_n,
- for each state q of M_n, every input x of M_n, and every possible placement of the k heads on x, which cause the transition function to output state q', if the finite automaton M_{step} contains x, as well as the description of M_n on its input tape, a tape head positioned at the beginning of q on this description and the other k heads placed at the positions of the input, then M_{step} moves a tape head to the beginning of q' and the remaining k heads to the positions corresponding to the transition.

Now, take $\mathcal{M}_1$ as the class of finite automata satisfying Property 4.30 and $\mathcal{M}_2$ as the class of finite automata. Proposition 4.31 shows that a sequence of automata that satisfies Property 4.30 for an encoding mechanism e satisfies Property 4.1.

Proposition 4.31. *Consider a sequence of finite automata with k heads running in time T' with descriptions of size f, using an encoding mechanism e that satisfies Property 4.30. This sequence satisfies Property 4.1 for this encoding mechanism, with $T \in O(T' \cdot f)$ and $S = 0$. Furthermore, if the automata M_{init} and M_{step} are of size m_{init} and m_{step} respectively, then the finite automaton M is of size $O(m_{\text{init}} + m_{\text{step}})$.*

Proof. Let M_1, M_2, ... be the finite automata in the sequence. We construct a finite automaton M with $k+1$ heads and an advice mechanism that simulates the sequence. For inputs of length n, the advice contains the description of M_n. The advice head is the $(k+1)$-st head, so M may call the automata from Property 4.30.

The simulating automaton M starts by moving the k input heads to the first symbol of the input. This way, the head positions of M correspond to the head positions of M_n before the first transition occurs. Now, M calls M_{init} to move the remaining head to the beginning of the initial configuration. This can be done in $O(f)$ steps. After this, M repeatedly calls M_{step} to find the next state of the computation, moving the input heads to the correct positions along the way, until a final state is reached. Each call takes at most $O(f)$ time.

Since finite automata satisfy Property 4.6, M is of size $O(m_{\text{init}} + T' \cdot m_{\text{step}})$. However, this direct construction does not yield a finite automaton since T' is not a constant. Instead, after each call of M_{step}, the finite automaton M checks if the

state M_n is in is a final state. This can be done with $O(1)$ states. But then, M can just use the same copy of M_{step} in its finite control over and over again. Thus, M is of size $O(m_{\text{init}} + m_{\text{step}})$.

The finite automaton M simulates M_n using $O(T' \cdot f)$ time. Note that finite automata have no implementation of mutable data, so $S = 0$. Thus, Property 4.1 is satisfied with $T \in O(T' \cdot f)$ and $S = 0$.

□

By combining the results in the subsection, we obtain the following result.

Proposition 4.32. *Let L be a language decided by a sequence of finite automata with k heads and descriptions of size f satisfying Property 4.30. Then L can be decided by a finite automaton with k input heads and an advice of size f.*

Proof. The result follows from Proposition 4.31 and Theorem 4.12.

□

Sequences of Finite Automata with Bounded Bandwidth

As an example, we apply Proposition 4.31 to sequences of finite automata with bounded bandwidth. A matrix has bandwidth b if b is the smallest integer for which the entries with index (i, j) are empty whenever $|j - i| \geq b$. A finite automaton has *bandwidth* b if there is an ordering of the states such that the transition matrix for this ordering has bandwidth b.

Example 4.33. Consider a sequence of finite automata with k heads and bandwidth bounded by b, operating over an alphabet of size c. We construct a description of the automata in the sequence that satisfies Property 4.30. We order and list the states of an automaton such that the transition matrix has bandwidth b, starting with the initial state. From any state in this list, the next states all lie within the b previous and b next states. Thus, we describe the next state with an integer i between $-b$ and b.

An automaton of size m has mc^k transitions and each transition needs a string of length $\log c^k + \log b$. It follows that an automaton with m states can be described with a string of length $O(mc^k \cdot \log(c^k b))$.

Since the first state in the list is the initial state, the finite automaton M_{init} halts immediately, using just one state.

To determine the next transition, the finite automaton M_{step} needs the symbols under the heads. For an automaton with k heads, $O(k)$ states are needed to find the transition corresponding to the k symbols under the heads. The head movements, which are listed next, can be applied with $O(1)$ states. The next state is determined by parsing the integer i belonging to this transition. This can be done with a full tree of size $O(b)$.

Now, if we place distinguishing markers between every state and its transitions, then we can move in the list in the proper direction until we encounter the i-th marker to find the next state. This can be done with another i states. To sum up, M_{step} uses $O(k + b^2)$ states.

Proposition 4.34. *If a language L over an alphabet of size c can be decided by a sequence of automata of size f with k heads and bandwidth bounded by an integer b, then L can be decided by a finite automaton M of size $O(k + b^2)$ with k input heads and an advice of length $O(f(n)c^k \log(c^k b))$.*

Proof. We use the description and the automata M_{init} and M_{step} from Example 4.33. By Proposition 4.31, the automaton M is of size $O(k + b^2)$. By Proposition 4.32, the automaton M decides L using the descriptions of the automata in the sequence as advice.

Observe that we don't use the descriptions of the states, we only need the descriptions of the transitions and the markers. Thus, the length of the description of the n-th automaton in the sequence is $O(f(n)c^k \cdot \log(c^k b))$.

□

Since c, k and b are constants, it follows that a sequence of k-head finite automata of size f and bandwidth bounded by an integer b can be simulated by a finite automaton with k input heads and an advice of size $O(f)$.

Corollary 4.35. *If a language L can be decided by a sequence of k-head polynomially sized finite automata with bandwidth bounded by an integer b, then L can be decided by a finite automaton with k input heads and a polynomially sized advice.*

Corollary 4.36. *If a language L can be decided by a sequence of k-head logarithmically sized finite automata with bandwidth bounded by an integer b, then L can be decided by a finite automaton with k input heads and a logarithmically sized advice.*

Log-Space Turing Machines with Advice

As another example, consider the sequence of finite automata that is constructed in Proposition 4.26. This sequence satisfies Property 4.30. We use this fact to show that "*(i)* $\Rightarrow$ *(iv)*" of Theorem 4.25 holds.

Proposition 4.37. *Let L be a language in $LOGSPACE/poly$. Then, L can be decided by a sequence of finite automata with one head and polynomially sized descriptions that satisfy Property 4.30.*

Proof. Let L be decided by a Turing machine M in logarithmic space, using an advice of polynomial size. By Proposition 4.26, L can be decided by a sequence of finite automata. We modify the automata in the sequence, by adding a few extra states to the automata, to simplify the order of the states. Since the transition function remains unaltered, this new sequence decides the same language L. We construct a description for this sequence of automata that satisfies Property 4.30.

Let m' be the number of states of the Turing machine M. Let M_n be the n-th automaton in the sequence. A state of M_n corresponds to a tuple $(q, v_1, \ldots, v_k, l, a_l)$, where q is a state of M, l is the unary encoding of the position of the advice head of M, a_l is the symbol under the advice head and v_i is a string representing the contents of the i-th work tape of M.

The string v_i consists of symbols from the input alphabet plus the blank symbol and the extra symbol $\sharp$, which denotes the position of the head. Since the Turing machine M works in logarithmic space, there is an integer d such that the work tapes use at most $d \log n$ cells. Thus, the tape contents of M fit into strings of length $d \log n$. For every $1 \leq i \leq k$, the string v_i has length $d \log n$ (shorter tape contents get padded with blank symbols). A string v_i is *valid* if it satisfies the following constraints:

- the string contains exactly one $\sharp$ symbol,
- the string does not have any blank symbols before a $\sharp$ symbol,
- the string does not have any non-blank symbols behind a blank symbol.

The symbol directly to the right of the $\sharp$ symbol corresponds to the symbol under the head of the i-th tape of M.

A state is valid iff all the strings in the tuple are valid. Note that a state is valid iff it corresponds to a configuration of M. Thus, the states that are added to the original sequence are exactly the states that are not valid.

Assume that the symbols of the strings have a numerical value and that the strings are ordered reversed lexicographically using this numerical value, i.e., the first symbol of a string is the least significant and the last symbol is the most significant. The description of M_n lists all the states of M_n. The order of the states depends on q, $v_1, \ldots, v_k$ and l. These components are ordered reversed lexicographically (l is the most significant bit and q is the least significant bit). The numerical values are chosen such that the initial state of M_n is the first state in the order.

Each state is accompanied by a list of transitions. Thus, each state corresponds to a labeled list of transitions. A marker is placed between every labeled list of transitions and another marker is placed between every transition within the labeled lists. Using these markers, a finite automaton can move a tape head between transitions or between states by simply counting the markers.

In addition to these labeled lists, the description starts with strings (yardsticks) of length $(c+2)^{j+(i-1)d \log n}$ for every $1 \leq i \leq k$ and every $0 \leq j \leq d \log n$, ordered lexicographically (i is the most significant bit), separated by a third type of markers. A special marker is used to separate the yardsticks from the labeled lists.

Since m' is constant, the length of the string v_i is logarithmically bounded and l is polynomially bounded, each state of M_n can be described with string of polynomial length and M_n contains polynomially many states. Since i is finite and j is logarithmically bounded, there are polynomially many yardsticks of polynomial length in the description. Thus, the description of M_n is polynomial in n.

A finite automaton M_{init} can find the initial state of M_n by moving a tape head over the description of M_n until the special marker is reached, using $O(1)$ states to recognize the marker.

Consider a string v_i, with $1 \leq i \leq k$. It is of the form

$$u_1 \ldots u_{j-2}\ u_{j-1} \quad \sharp \quad u_{j+1}\ u_{j+2} \ldots u_{d \log n}\ . \tag{4.1}$$

After a transition in which the i-th head of M does not move, the next string v_i' has the form of the second string in (4.2), where a is a symbol from the alphabet.

$$\begin{array}{llll} u_1 \dots u_{j-2} & u_{j-1} & \sharp & \boxed{u_{j+1}}\, u_{j+2} \dots u_{d \log n} \ , \\ u_1 \dots u_{j-2} & u_{j-1} & \sharp & \boxed{a}\, u_{j+2} \dots u_{d \log n} \ . \end{array} \tag{4.2}$$

It follows from (4.2) that the two strings differ only in one place. In the reversed lexicographical order, the string v_i' is located $(a - u_{j+1})(c+2)^j$ strings to the right from v_i.

If the head moves right during the transition, the string v_i' has the form of the last string in (4.3). Notice that the intermediate strings differ only in one place.

$$\begin{array}{llll} u_1 \dots u_{j-2} & u_{j-1} & \sharp & \boxed{u_{j+1}}\, u_{j+2} \dots u_{d \log n} \ , \\ u_1 \dots u_{j-2} & u_{j-1} & \boxed{\sharp} & \boxed{\sharp}\, u_{j+2} \dots u_{d \log n} \ , \\ u_1 \dots u_{j-2} & u_{j-1} & \boxed{a} & \sharp \quad u_{j+2} \dots u_{d \log n} \ . \end{array} \tag{4.3}$$

The string v_i' is located $(\sharp - u_{j+1})(c+2)^j$ plus $(a - \sharp)(c+2)^{j-1}$ strings to the right of v_i.

If the head moves left during the transition, the next string v_i' has the form of the last string in (4.4).

$$\begin{array}{llll} u_1 \dots u_{j-2} & u_{j-1} & \sharp & \boxed{u_{j+1}}\, u_{j+2} \dots u_{d \log n} \ , \\ u_1 \dots u_{j-2} & \boxed{u_{j-1}} & \sharp & \boxed{a}\, u_{j+2} \dots u_{d \log n} \ , \\ u_1 \dots u_{j-2} & \boxed{\sharp} & \sharp & a \quad u_{j+2} \dots u_{d \log n} \ , \\ u_1 \dots u_{j-2} & \sharp & \boxed{u_{j-1}} & a \quad u_{j+2} \dots u_{d \log n} \ . \end{array} \tag{4.4}$$

The string v_i' is located $(a - u_{j+1})(c+2)^j$ plus $(\sharp - u_{j-1})(c+2)^{j-2}$ plus $(u_{j-1} - \sharp)(c+2)^{j-1}$ strings to the right of v_i.

Suppose the first head of M_{step} is located on the description of a state $(q, v_1, \dots, v_k, l, a_l)$. A transition is simulated by moving the head to the description of the next state. The description of the transition includes the symbols under the k heads, the k head movements, the next state q' of M and the next values of l' and $a_{l'}$. The location of the next state is found by first moving to the state $(q', v_1, \dots, v_k, l, a_l)$. Note that this state lies at most m states to the left or right of the original state. Thus, it can be found by a finite automaton.

Then, the contents of v_i are replaced by v_i', one by one, until the first head of M_{step} is on the description of the state $(q', v_1', \dots, v_k', l, a_l)$. In the case of (4.2), this requires moving $(a - u_{j+1})(c+2)^j$ times $m' \cdot (c+2)^{(i-1)d \log n}$ states to the right to replace v_i. Since $(a - u_{j+1})$ and m' are bounded and the description contains a yardstick of length $(c+2)^{j+(i-1)d \log n}$, this move can be done by a finite automaton. However, this yardstick has to be located by M_{step} first. Note that this yardstick is the $(j + (i-1)d \log n)$-th yardstick in the list. Thus, M_{step} can move the second head to this yardstick, using i (in its finite control), $d \log n$ (the length of the string v_i) and j (indicated by the $\sharp$ symbol). Then, M_{step} can use the second head and the yardstick to move the first head to the next state. The situations for (4.3) and (4.4) are handled similarly. (Note that the order of the yardsticks implies that the yardsticks of length $(c+2)^{j-2+(i-1)d \log n}$, $(c+2)^{j-1+(i-1)d \log n}$ and $(c+2)^{j+(i-1)d \log n}$ are right next to each other.)

Finally, l and a_l need to be updated. The value of $a_{l'}$ is determined by l', which is between $l-1$ and $l+1$. Thus, the state $(q', v_1', \dots, v_k', l', a_{l'})$ is located another $m' \cdot (c+2)^{k \cdot d \log n}$ states to the left or right. This move can be done in a similar fashion as well.

There is a small problem with this method. Both tape heads are used to find the next state. Thus, we loose the position of the input head of M. Fortunately, l encodes the position of the input head of the Turing machine when $l \leq n$, so the second head can be moved back to the correct input position, using the encoding of l. If $l > n$, then the input is not used by the Turing machine, instead, an advice symbol is read. In this case, the second head should just move to the last input symbol. It follows that M_{step} can move from state to state on a tape containing this description, updating the heads after each move. Thus, the description satisfies Property 4.30.

□

Proposition 4.38. *Let L be a language decided by a Turing machine running in logarithmic space with an advice of polynomial size. Then L can be decided by a two-way finite automaton with one input head and an advice of polynomial size.*

Proof. The result follows from Propositions 4.37 and 4.32.

□

It is already known that polynomially sized finite automata with multiple heads have the same computational power as Turing machines running in logarithmic space (see Wagner and Wechsung[44]). The number of heads of the automaton depends on the constant factor of the logarithm. We see however, that when both devices may use an advice mechanism, one input head (plus one head for the advice) for the finite automaton is sufficient, independent of the factor of the logarithm.

Multi-Head Finite Automata with Advice

Next, we show that "*(iv)* $\Rightarrow$ *(iii)*" of Theorem 4.25 holds. It is a direct consequence of the stronger statement in Proposition 4.39.

Proposition 4.39. *Let L be a language over an alphabet of size c decided by a finite automaton of size m' with k input heads and an advice of length f. Then L can be decided by a sequence of finite automata of size $m'f(n)$ with k heads and descriptions of size $O(m'f(n)c^k \log(c^k m'))$ satisfying Property 4.30.*

Proof. The proof consists of two parts. The first part shows that an arbitrary finite automaton with advice can be simulated by a sequence of finite automata. The second part shows that the particular sequence that was constructed in the first part satisfies Property 4.30, by giving a description that satisfies it.

Let M be a finite automaton of size m' with k input heads and an advice of size f that decides L. A configuration of M consists of a state, the contents of the advice tape and the position of the advice head. For the n-th automaton M_n in the sequence, its states are those configurations where the advice tape contains the n-th advice, and the (relative) position of the advice head is at most $f(n)$. Since the advice is fixed for inputs of length n, the n-th automaton has $m'f(n)$ states, tuples of the form (q, w_j, j), where q is a state from M and w_j is the j-th symbol of the n-th advice string, for $1 \leq j \leq f(n)$.

Now, M_n starts in the configuration that contains the initial state of M and the advice for inputs of length n and has the k tape heads in the first position. Then, M_n reads the symbols under the k heads. Using the transition function, the k heads of M_n simulate one move of the input heads of M. Then, M_n changes to the configuration corresponding to the next state and the next position of the advice head of M. This way, the sequence of finite automata decides L.

Next, we give a description for the automata in the sequence that satisfies Property 4.30. We place the initial configuration first in the list, so that it can be found by a finite automaton M_{init}.

Suppose that the automaton M_n of the sequence is in a state q, with the advice head at position j. Since the heads of M_n can move at most one position at a time, after one transition M_n is in a state q', with the advice positioned at j', such that $j-1 \leq j' \leq j+1$. Thus, if we order the states reversed lexicographically, then the transition matrix has a bandwidth of $2m'$. It follows from Example 4.33 that M_n has a description of size $O(m'f(n)c^k \log(c^k m'))$ that satisfies Property 4.30.

□

Corollary 4.40. *Let k be a positive integer. Let L be a language decided by a finite automaton with one input head and an advice of polynomial size. Then L can be decided by a sequence of polynomially sized finite automata with k heads.*

Proof. This follows directly from Proposition 4.39 and the fact that m' is a constant.

□

Just as in Section 4.3, we can compare advice lengths to description sizes of finite automata. We combine the results in Theorem 4.41.

Theorem 4.41. *A language L can be decided by a sequence of finite automata with k heads and descriptions of size $O(f)$ satisfying Property 4.30 iff L can be decided by a finite automaton with k input heads and an advice of size $O(f)$.*

Proof. The result follows from Proposition 4.32, Proposition 4.39 and the fact that k, c and m' are constants.

□

4.5 Sequences of Multi-Head Pushdown Automata

As a next case, we consider multi-head automata with the simplest kind of external memory commonly considered in automata models, i.e., the pushdown store. We will show the following characterization of the computational power of sequences of multi-head pushdown automata, as another example of the power of the framework.

Theorem 4.42. *Let k be a positive integer. The following statements about a language L are equivalent:*

(i) L is in P/poly,
(ii) L can be decided by a sequence of polynomially sized Boolean circuits,

(iii) *L can be decided by a sequence of polynomially sized polynomial time k-tape Turing machines,*

(iv) *L can be decided by a sequence of polynomially sized two-way pushdown automata with k' heads, for a certain integer k',*

(v) *L can be decided by a pushdown automaton with k' input heads and an advice of polynomial size, for a certain integer k'.*

The equivalence between *(i)* and *(ii)* has already been shown by Karp and Lipton[22] (see Corollary 4.14). Corollary 4.19 shows the equivalence between *(i)* and *(iii)*. We will complete the proof by showing the equivalence between *(i)* and *(iv)* and the equivalence between *(iv)* and *(v)*. See also Petersen[37] for a direct approach to the equivalence between *(i)* and and *(v)*.

4.5.1 Turing Machines with Advice

The next Proposition shows that "*(i)* $\Rightarrow$ *(iv)*" of Theorem 4.42 holds. We will build on the ideas from Cook[9].

Proposition 4.43. *If a language L is decided by a Turing machine running in polynomial time with an advice of polynomial size, then there is an integer k such that L can be decided by a sequence of polynomially sized two-way pushdown automata with k heads.*

Proof. Consider a Turing machine that decides L in polynomial time T, using an advice of polynomial size. Then, there is a Turing machine with one writable tape that decides L, in time $O((T(n))^2)$, which is polynomial too. This Turing machine moves its head in a sweeping pattern. By the results proved by Cook[9], this Turing machine can be simulated by a pushdown automaton with an additional work tape of length $O(\log T(n))$. Since T is polynomial, the work tape is logarithmically bounded. The finite control incorporates the transition function of M, (using $O(1)$ size), and the function of t which produces the position the head is in at time t (using $O(1)$ size and a logarithmically bounded work-tape).

Now, this pushdown automaton is not the one we want, since it uses a work-tape and takes an advice with the input. We solve the latter by storing the advice directly in the finite control of the pushdown automaton (adding polynomially many states), and the former by noting that an automaton with logarithmically bounded work-tapes is equivalent to an automaton with k tape heads, for a certain integer k, see Wagner and Wechsung[44]. Note that k depends on the running time of the Turing machine, so we can use the same k for all pushdown automata in the sequence. Putting everything together, the sequence uses polynomially sized pushdown automata with k heads to decide L.

□

The following Proposition shows that "*(iv)* $\Rightarrow$ *(i)*" of Theorem 4.42 holds.

Proposition 4.44. *Let k be a positive integer. If a language L is decided by a sequence of polynomially sized two-way pushdown automata with k heads, then L can be decided by a Turing machine running in polynomial time with an advice of polynomial size.*

Proof. The Turing machine takes the description of the pushdown automata as advice. Polynomially sized pushdown automata have descriptions of polynomial size.

The simulation is achieved by examining realizable pairs of configurations of the pushdown automaton. A configuration consists of a state, the contents of the k heads and the symbol on top of the pushdown store. For the definition of realizable pairs and the operation to yield a new realizable pair given two realizable pairs, see Cook[9]. A realizable pair (C, C') is accepting if C is the initial configuration and C' is an accepting configuration. The Turing machine starts by storing all realizable pairs (C, C). Then, the machine repeatedly applies the yield operation to all pairs in this list, adding the newly found pairs to this list, until an accepting pair has been found (in which case the machine accepts) or no new realizable pairs can be found anymore (the input is rejected).

To perform the yield operation, the Turing machine has to look up the transition function from its advice and apply it to the configurations in the pairs. Let $f(n)$ be the size of the description of the n-th pushdown automaton M_n. Then an application of the yield operation takes $O(f)$ time.

Let c the size of the alphabet of the pushdown automata. If M_n has $m(n)$ states, then there are $m(n)n^k c$ configurations. To avoid confusion, we refer to realizable pairs as items in the list. Let r_i be the number of items in the list after the yield operation has been applied to all pairs of items in the list i times. Then $r_0 = m(n)n^k c$. The yield operation has to be applied to all pairs of items in the list, so the i-th round takes $O((r_i)^2 f)$ time. In the worst case, r_i increases by one each round and every pair of configurations is realizable. Then, the time to compute all realizable pairs is

$$\sum_{j=r_0}^{(r_0)^2} O\left(j^2 f\right) = O\left(\left(r_0\right)^4 f\right) = O\left(\left(m(n)n^k c\right)^4 f(n)\right) \quad . \tag{4.5}$$

Thus, all realizable pairs can be found in polynomial time.

□

4.5.2 Multi-Head Pushdown Automata with Advice

The main difference between a finite automaton and a pushdown automaton is that a pushdown automaton can store infinite amounts of information in a pushdown store. The pushdown store is stored on a separate tape with one tape head. We define a pushdown automaton with advice capabilities as a pushdown automaton with two heads for its input tape (or two tapes, one containing the input, the other containing the advice).

Ibarra[20] proved that we cannot simulate a pushdown automaton with multiple heads by a pushdown automaton with one head. It follows that the advice mechanism requires at least two heads. On a multi-head pushdown automaton with advice, one of the heads of the input tape is designated as the *advice head*, the other heads are the *input heads*.

We list a property of descriptions of sequences of pushdown automata that enables us to simulate the sequence with a pushdown automaton that takes these descriptions as advice.

Property 4.45. Let k be an integer. Consider a sequence of pushdown automata with k heads. There is an encoding mechanism for the automata in the sequence and pushdown automata M_{init} and M_{step} with $k+1$ heads such that for an automaton M in the sequence, the description of M using this mechanism has the following properties:

- The description is based on a transition-list,
- the initial state of M can be found by M_{init} if its input tape contains the description of M,
- for each state q of M, every input x of M, every possible placement of the k heads on x and every string y in the pushdown store of M, which cause the transition function to output state q' and replace y by y' in the pushdown store, if the pushdown automaton M_{step} contains x, as well as the description of M on its input tape, a tape head positioned at the beginning of q on this description, the other k heads placed at the same positions of the input and the pushdown store contains y, then M_{step} moves a tape head to the beginning of q' and the remaining k heads to the positions corresponding to the transition and replaces y by y' in the pushdown store.

Note that Property 4.45 is very similar to Property 4.30, only the occurrences of the word "finite" are replaced by the word "pushdown" and the contents of the pushdown store are taken into account. Another difference between the two Properties is that it is unlikely that every finite automaton has a description satisfying Property 4.30, while there is an encoding that satisfies Property 4.45 for every sequence of pushdown automata.

Proposition 4.46. *Consider a sequence of two-way pushdown automata with k heads, for an integer k. If the automata are described by transition-lists, then the descriptions satisfy Property 4.45.*

Proof. Since the initial state is marked, the pushdown automaton M_{init} can move an input head on the input tape holding the description of M until it locates this (finite) marker, using $O(1)$ states.

The pushdown automaton M_{step} contains the input to M as well as the description of M on its input tape. One head is positioned at the beginning of the description of a state q, the remaining k heads are positioned on the input cells corresponding to the positions of the k heads of M. The pushdown store of M_{step} contains the pushdown store contents of M.

The automata all operate over a fixed alphabet, say of size c. Then, the number of transitions starting in one state is at most c^{k+1}. Thus, we need $O(c^k)$ states to distinguish between the different descriptions of transitions following q.

Suppose the transition corresponding to the c^{k+1} tape symbols is found. If the transition pops a symbol from the pushdown store of M, then M_{step} pops its pushdown store too. Similarly, if a symbol is pushed onto the pushdown store of M, the same symbol is pushed onto the pushdown store of M_{step}. Then, k of the heads of M_{step} are moved according to the transition of M.

Next, a special marker is pushed onto the pushdown store, followed by the description of q' (using $O(1)$ states). The remaining head is moved to the end of the description and from there M_{step} searches for a match to the state in the

pushdown store. This is done by using the markers between the states in the list to find the end of a state. From here, the head moves backwards until the previous marker is reached, comparing the symbol under the pointer head with the symbol that is popped from the pushdown store. If the symbols don't match, or if either one reaches the end before the other, then the head is moved forwards again, pushing back the symbols that were popped off the pushdown store. Then, the head is moved backwards to the end of the state before the one it is on now and tries again. Eventually, q' will be found. Now, the marker is popped from the pushdown store again, so that M_{step} contains the correct contents of the pushdown store again. The number of states used in this part of the procedure depends only on the size of the markers. Thus, $O(1)$ states are sufficient.

All in all, M_{step} uses finitely many states to move the heads to the correct positions and modify the pushdown store.

□

The implication "*(iv)* ⇒ *(v)*" of Theorem 4.42 is a direct consequence of the next Proposition. It is stated in Corollary 4.48.

Proposition 4.47. *Let L be a language that is decided by a sequence of pushdown automata with k heads running in time T with descriptions of size f, using an encoding that satisfies Property 4.45 for a certain integer m. Then, L can be decided by a pushdown automaton with k input heads and an advice of size f.*

Proof. Let $M_1, M_2, \ldots$ be the pushdown automata in a sequence that decides L. We construct a pushdown automaton M with advice that decides L. For inputs of length n, the advice contains the description of the n-th pushdown automaton M_n. The advice head is used as the $(k+1)$-st tape head, so M may call the pushdown automata described in Property 4.45.

The simulating pushdown automaton M starts by moving the k heads to the first symbol of the input. Now, M calls M_{init} to find a pointer (the advice head) to the initial state. This takes $O(1)$ states. After this, M can repeatedly call M_{step} to find the next state of the computation, updating the head positions and the pushdown store along the way, using $O(1)$ states, until a final state is found. It follows that M simulates M_n using a finite number of states.

□

Corollary 4.48. *Let L be a language that is decided by a sequence of polynomially sized pushdown automata with k heads. Then L can be decided by a pushdown automaton with k input heads and an advice of polynomial size.*

Proof. If a pushdown automaton has polynomial size, then its transition-list has polynomial size. The result then follows from Propositions 4.46 and 4.47.

□

Corollary 4.50 proves that "*(v)* ⇒ *(iv)*" of Theorem 4.42 holds. It follows from the next Proposition.

Proposition 4.49. *Let L be a language that is decided by a pushdown automaton of size m' with k heads and an advice of length f. Then L can be decided by a sequence of pushdown automata of size $O(m'f(n))$ with k heads.*

Proof. Let M be a pushdown automaton of size m' with advice of size f that decides L. We are going to simulate M by moving from configuration to configuration again. Since the pushdown store can be unbounded, the number of configurations is unbounded too, so we exclude the contents of the pushdown store from the configurations. Instead, the contents of the pushdown store are stored in the pushdown store of the machines in the sequence. Thus, a configuration consists of a state, the contents of the advice tape and the position of the advice head. For the n-th pushdown automaton in the sequence, the advice tape contains the advice for strings of length n and the (relative) position of the advice head is at most $f(n)$. So, the n-th pushdown automaton has $O(m'f(n))$ states.

The simulation proceeds similar to previous results (e.g. Proposition 4.39), except the symbol at the top of the pushdown store is also taken into account to determine the next configuration. Of course, the pushdown store is synchronized with the pushdown store of M at every step.

□

Corollary 4.50. *Let k be a positive integer. Let L be a language decided by a pushdown automaton with k input heads and an advice of polynomial size. Then L can be decided by a sequence of polynomially sized pushdown automata with k heads.*

Proof. This follows from Proposition 4.49 and the fact that m' is a constant.

□

We finish this section by comparing advice lengths to description sizes of pushdown automata.

Theorem 4.51. *A language L can be decided by a sequence of pushdown automata with k heads and descriptions of size $O(f)$ based on transition-lists iff L can be decided by a pushdown automaton with k input heads and an advice of size $O(f)$.*

Proof. The first part of the equivalence follows from Propositions 4.46 and 4.47. The sequence of pushdown automata constructed in Proposition 4.49 has a bounded bandwidth, so the machines in the sequence can be described as in Example 4.33. Since c, k and m' are constants, the result follows.

□

4.6 Using Transition Lists or Flow Lists for Advice

A sequence of machines of type $\mathcal{M}_1$ can be simulated by a machine of type $\mathcal{M}_2$ with advice if there is an encoding mechanism that satisfies Property 4.1. Unfortunately, this is a rather abstract characterization. It was shown that sequences of finite automata with an encoding mechanism satisfying Property 4.30 also satisfy Property 4.1. Similarly, pushdown automata were shown to have an encoding mechanism satisfying Property 4.45, which in turn implied that Property 4.1 was satisfied. For sequences of Turing machines, we used the two-part encoding from the Proof of Proposition 4.21. This encoding consisted of listing w, describing a machine to generate tuples and describing the simulating machine $M_{n,w}$. If the machines

are described by transition-lists, then the total description is also a transition-list. This encoding mechanism then satisfies a property similar to Property 4.30 and 4.45. Furthermore, this encoding mechanism satisfies Property 4.1. This indicates that there is a more practical Property that we can use to the same effect.

Property 4.52. Let $\mathcal{M}_1$ be a transition-based machine model. Given a sequence of machines of type $\mathcal{M}_1$, there is an encoding mechanism for the machines in the sequence and machines M_{init} and M_{step} of type $\mathcal{M}_2$ such that for an automaton M in the sequence, the description of M using this mechanism has the following properties:

- The description is a transition-list,
- a pointer to the initial state of M can be found by M_{init} if its data contains the description of M,
- for every state q and every configuration C of M, which cause the transition function to output state q' and configuration C', if the data of the machine M_{step} contains the description of M and a pointer to q on this description, as well as a description of C, then M_{step} replaces the configuration C by C' in its data and produces a pointer to q'.

Proposition 4.53. *Let $\mathcal{M}_1$ and $\mathcal{M}_2$ be two machine models. Suppose $\mathcal{M}_2$ satisfies Property 4.6. Then Property 4.52 implies Property 4.1.*

Proof. Let M_1, M_2,... be a sequence of machines of type $\mathcal{M}_1$, satisfying Property 4.52. Let x be a string of length n and let w_n be the description of M_n. We construct a machine M of type $\mathcal{M}_2$ that simulates M_n on x. Machine M uses advice w_n, so w_n is stored as data. By Property 4.6, M can call the machine M_{init} to find a pointer to the initial state. Observe that the data of M contains the initial configuration of M_n.

Now, M operates in phases. Each phase simulates one transition of M_n. A phase starts when M has a pointer to the current state, and its data contains the configuration of M_n up to this point. The machine M_{step} is called by M. Since the data of M contains a description of M_n, a pointer to the current state and the configuration of M_n, the configuration gets updated correctly and a pointer to the next state is found. When this is done, M is ready to start the next phase.

It follows that M simulates M_n with advice w_n. Let T' be the running time of M_n, let S' be the space usage of M_n and let m' be the size of M_n. Then, the running time T and the space usage S of M depend on the length of the stored data of M (n, $|w_n|$ and S'), the actual implementations of M_{init} and M_{step} (a constant factor) and the number of configurations of M_n, which in turn depends on T', S' and m'. Thus, for a machine described by w with an input x, the functions T and S depend on $|x|$, $|w|$, S', T' and m'.

□

We give a similar property for flow-based machine models.

Property 4.54. Let $\mathcal{M}_1$ be a flow-based machine model. Given a sequence of machines of type $\mathcal{M}_1$, there is a encoding mechanism for the machines in the sequence and machines M_{step} and M_{flow} of type $\mathcal{M}_2$ such that for an automaton M in the sequence, the description of M using this mechanism has the following properties:

- The description is a flow-list,
- a pointer to the next gate of M can be found by M_{step} if it has the description of M stored as data, and a pointer to the current gate,
- for each gate q of M with edges coming in from gates $q_1, \ldots, q_k$, the output of q can be produced by M_{flow}, if it has the description of M, as well as the outputs of gates $q_1, \ldots, q_k$ stored as data, and a pointer to q.

We assume that M_{step} finds the first gate when it has a pointer to the beginning of the description.

Proposition 4.55. *Let $\mathcal{M}_1$ and $\mathcal{M}_2$ be two machine models. Suppose $\mathcal{M}_2$ satisfies Property 4.4 and 4.6. Then Property 4.54 implies Property 4.1.*

Proof. Let $M_1, M_2, \ldots$ be a sequence of machines of type $\mathcal{M}_1$, satisfying Property 4.54. Let x be string of length n and let w_n be the description of M_n. We construct a machine M of type $\mathcal{M}_2$ that simulates M_n on x. Machine M uses advice w_n, so w_n is stored as data. By Property 4.6, M can call the machine M_{step} to find a pointer to the first gate. From here, M alternately calls the machines M_{flow} and M_{step} to find and store all the outputs of the gates, until a stopping criterion is met. It follows that M simulates M_n with advice w_n. Observe that M needs to have enough space to store the outputs of all gates of M_n, which is possible by Property 4.4.

Let T' be the running time of M_n, let S' be the length of the outputs of the gates of M_n and let m' be the size of M_n. Then, the running time T and the space usage S of M depend on length of the stored data of M (n and w_n), the actual implementations of M_{step} and M_{flow} (constant factor), the outputs of the gates (S') and the number of flow steps (T'). Thus, the functions T and S depend on $|x|$, $|w|$, S', T' and m'.

□

It is not necessarily true that Property 4.1 implies Property 4.52 or 4.54. This is because the machine used in Property 4.1 may use any advice. Thus, advices which behave like characteristic functions can be used, but they convey no information about the internal workings of the machine.

4.7 Conclusions

The equivalence result expressed in Theorems 4.12 and 4.13 is stated in such a way that it can be applied to many machine models once the necessary translations have been made. Indeed, the structure of this Chapter is such that this has been done. With these common results in place, the bulk of the work is constructing machines of one class that simulate machines of another. The real strength of the results however, lies in the comparisons that can be made with it. By casting the results in the framework of the abstract properties, it becomes apparent that most results are in fact based on a few very similar core ideas. It enables us to relate the power of sequences of machines to classical advice classes, in several important cases of evolving system models.

Historically, resource bounds of sequences of machines were measured by the sizes of the machines. As we observed, if we relate this measure to advice lengths, then logarithmic blowups occur. This is because the size of a machine is an incomplete measure, its program is also needed. On the other hand, if we use the description size as a resource bound, then no blowups occur (see e.g. Corollaries 4.23 and 4.24 and Theorems 4.41 and 4.51). The description size is a more natural measure for the complexity of a machine (see also Li and Vitányi[31]). Thus, we propose to use description sizes when we discuss sequences of machines. Fortunately, when dealing with logarithmic or polynomial sizes, the blowups have no impact on the resulting sizes.

CHAPTER 5

Interactive Turing Machines

The interactive Turing machine, which was introduced by Van Leeuwen and Wiedermann[28, 30], is an extension of the classical Turing machine model. It attempts to model modern real-life computing systems more accurately. One aspect of such systems is that they, unlike classical models, interact with their environment. With this, we mean that the system allows new, unforeseen inputs to appear as the computation proceeds, instead of all inputs being fixed before the computation starts. The new inputs may depend on outputs that were produced earlier. This fundamental difference between classical models and the systems they try to model implies that several important ideas about the theory simply do not match reality. For instance, real-life computations can be extended without limits.

The model of interactive Turing machine models tries to capture this aspect of interactivity of real-life computing systems. Using this model, we set up a new theory, to better approximate the reality of computational systems.

Another aspect of real-life computing systems is that they possess the ability to change their behavior during a computation, or to evolve over time. This aspect is modeled by equipping interactive Turing machines with an advice mechanism. This evolving behavior gives interactive Turing machines the ability to "break the Turing barrier".

The Chapter is set up as follows: First, interactive Turing machines and the translations that they compute are introduced. Next, some properties of these translations are explored. Then, the efficiency of the model is examined by defining complexity measures and proving that these measures induce non-trivial complexity hierarchies. Finally, interactive Turing machines are extended with an advice mechanism. A complexity measure based on the size of the advices is defined and the influence on the translations that can be realized is examined. We prove that there are translations that can be processed by an interactive Turing machine with an advice of length g, but not by any interactive Turing machine with an advice of length f, when $f \in o(g)$. If we only allow advices over alphabets of sizes bounded

by an integer larger than 2, then the result holds even if $f(n) < g(n)$ for infinitely many n.

5.1 Interactive Turing Machines

Let M be an interactive Turing machine (ITM). An ITM M has an input port, an output port and k work tapes. At each time-frame, a symbol appears at the input port and the ITM produces a symbol at the output port, determined by its transition function. The operation of an ITM is comparable to that of a Turing machine with a one-way input and output tape. The major difference is the lack of final states and the fact that the length of the input is not known in advance. Essentially, ITM's can process infinite inputs, which makes an ITM a kind of ω-Turing machine. See Staiger[39] for an overview of ω-languages. Since an input can be extended arbitrarily at any point, the machine has no way of knowing when the computation should halt. Thus, final states are of not much use. Next, a formal definition of ITM's is given.

Definition 5.1. *An interactive Turing machine (ITM) with k work tapes is a 6-tuple $M = (\Sigma, \Omega, Q, I, q_{\text{in}}, \delta)$, where Σ and Ω are non-empty finite alphabets, Q is a set of states, I is a subset of Q, q_{in} is a state of $Q - I$ and $\delta : Q \times \Sigma^k \times (\Sigma \cup \{\lambda\}) \to Q \times \Sigma^k \times \{\text{left}, \text{none}, \text{right}\}^k \times (\Omega \cup \{\lambda\})$ is a (partial) transition function. Σ is the input alphabet and Ω is the output alphabet. λ is a special symbol. We call q_{in} the initial state, I the set of* internal states *and $Q - I$ the set of* external states.

It is convenient to use a transition function based on configurations instead of states, as it simplifies many definitions. A configuration consists of a state, the contents of the tapes and the positions of the tape heads. Thus, a configuration is a tuple $(q, v_1, \ldots, v_k, i_1, \ldots, i_k)$, for a state $q \in Q$, strings $v_j \in \Sigma^\star$ and integers $1 \leq i_j \leq |v_j + 1|$ for every $1 \leq j \leq k$. Let $\mathbf{Q}$ be the set of configurations. The initial configuration $\mathbf{q}_{\text{init}}$ is the configuration consisting of the initial state and k empty tapes, with the tape heads at the first position. Defining the transition function $\boldsymbol{\delta} : \mathbf{Q} \times (\Sigma \cup \{\lambda\}) \to \mathbf{Q} \times (\Omega \cup \{\lambda\})$ is left as an exercise to the reader. In a sense, we view the machines as infinite state (or configuration) automata.

5.1.1 Internal and External Phases

In addition to the symbols from Σ, a special symbol λ may be produced at the input port. This symbol conveys no meaningful information. Its only use is to have a symbol produced at the input port when there is no actual input. Similarly, the ITM can produce λ at the output port when it has no useful outputs to produce.

Definition 5.2. *Let x be a string in $(\Sigma \cup \{\lambda\})^\infty$. We can* filter *$x$ to a string x' in Σ^∞ by replacing all substrings of the form $\lambda^\star$ in x by empty strings. The same can be done with strings in $(\Omega \cup \{\lambda\})^\infty$.*

Sometimes, an ITM needs some time for non-interactive calculations. During this time, it cannot accept new symbols at the input port. To accommodate these

calculations, the computation is split into internal phases and external phases. During an internal phase, the only symbols that are accepted at the input port are λ's. The ITM produces λ's at the output port until the internal phase ends. Since ITM's model interactive systems, we require that the internal phases end after some time, i.e., we want every internal phase to be *finite*.

Formally, we designate a subset of the control states of the ITM as *internal states*. The complement is referred to as the set of *external states*. The initial state q_{init} is an external state.

Definition 5.3. *A configuration with an internal state is an* internal configuration *and a configuration with an external state an* external configuration. *An* internal phase *is a maximal part of a computation that consists of only consecutive internal configurations.*

Thus, an ITM may initiate an internal phase by changing to an internal configuration. An internal phase can be seen as a brief time-out for the ITM. The environment can also initiate time-outs by passing λ's to the input port, which are ignored by the ITM.

The transition function of an ITM satisfies some constraints. Consider a transition $\boldsymbol{\delta}(\mathbf{q}, a) = (\mathbf{r}, b)$, for configurations $\mathbf{q}$ and $\mathbf{r}$ and input symbol a and output symbol b. Then it is required that:

- If $\mathbf{q}$ is external and $a = \lambda$, then $\mathbf{r} = \mathbf{q}$ and $b = \lambda$.
- If $\mathbf{q}$ is internal, then $a = \lambda$.
- If $\mathbf{r}$ is internal, then $b = \lambda$.
- If $\mathbf{r}$ is external and $a \neq \lambda$, then $b \neq \lambda$.

Note that the first requirement follows from the assumption that the λ symbols carry no information.

Remark 5.4. It follows from the constraints that every internal phase starts with a non-λ input symbol and ends with a non-λ output symbol, all other symbols received or produced in an internal phase are λ's. Thus, for every sequence of symbols appearing at the input port, the number of non-λ input symbols equals the number of non-λ symbols produced at the output port. It follows furthermore that between any two non-λ input symbols, exactly one non-λ symbol is produced at the output port.

5.1.2 Inputs and Outputs

Let $x \in (\Sigma \cup \{\lambda\})^\infty$ be a sequence of symbols appearing at the input port of an ITM M. We define a sequence of configurations $(\mathbf{q}_j)_{j \geq 1}$ in $\mathbf{Q}$ such that:

$$\begin{aligned} \mathbf{q}_1 &= \mathbf{q}_{\text{in}} \ , \\ \mathbf{q}_{j+1} &= \pi_1 \left(\boldsymbol{\delta} \left(\mathbf{q}_j, x_j \right) \right) \ , \end{aligned} \tag{5.1}$$

where $\boldsymbol{\delta}(\mathbf{q}_j, x_j)$ is undefined if $\mathbf{q}_j$ is and $\mathbf{q}_{j+1}$ is undefined iff $\boldsymbol{\delta}(\mathbf{q}_j, x_j)$ is.

Definition 5.5. *We say that M can process x if the following holds:*

- $\mathbf{q}_j$ *is defined for every* $j \leq 1 + |x|$,

- *if* $\mathbf{q}_j$ *is an internal configuration, then there is an* $i > j$ *such that* $\mathbf{q}_i$ *is an external configuration.*

In this case, the unfiltered output *of* M *on* x *is the string* $y \in (\Omega \cup \{\lambda\})^\infty$ *such that* $y_j = \pi_2(\boldsymbol{\delta}(\mathbf{q}_j, x_j))$*, for every* $1 \leq j \leq |x|$*. The* output *of* x *is the string in* Ω^∞ *that we get after filtering* y*.*

Remark 5.6. It follows from the definition that if M processes a string x, then every internal phase is finite. The first configuration in the sequence is external. Furthermore, if x is finite, then the last configuration must be external as well.

Definition 5.7. *We say that a string* $x \in \Sigma^\infty$ *is a* valid input *to an ITM* M *if* M *can process a string* x' *that filters to* x*.*

Consider an ITM M and let x be a valid input to M. Since x is a valid input, there is a string x' that filters to x and can be processed by M. In fact, there are infinitely many such strings, as follows from the constraints on the transition function. The set of these strings contains a minimal string in a sense described next. This string can be seen as the most efficient way in which the environment can write its information to the input port, i.e., whenever M allows non-λ symbols to appear at the input port, the environment writes non-λ symbols to the input port. The string is called an *efficient input*. To define efficient inputs, we use the concept of *external* λ's.

Definition 5.8. *Let* $x \in (\Sigma \cup \{\lambda\})^\infty$ *be a string that can be processed by an ITM* M*. Let* $\mathbf{q}_1$*,* $\mathbf{q}_2$*, ... be the configurations of the computation on* x*. A* λ *symbol at an index* n *of* x *is an* external λ *of* x *if* $\mathbf{q}_n$ *is an external configuration.*

The following Lemma shows how to remove an external λ from a string that can be processed by an ITM.

Lemma 5.9. *Let* $x = x_1 x_2 \ldots \in (\Sigma \cup \{\lambda\})^\infty$ *be a string that can be processed by an ITM* M*. Let* $y = y_1 y_2 \ldots$ *be the unfiltered output of* M *on* x*. Let* x_n *be an external* λ *of* x*. Then, the string* x'*, defined by* $x' = x_1 \ldots x_{n-1} x_{n+1} x_{n+2} \ldots$*, can also be processed by* M*. The unfiltered output of* M *on* x' *is the string* $y' = y_1 \ldots y_{n-1} y_{n+1} y_{n+2} \ldots$*.*

Proof. Let $\mathbf{q}_1$, $\mathbf{q}_2$, ... be the configurations of the computation on x. Since x_n is an external λ, $\mathbf{q}_n$ is an external configuration. By the constraints on the transition function, $y_n = \lambda$ and $\mathbf{q}_{n+1} = \mathbf{q}_n$. Thus, the sequence of configurations $\mathbf{q}_1$, ..., $\mathbf{q}_{n-1}$, $\mathbf{q}_{n+1}$, $\mathbf{q}_{n+2}$, ... corresponds to a computation of M on x'. Clearly, the unfiltered output of M on x' is y'.

□

Note that x and x' filter to the same string, as do y and y'. Thus, M produces the same output on x and x'.

Definition 5.10. *Let* u *be a valid input to an ITM* M*. An* efficient input *for* u *is a string* $x \in (\Sigma \cup \{\lambda\})^\infty$ *such that*

- x *filters to* u*,*

- *x can be processed by M,*
- *x contains no external λ's.*

If x is a finite string that can be processed by an ITM, then we can find an efficient input that produces the same output as x by repeatedly applying Lemma 5.9. However, if x contains infinitely many external λ's, then we need a more subtle approach, as detailed in the proof of Proposition 5.11.

Proposition 5.11. *Let u be a valid input to an ITM M. Then, there is an efficient input for u.*

Proof. Let $x \in (\Sigma \cup \{\lambda\})^\infty$ be a string that filters to u and can be processed by M. We define a sequence of strings $x(1)$, $x(2)$, ... with the following properties for every $n \leq |u|$:

- $x(n)$ is a prefix of $x(n+1)$,
- $x(n)$ filters to $u_{[1:n]}$,
- $x(n)$ can be processed by M,
- $x(n)$ contains no external λ's.

It follows from the first item that the sequence converges to a string x' in $(\Sigma \cup \{\lambda\})^\infty$. The second item implies that x' filters to u. By the third item, x' can be processed by M. The last item ensures that x' has no external λ's. Thus, if the sequence $x(1)$, $x(2)$,... exists, then its limit x' is an efficient input for u.

For any n, the string $x(n)$ is defined by taking the smallest prefix of x that filters to $u_{[1:n]}$ and removing all external λ's. Since the prefix is finite, all external λ's can be removed from the prefix by repeatedly applying Lemma 5.9.

Consider the smallest prefix of x that filters to $u_{[1:n]}$. The last symbol of this prefix has to be u_n, which is a non-λ symbol. Thus, this prefix can be processed by M. It follows by Lemma 5.9 that $x(n)$ can also be processed by M. The remaining properties follow directly from the construction of the strings $x(n)$. Hence, x' is an efficient input for u.

□

Note that for every n, the string $x(n)$ is an efficient input for the prefix $u_{[1:n]}$. The following Proposition shows that an efficient input is unique.

Proposition 5.12. *Let x and x' be two efficient inputs for a valid input u. Then x equals x'.*

Proof. Write $u = u_1 u_2 \ldots$ and write x and x' as

$$\begin{aligned} x &= \lambda^{k_1} u_1 \lambda^{k_2} u_2 \lambda^{k_3} u_3 \ldots \ , \\ x' &= \lambda^{l_1} u_1 \lambda^{l_2} u_2 \lambda^{l_3} u_3 \ldots \ . \end{aligned} \tag{5.2}$$

Let $\mathbf{q}_1$, $\mathbf{q}_2$, ... and $\mathbf{q}'_1$, $\mathbf{q}'_2$, ... be the two computations corresponding to x and x'. Suppose that x and x' are different. Let i be the first index for which k_i and l_i differ. Assume that $k_i < l_i$ and let $j = i + \sum_{m=1}^{i} k_m$. Then, j is the first index for which x and x' differ. Now, $x_j = u_i$ and $x'_j = \lambda$. Since x_j is not λ, the configuration $\mathbf{q}_j$ must be external. Since ITM's are deterministic machines, the configuration $\mathbf{q}'_j$ is external as well. But since $x'_j = \lambda$, it follows that x' is not an efficient input,

contradicting the choice of x'. We conclude that i does not exist, so x must equal x'.

□

Incidentally, since $\mathbf{q}_{\text{init}}$ is an external configuration, it follows that k_1 must be 0, i.e., an efficient input for u necessarily begins with the non-λ symbol u_1.

Inputs of ITM's as strings of Σ^∞ are properly defined. However, we still miss one definition for the corresponding outputs in Ω^∞.

Definition 5.13. *Let x be a valid input for an ITM M. The* output *of M on x is the output of an efficient input for x.*

With the concepts of input and output firmly in place, we can define the functions that are realized by the ITM's.

Definition 5.14. *Let M be an ITM. We define the partial function $\Phi^M : \Sigma^\infty \to \Omega^\infty$ by letting $\Phi^M(x)$ be the output of M on x if x is a valid input and undefined otherwise, for every string x. We say that Φ^M is* interactively *realized by the ITM M. In general, we say that a partial function $\Psi : \Sigma^\infty \to \Omega^\infty$ is* interactively *realizable if there is an ITM M such that Ψ equals Φ^M.*

5.2 A New Halting Criterion for Turing Machines

It is clear that ITM's have a close relationship to classical Turing machines. However, ITM's have no concept of final states, so the halting criterion of Turing machines has no equivalent in ITM's. We introduce a new halting criterion for Turing machines that is closer to the spirit of extensible inputs.

Definition 5.15. *A* halt-on-blank *Turing machine (BTM) is a Turing machine that halts when the symbol under its input head is a blank symbol.*

Note here that the concept of a blank symbol is different from that of the λ-symbol. In particular, the blank symbol indicates the end of the input and does carry meaningful information. See Landweber[26] for other possible halting criteria for machine models.

Due to their sequential nature, ITM's have a close relation to BTM's. The following Proposition explains this relation in more detail.

Proposition 5.16. *Let Φ be an interactively realizable translation and define the partial function $\Phi' : \Sigma^\star \to \Omega^\star$ by letting $\Phi'(x) = \Phi(x)$ for every finite string x. Then, there is an ITM M that interactively realizes Φ iff there is a BTM M' that realizes Φ.*

Proof. Assume that M' produces the last output symbol when it reaches the blank after the input. Let x be a string. For any integer $1 \leq n < |x|$ such that the string $x_{[1:n]}$ is in the domain of Φ', the machine M' produces the output $\Phi'(x_{[1:n]})$ on input $x_{[1:n]}$, halting in a configuration $\mathbf{q}_n$.

The construction of M proceeds recursively. Suppose M receives x at its input port. Suppose that n output symbols have been produced, and that M is in an

external configuration. Part of this configuration is a special tape, containing the string $x_{[1:n]}$, with the head positioned on the blank at the end of the tape. Furthermore, the configuration contains a representation of $\mathbf{q}_n$, in a way that it can be used to simulate M'. Note that this is true for the base case $n = 0$.

Since M is in an external configuration, its configuration does not change until x_{n+1} is read from the input port. At this point, M initiates an internal phase and appends x_{n+1} to the special tape. Then, the machine M' is simulated on the special tape, starting from the configuration $\mathbf{q}_n$. If M' halts on the string $x_{[1:n+1]}$, then M' produces the $n+1$)-st output symbol when it encounters the blank on the input tape. In this case, the simulation of M' produces exactly one non-λ symbol, ending the internal phase. Now, the configuration contains $x_{[1:n+1]}$ on the special tape, with the head at the blank, and a representation of $\mathbf{q}_{n+1}$. It follows that M produces $\Phi'(x_{[1:n+1]})$ on input $x_{[1:n+1]}$.

If the string $x_{[1:n]}$ is not in the domain of Φ' for a certain n, then either M' doesn't halt, or an undefined transition is encountered. In both cases, the string x is not a valid input to M, nor does it belong to the domain of Φ. We conclude that M interactively realizes Φ.

On the other hand, the ITM M can be converted to a 1-way Turing machine M' by simulating M, inserting λ's in the input when M is in an internal configuration and suppressing the λ's in the output. On an input of length n, the simulation produces n non-λ symbols before moving on to the $(n+1)$-st input symbol (the blank). Thus, M' may halt as soon as it encounters a blank on the input tape. In other words, M' is a BTM that realizes Φ'.

□

5.2.1 A Complexity Hierarchy for BTM's

The time and space complexity of a BTM are defined similar to the time and space complexity of classical Turing machines. By the nature of the halting criterion, the time and space functions of a BTM must be non-decreasing functions. We show that a time-complexity hierarchy also exists for translations realized by BTM's. Essential in the proof is the concept of time-constructibility (see Balcázar et al.[1]). A function f is *time-constructable* if there is a Turing machine that can produce an output of $f(n)$ zeroes if it is given an input of n zeroes, using $O(f(n))$ steps. We restrict ourselves to time-constructable functions that are in $\Omega(n)$.

Proposition 5.17. *Let f and g be non-decreasing functions such that $f \log f$ is in $o(g)$. Suppose that the function $n \mapsto g(n) - g(n-1)$ is time-constructable. Then there is a language L that can be decided by a BTM M running in time g, but not by any BTM running in time f.*

Proof. Define the function h by $h(0) = g(0)$ and $h(n+1) = g(n+1) - g(n)$. Let M_h be a machine that executes $h(n)$ steps on inputs of length n. Since h is time-constructable, such a machine exists.

We construct a Turing machine M that is defined on all inputs. It halts when the blank symbol on its input tape is reached and runs in time $O(g)$. The language it decides is the language L we seek.

Consider a string of the form $x = 0^{|w|}1wu$, for finite strings w and u. Let n be the length of x. If w is the encoding of a two-tape Turing machine M_w, then M simulates M_w either until M_w halts, or until a blank on the input of M is reached, or until $g(n)$ steps have been simulated, whichever comes first. When the simulation stops because of one or more of the criteria is satisfied, the input is rejected iff the simulation of M_w ended in an accepting state.

In addition to the work tapes used for the executions of M_w and M_h, a number of special tapes is used by M:

- a tape T_1 that stores the string w, used to simulate M_w,
- a tape T_2 that contains a partial copy of the input, which is used as input to M_w and
- an initially empty tape T_3 that measures the time used for the simulation.

When the machine is given a string as input, the first thing it does is move the input head to the right, counting the number of zeroes before the first 1 is located on the tape. Then, an equal number of symbols is copied to T_1. At the same time, a copy of the input read up to now is placed on T_2. This part takes $O(n)$ time. If a blank symbol on the input tape is reached before this part is done, the machine accepts the input. Otherwise, the tape T_1 contains the string w, the tape T_2 contains $0^{|w|}1w$ and the tape T_3 is still empty.

Now, the machine starts to simulate M_w using T_2 as input and M_h with T_3 as input. The machine uses the scheme described in the code fragment of Algorithm 5.1.

```
while ( true ) do
    if M_w halts then
        stop M_h
        append-input-to-T_2
    elseif M_h halts then
        stop M_w
        if T_2 and T_3 have the same length then
            append-input-to-T_2
        endif
        append 0 to T_3
    endif
    resume M_w and M_h
enddo
```

Algorithm 5.1. Simulate M_w for $O(f)$ steps.

The call to `append-input-to-T_2` is implemented as follows: first, the position of the head of T_2 on the tape is marked. Then, the next symbol from the input tape is read. If it is blank, then the input is rejected iff M_w is in an accepting state. Otherwise, the symbol is appended to the end of T_2. After that, the head of T_2 is moved back to its previous position and the marker is removed.

Adding a symbol to the end of T_2 takes $O(n)$ time. Testing if T_2 and T_3 have the same length also takes $O(n)$ time. Observe that the algorithm halts when the length of T_2 is increased for the $(n+1)$-st time. Since the length of T_2 is bounded by n and the length of T_3 is bounded by T_2, these operations can occur at most $2n$

times, amounting to $O(n^2)$ time. The running time is bounded by the time it takes to execute M_h on the contents of T_3, ranging in length from 0 to n. This takes $O\left(\sum_{i=0}^{n} h(n)\right)$ time, which equals $O(g)$. Observe that $n \in O(h)$, which implies that $n^2 \in O(g)$. Thus, the total time adds up to $O(g)$.

Now, suppose that L can be decided by a Turing machine running in time f that halts when it encounters a blank on the input tape. Then, there is a two-tape Turing machine that decides L in time $O(f \log f)$ (see Hennie and Stearns[17]). Let w be a description for this machine and consider a string $x = 0^{|w|}1w0^{\star}$ of length n, such that $f(n) \log f(n) < g(n)$. The simulation of M_w is set up in such a way, that the computation on $x_{[1:i]}$ can be extended to the computation on $x_{[1:i+1]}$, for all $0 \leq i < n$. Thus, M will simulate the operation of M_w on x for $O(g)$ steps. Since n is big enough, M can simulate $f(n)$ steps of M_w, which is enough to finish the simulation of M_w. But then, $x \in L$ iff M accepts x iff M_w rejects x iff $x \notin L$. This is a contradiction, so L cannot be decided by a Turing machine running in time f that halts when it encounters a blank on the input tape.

□

Observe that the condition that the function $n \mapsto g(n) - g(n-1)$ is time-constructable implies that $n^2 \in O(g)$. If g is a function in $o(n^2)$, then we have to let go of the restriction that time-constructable functions are in $\Omega(n)$. In this case, proofs using the ideas of Proposition 5.17 fail to work. As a special case, we can prove a similar result when $f \in o(n)$ and $g \in \Omega(n)$.

Proposition 5.18. *Let f and g be non-decreasing functions such that $f \in o(n)$ and $g \in \Omega(n)$. Then there is a language L that can be decided by a BTM M running in time g, but not by any BTM running in time f.*

Proof. Consider a BTM M that accepts an input x iff x contains only 1's. The last bit can be checked in linear time, so M works in time g. If a BTM M' operates in time f, then there are integers n such that inputs of length n cannot be processed whole by M'. Thus, there are parts of inputs that cannot be read by M'. If M' accepts these inputs, then it fails to reject inputs which contain 0's at these places. If M' rejects these inputs, then it fails to accept inputs which contain 1's at these (as well as at all other) places. It follows that M' cannot decide the same language as M. This proves the result.

□

It is an open question if there are other techniques to prove the hierarchy for functions f and g such that both f and g are in $o(n)$, or both f and g are in $\Omega(n) \cap o(n^2)$.

Similar to Proposition 5.17, one can also prove the existence of a space-complexity hierarchy. A function f is *space-constructable* if there is a Turing machine that can produce $f(n)$ zeroes on an input on n zeroes, using $O(f(n))$ space.

Proposition 5.19. *Let f and g be non-decreasing functions such that f is in $o(g)$. Suppose that g is space-constructable. Then there is a language L that can be decided by a BTM M running in space g, but not by any BTM running in space f.*

Proof (Sketch). The essence of the proof uses the same concept as that of Proposition 5.17, i.e., construct a BTM M that uses a machine M_g to measure a tape of length $g(1)$, $g(2)$, ... This tape may be used in the simulation of a machine M_w on the prefixes of length 1, 2, ... of inputs of the form $0^{|w|}1wu$. Note that in this case, the execution of M_w and M_g should not be performed parallel, since the tape has just one tape head. Fortunately, this has no impact on the space usage of M. The result of the simulation is then used to reject the input iff the simulation accepted its input. This machine accepts a language L in space $O(g)$. Using a standard diagonalizing argument, L cannot be accepted by any BTM using space $O(f)$. The actual construction of M, as well as the details of the diagonalizing argument, are left to the reader.

□

5.3 Properties of Interactively Realizable Translations

In this section, we show the basic properties of the class of translations that are interactively realized by ITM's. First, we give a relationship to partial recursive functions. Then, we show that the class is closed under composition and inversion.

5.3.1 Interaction and Recursion

There is a direct connection between interactively realizable translations and partial recursive functions.

Theorem 5.20. *Let $\Phi : \Sigma^\infty \to \Omega^\infty$ be a translation with domain D and define the partial function $\Phi' : \Sigma^\star \to \Omega^\star$ by letting $\Phi'(x) = \Phi(x)$ for every finite string x. Then, Φ is interactively realizable iff*

- *$|\Phi(x)| = |x|$ for every $x \in D$,*
- *if u is a prefix of $x \in D$, then $u \in D$ and $\Phi(u)$ is a prefix of $\Phi(x)$.*
- *D is a closed set,*
- *Φ' is partial recursive.*

We prove it with the following Propositions.

Proposition 5.21. *Let Φ be an interactively realizable translation with domain D. Then $|\Phi(x)| = |x|$ for every $x \in D$.*

Proof. This follows immediately from the observations in Remark 5.4 and Definition 5.14.

□

Proposition 5.22. *Let Φ be an interactively realizable translation with domain D. If u is a prefix of $x \in D$, then $u \in D$ and $\Phi(u)$ is a prefix of $\Phi(x)$.*

Proof. Let Φ be interactively realized by the ITM M. Let $x \in D$. Then x is a valid input to M. If u is a prefix of x, then u is also a valid input. The result then follows from (5.1) and Definition 5.7 and 5.14.

□

Proposition 5.23. *Let Φ be an interactively realizable translation with domain D. Then D is a closed set.*

Proof. Let M be an ITM that interactively realizes Φ. Let x be a string in the complement of D. Then x is not a valid input to M.

Suppose that every finite prefix of x is a valid input to M. This implies that an efficient input for each of these prefixes can be processed by M. Following the proof of Proposition 5.11, the efficient inputs form a sequence of strings such that for every n, the n-th string is a prefix of the $(n+1)$-st string. Thus, the sequence converges to a string. We conclude that this string filters to x and can be processed by M, contradicting the assumption.

Therefore, there must be a finite prefix u of x that is not a valid input to M. Let y be a string in $\mathbb{B}(u)$. After processing the prefix u of y, the ITM M either enters an infinite loop or cannot change to a next state. Thus, y cannot be processed by M. We conclude that the open set $\mathbb{B}(u)$ does not intersect D, which implies that D is a closed set.

□

Proposition 5.24. *Let $\Phi : \Sigma^\infty \to \Omega^\infty$ be an interactively realizable translation. Define the partial function $\Phi' : \Sigma^\star \to \Omega^\star$ by letting $\Phi'(x) = \Phi(x)$ for every finite string x. Then Φ' is partial recursive.*

Proof. By Proposition 5.16, ITM's are equivalent to BTM's, thus Φ' can be realized by a BTM. A BTM is a Turing machine, so Φ' is partial recursive.

□

Proposition 5.25. *Let $\Phi : \Sigma^\infty \to \Omega^\infty$ be a translation with domain D and define the partial function $\Phi' : \Sigma^\star \to \Omega^\star$ by letting $\Phi'(x) = \Phi(x)$ for every finite string x. Suppose that*

- *$|\Phi(x)| = |x|$ for every $x \in D$,*
- *if u is a prefix of $x \in D$, then $u \in D$ and $\Phi(u)$ is a prefix of $\Phi(x)$.*
- *D is a closed set,*
- *Φ' is partial recursive.*

Then Φ is interactively realizable.

Proof. Let M' be a Turing machine that computes Φ'. We will construct an ITM M that uses M' to interactively realize Φ. Suppose the ITM M receives a string $x \in D$ at its input port. The computation of the output proceeds recursively. For any integer $0 \leq n < |x|$, the string $\Phi(x_{[1:n]})$ is produced at the output port, using a copy of $x_{[1:n]}$ on a special work tape. Observe that $x_{[1:0]}$ is the empty string, so initially the work tape contains $x_{[1:0]}$ and the string $\Phi(x_{[1:0]})$ is produced at the output port.

Suppose the work tape contains the string $x_{[1:n]}$, while the computation up to now has produced the string $\Phi(x_{[1:n]})$ at the output port. Assume that M is in an external configuration and that x_{n+1} appears at the input port. Then, M initiates an internal phase, in which it writes x_{n+1} to the end of the special work tape and uses this tape to simulate M' with input $x_{[1:n+1]}$, discarding the first n

output symbols. The next symbol is written to the output port, ending the internal phase.

Since $\Phi(x_{[1:n]})$ is a prefix of length n of the string $\Phi(x_{[1:n+1]})$ of length $n+1$, we conclude that M has produced $\Phi(x_{[1:n+1]})$ at the output port.

For an $x \in D$, every prefix of $\Phi(x)$ is produced at the output port in this fashion, so the ITM produces $\Phi(x)$. Suppose on the other hand that $x \notin D$. Since D is closed, there is a basis set $\mathbb{B}(u)$ that contains x, which does not intersect D. It follows that $u \notin D$. Hence $\Phi'(u)$ is undefined, so M' cannot process u. We conclude that M cannot process u, so u and x are not valid inputs to M. Thus, M interactively realizes Φ.

□

This concludes the proof of Theorem 5.20. Van Leeuwen and Wiedermann[27, 29] gave a different but related view on the connections between partial functions and interactive translations.

5.3.2 Operations on Interactively Realizable Translations

The class of interactively realizable translations is closed under a number of usual operations, such as composition and inversion.

Proposition 5.26. *Let $\Phi : \Sigma^\infty \to \Omega^\infty$ and $\Psi : \Omega^\infty \to \Theta^\infty$ be two interactively realizable translations. Then, the translation $\Psi \circ \Phi$ is also interactively realizable.*

Proof. Let M_Φ and M_Ψ be ITM's that interactively realize Φ and Ψ respectively. We design an ITM M that simulates M_Φ and M_Ψ. Let x be an input string. Suppose the prefix $x_{[1:n]}$ has been translated into the string $\Psi(\Phi(x_{[1:n]}))$ by M. Then M is in an external configuration. This configuration contains a representation of the configuration $\mathbf{q}$ that M_Φ is in after processing $x_{[1:n]}$, as well as a representation of the configuration $\mathbf{q}'$ that M_Ψ is in after processing $\Phi(x_{[1:n]})$, plus a special tape containing $\Phi(x_{[1:n]})$.

When M receives x_{n+1} at the input port, it simulates M_Φ internally, starting in the configuration $\mathbf{q}$, until it produces the next output symbol. At this time, the configuration of M contains a representation of the configuration M_Φ is in after processing $x_{[1:n+1]}$. The output symbol is written to the end of the special tape. During this simulation, the representation of $\mathbf{q}'$ has not changed. Now, M simulates M_Ψ using the configuration $\mathbf{q}'$, until it produces the next output symbol, which is written to the output port. The representation of the configuration of M_Φ has not changed, so the configuration of M again contains representations of the configurations of M_Φ and M_Ψ after processing $x_{[1:n+1]}$ and $\Phi(x_{[1:n+1]})$ respectively, plus a tape containing $\Phi(x_{[1:n+1]})$. Note that before the simulations of M_Φ and M_Ψ start, the configuration of M has the same property. Thus M is an ITM that interactively realizes $\Psi \circ \Phi$.

□

Proposition 5.27. *Let $\Phi : \Sigma^\infty \to \Omega^\infty$ be an injective interactively realizable translation with domain D. Then, the translation Φ^{-1} with domain $\Phi(D)$ is interactively realizable too.*

Proof. Let M be an ITM that interactively realizes Φ. Consider the transition function $\boldsymbol{\delta}$. Since Φ is injective, we have for every configuration $\mathbf{q}$ that $\boldsymbol{\delta}(\mathbf{q}, a) = (\mathbf{r}, b)$ and $\boldsymbol{\delta}(\mathbf{q}, a') = (\mathbf{r}', b)$ iff $a = a'$ and $\mathbf{r} = \mathbf{r}'$. Thus the transition function $\boldsymbol{\delta}'$, defined by $\boldsymbol{\delta}'(\mathbf{q}, a) = (\mathbf{r}, b)$ if $\boldsymbol{\delta}(\mathbf{q}, b) = (\mathbf{r}, a)$ and undefined otherwise, is well-defined.

The ITM M' that is obtained by replacing the transition function $\boldsymbol{\delta}$ by $\boldsymbol{\delta}'$, produces the string $\Phi^{-1}(y)$ on input y.

Let y be a string not in $\Phi(D)$. Then, there is a largest prefix $y_{[1:n]}$ of y that is in $\Phi(D)$. Suppose u causes M to produce $y_{[1:n]}$, ending in the configuration $\mathbf{q}$. Since M cannot produce the string $y_{[1:n+1]}$, there is no transition from $\mathbf{q}$ with output y_{n+1}. Thus, the transition $\boldsymbol{\delta}'(\mathbf{q}, y_{n+1})$ is undefined. Now, the input $y_{[1:n]}$ causes M' to enter the configuration $\mathbf{q}$, which implies that the string $y_{[1:n+1]}$ cannot be processed by M'. Thus, y is not a valid input to M'. It follows that M' interactively realizes Φ^{-1}.

□

5.4 The Complexity of Interactively Realizable Translations

Just like classical machine models, we expect ITM's to operate as efficiently as possible. The efficiency of an ITM is measured by observing the time an ITM needs to respond to a non-λ input symbol. Since this time may also depend on the configuration the ITM is in, the time needed to reach this configuration is also included. ITM's are deterministic machines, so the input completely determines which configuration an ITM will be in at any time. However, time-outs that are not initiated by the ITM's, i.e., time-outs caused by the environment writing external λ's to the input port, introduce delays into the response times, for which the ITM really can't be blamed. Thus, efficient inputs are used to measure computation times.

Definition 5.28. *Let M be an ITM and let x be a finite string that is a valid input to M. Then, $t(x)$ is the length of an efficient input for x. The time complexity function $T : \mathbb{N} \to \mathbb{N}$ is defined by letting $T(n)$ be the maximum of $t(x)$ over all valid inputs x of length n. A translation Φ is of time complexity T if there is an ITM M that interactively realizes Φ using at most time T. The time complexity class $ITIME(T)$ is defined as the class of interactively realizable translations of time complexity T.*

Note that $T(n)$ measures the maximum time it takes to produce n non-λ symbols at the output port, when there are no external λ's involved. The response time mentioned earlier can easily be computed by taking an input x which yields a configuration $\mathbf{q}$ and computing the maximum value of

$$\{\ t(xa) - t(x) \quad | \quad a \in \Sigma\ \}\ . \tag{5.3}$$

Since t depends on the transition function, it follows that (5.3) does not depend on the choice of x, but on the configuration $\mathbf{q}$, which is what we wanted.

Similar to the time complexity, the efficiency of an ITM can also be measured by its space usage.

Definition 5.29. *Let M be an ITM and let x be a finite string that is a valid input to M. Then, $s(x)$ is the length of the used part of the work tapes after reading an efficient input for x. The space complexity function $S : \mathbb{N} \to \mathbb{N}$ is defined by letting $S(n)$ be the maximum of $s(x)$ over all valid inputs x of length n. A translation Φ is of space complexity S if there is an ITM M that interactively realizes Φ using at most space S. The space complexity class $ISPACE(S)$ is defined as the class of interactively realizable translations of space complexity S.*

In other words, $S(n)$ measures the maximum space needed to produce n non-λ symbols at the output port, when there are no external λ's involved.

5.4.1 A Complexity Hierarchy

Just like the space complexity defined for classical Turing machines, the space complexity of ITM's induces a hierarchy on the translations that are interactively realized by ITM's. The hierarchy results are based on the complexity hierarchy for BTM's.

Proposition 5.30. *Let Φ be an interactively realizable translation and define the partial function $\Phi' : \Sigma^\star \to \Omega^\star$ by letting $\Phi'(x) = \Phi(x)$ for every finite string x. Then, there is an ITM M running in time $O(T)$ and space $O(S)$ that interactively realizes Φ iff there is a BTM M' running in time $O(T)$ and space $O(S)$. that realizes Φ.*

Proof. Given a BTM M' that realizes Φ', the ITM M from the proof of Proposition 5.16 interactively realizes Φ. Define the function f by $f(0) = T(0)$ and $f(n+1) = T(n+1) - T(n)$. Let T' be the time that M needs. In this case, M needs $T'(n)$ to produce the first output n symbols, plus an additional $O(f(n+1))$ to produce the $(n+1)$-st symbol. Solving the recursion $T'(n+1) = T'(n) + O(f(n+1)$, it follows that $T' \in O(T)$. In other words, M works in time $O(T)$. The space used by M to produce the first n symbols can be reused to produce the $(n+1)$-st symbol. Thus, M uses space $O(S)$.

Given an ITM M that interactively realizes Φ, the BTM M' from the proof of Proposition 5.16 realizes Φ'. The simulation of M by M' takes $O(T)$ time and $O(S)$ space.

□

Using Proposition 5.30, we can establish a time and space complexity hierarchy for interactively realizable translations.

Proposition 5.31. *Let f and g be non-decreasing functions such that $f \log f \in o(g)$. Then there is a translation Φ that can be interactively realized by an ITM running in time $O(g)$, but not by any ITM running in time $O(f)$.*

Proof. By Proposition 5.17, there is a language L that can be decided by a BTM in time g, but not by any BTM running in time $O(f)$. Consider the translation Φ, defined by

$$(\Phi(x))_n = \begin{cases} 1 \text{ if } x_{[1:n]} \in L \\ 0 \text{ otherwise} \end{cases} . \tag{5.4}$$

Let Φ' be the function defined by $\Phi'(x) = \Phi(x)$ for every finite string x. Clearly, there is a BTM M' that realizes the function Φ' in time $O(g)$. By Proposition 5.30, there is an ITM M that interactively realizes Φ in time $O(g)$.

On the other hand, if an ITM interactively realizes Φ in time $O(f)$, then, again by Proposition 5.30, there is a BTM that realizes Φ' in time $O(f)$ and this BTM can be used to construct a BTM that decides L in time $O(f)$, which contradicts the choice of L. Thus, Φ cannot be interactively realized by an ITM in time $O(f)$. □

Proposition 5.32. *Let f and g be non-decreasing functions such that $f \in o(g)$. Then there is a translation Φ that can be interactively realized by an ITM running in space $O(g)$, but not by any ITM running in space $O(f)$.*

Proof. By Proposition 5.19, there is a language L that can be decided by a BTM in space g, but not by any BTM running in space $O(f)$. Consider the translation Φ, defined by

$$(\Phi(x))_n = \begin{cases} 1 \text{ if } x_{[1:n]} \in L \\ 0 \text{ otherwise} \end{cases} . \tag{5.5}$$

By Proposition 5.30, Φ can be interactively realized by an ITM in space $O(g)$. Similarly, by Proposition 5.30, Φ cannot be interactively realized by any ITM in space $O(f)$, since this would imply that L can be decided by a BTM in space $O(f)$.

□

5.4.2 Properties of Complexity Classes

In section 5.3, some closure properties of classes of interactively realizable translations were examined. Here, the same properties are examined for complexity classes.

First, we look at composition. Let M be an ITM that interactively realizes the composition of two interactively realizable translations, of time (or space) complexity f and of time (or space) complexity g, respectively. From the proof of Proposition 5.26, it follows that M has time (or space) complexity $O(f + g)$.

Corollary 5.33. *Consider a time (or space) complexity class of interactively realizable translations. If two translations belong to this class, then so does their composition.*

Next, we look at inversion. The modified ITM from the proof of Proposition 5.27 has the same time and space complexity as the ITM we started with.

Corollary 5.34. *Consider a time (or space) complexity class of interactively realizable translations. If a translation from this class is invertible, then its inverse belongs to this class as well.*

5.5 Non-uniform Interactive Turing Machines

The interactive Turing machine model attempts to model real-life computing systems. For this reason, it was given the ability to extend its computations indefinitely. While this adds a whole new dimension to the environment that the machines can interact with, the basic computational power of the machines remains unchanged. Another aspect of real-life computing systems is their ability to evolve throughout their existence, while retaining their knowledge of computations. This aspect is modeled by extending ITM's with an advice mechanism. This mechanism comes in the form of an advice function. Recall the definition of an advice function from Chapter 3. The mechanism uses a special advice tape. The advice tape is a read-only tape with one head. When the n-th input symbol is read from the input port, the n-th value of the advice function is appended to the end of the advice tape, separated by a special marker symbol. This happens automatically in one step. The head of the tape is not needed for this operation and can remain where it was before the internal phase started.

Since advice functions can be given non-uniformly, ITM's using an advice mechanism are capable of "breaking the Turing barrier": they can compute translations that cannot be realized by Turing machines. This implies that they are more powerful than ITM's without advice mechanisms. For this reason, we cannot with good conscience say that an ITM with advice (ITM/A) "interactively" realizes a translation. Thus, we define the concept of "non-uniformly" realizable translations.

Definition 5.35. *Let M be an ITM/A with advice function α. The partial function $\Phi^{M:\alpha}$ is defined by letting $\Phi^{M:\alpha}(x)$ be the output of M on x using the advice $\alpha(|x|)$ if x is a valid input to M and undefined otherwise, for every string x. A translation Φ is* non-uniformly *realizable if there is an ITM/A M with an advice function α such that Φ equals $\Phi^{M:\alpha}$.*

Proposition 5.36. *Uncountably many non-uniformly realizable translations cannot be interactively realized.*

Proof. Let S be an undecidable set of integers and let χ_S be its characteristic function. Consider the translation Φ, defined by $(\Phi(x))_n = \chi_S(n)$ for every string x. Clearly, there is an ITM/A that non-uniformly realizes Φ using the advice function χ_S. However, if an ITM M would realize Φ, then M could be simulated by a Turing machine and used to decide S. Since there are uncountably many undecidable sets of integers, the result follows.

□

One of the reasons that out of all available models, the ITM/A's have grown so popular (see Van Leeuwen and Wiedermann[28, 30] and Wiedermann and Van Leeuwen[49, 50]), is the fact that their close relation to Turing machines allows us to easily define complexity measures on the ITM/A's. This allows us to look at the efficiency of other models, using the relation to ITM/A's as an indirect measure. When it is not so obvious to define a measure of complexity on a model, this indirect approach can yield some insights. We will use ITM/A's in this sense in Chapter 7. In addition to the time and space complexity measures defined on ITM's, the advice function can be used as a complexity measure as well.

Definition 5.37. *Let $f : \mathbb{N} \to \mathbb{N}$ be a function. An ITM/A with advice function α is of advice complexity f if the length of $\alpha(n)$ is at most $f(n)$ for every n.*

To determine the space complexity of an ITM/A, the advice tape should not be taken into account. The advice tape is seen more as an extension to the input tape than as a tape on which the machine may write arbitrary information. In fact, the advice tape is read-only, its contents depend at all times entirely on the length of the input-prefix processed thus far.

Similar to the case of classical Turing machines with advice, there is a complexity hierarchy of translations that are realized by an ITM/A. Proving this is similar to the ideas set forth in Chapter 3, with some additional techniques.

Definition 5.38. *Let CST be an ITM/A with input alphabet Σ, output alphabet $\{0, 1\}$ and advice alphabet Ω. Suppose CST uses an advice function α. For an input x, the n-th output symbol is a 1 iff the tuple $\langle x_{[1:n]}, \alpha(n)\rangle$ is in $CS_{\Sigma,\Omega}$ (see Definition 3.5).*

Proposition 5.39. *Let Σ be an input alphabet of size c and Ω an advice alphabet of size d. Let g be an integer-valued function such that $g(n) \leq c^n / \log d$ for all n. Let m and k be two integers. Let α be an advice function of size g over the alphabet Ω, such that $\alpha(n)$ is fixed when $n < m$. Then, there are $\prod_{n=m}^{k} d^{g(n)}$ translations that can be non-uniformly realized by CST with an advice function α, such that the translations all behave differently on inputs with lengths between m and k.*

Proof. The set of inputs of length n which cause CST to output a 1 is completely determined by the choice of $\alpha(n)$. Since the size of $\alpha(n)$ is at most $c^n / \log d$, it follows that $\alpha(n)$ corresponds to a characteristic string of length at most c^n (see Proposition 3.9). Therefore, each choice of $\alpha(n)$ corresponds to a translation that behaves different from the translations corresponding to other choices for $\alpha(n)$ on inputs of length n. It follows that CST can decide $\prod_{n=m}^{k} d^{g(n)}$ different translations with different choices of advice function. Furthermore, the differences all manifest themselves for inputs of length between m and k.

□

Proposition 5.40. *Let Σ be an alphabet of size c and Θ an advice alphabet of size b. Let f be an integer-valued function and let k be an integer. Then, an arbitrary ITM/A with an advice function of size at most f over the alphabet Θ can realize at most $\prod_{n=1}^{k} \left(\sum_{i=0}^{f(n)} b^i \right)$ translations that behave differently on inputs of length k or less.*

Proof. The behavior of the translations on strings of length k or less is completely determined by the choice of the advice strings for inputs of length 1 to k. Thus, the number of different translations is at most the product of the number of different advice strings for strings of length 1 to k. For the advice for inputs of length n, the number of choices is $\sum_{i=0}^{f(n)} b^i$. Thus, the total number of different advice functions is $\prod_{n=1}^{k} \left(\sum_{i=0}^{f(n)} b^i \right)$.

□

Observe that for any $m \leq k$, the number of translations that behave differently for strings of length between m and k is at most the number of translations that behave differently for strings of length k or less.

Let M be an ITM/A. Consider the following inequality.

$$\prod_{n=1}^{k} \left(\sum_{i=0}^{f(n)} |\Theta|^i \right) < \prod_{n=m}^{k} d^{g(n)} \ . \tag{5.6}$$

When it holds, there is a translation that can be non-uniformly realized by CST with an advice of size g over an alphabet of size d, but not by M with an advice of size at most f over the alphabet Θ. Inequality (5.6) is the basis for the next results. Basically, for every combination of an ITM/A and an advice size, a suitable interval $[m, k]$ has to be found. Furthermore, these intervals have to be mutually disjoint.

Lemma 5.41. *Let the following be given:*

- Σ *is an input alphabet of size* c,
- Ω *is an advice alphabet of size* d,
- $\mathcal{A}$ *is a class of allowed advice alphabets,*
- g *is an integer-valued function,*
- $\mathcal{F}'$ *is a countable class of integer-valued functions,*
- Φ *is a translation that can be non-uniformly realized by* CST *with an advice of size* g,
- N *is an integer.*

Define the class $\mathcal{F}$ *by:*

$$\mathcal{F} = \{\ f \quad | \quad \exists f' \in \mathcal{F}'\ f(n) = f'(n) \text{ for all but finitely many } n\ \} \ . \tag{5.7}$$

Suppose $g(n) \leq c^n / \log d$ *for every* n. *Suppose that for every* $f \in \mathcal{F}$ *and every* $\Theta \in \mathcal{A}$ *there are infinitely many disjoint intervals* $[m, k]$ *such that (5.6) holds. Then, there is an non-uniformly realizable translation* Φ' *that can be decided by an ITM/A with an advice of size* g *over the alphabet* Ω, *but not by any ITM/A with an advice of size bounded by a function* $f \in \mathcal{F}$ *over an alphabet in* $\mathcal{A}$. *Furthermore,* $\Phi'(x) = \Phi(x)$ *for all inputs* x *with length* N *or less.*

Proof. Observe that $\mathcal{F}$ is countable if $\mathcal{F}'$ is countable. We construct an advice function α of size g, to be used by CST to non-uniformly realize a translation Φ.

Consider an enumeration of all tuples of the form (f, b, M) with a function $f \in \mathcal{F}$, an integer b and an ITM/A M. To each tuple, an interval $[m, k]$ is assigned such that (5.6) holds for f and any advice alphabet $\Theta \in \mathcal{A}$ with $|\Theta| = b$. The interval $[m, k]$ is chosen such that it does not intersect the intervals that were assigned to previous tuples in the enumeration. This is possible since there are infinitely many disjoint intervals for every f and b. The interval is chosen such that $N < m$.

Assume that $\alpha(n)$ is defined for all $n < m$ at this point. If $\alpha(n)$ has not been fixed for an $n < m$, we choose the advice string that was used to non-uniformly realize Φ on inputs of length n. By combining (5.6) and Propositions 5.39 and 5.40,

it follows that there are choices for $\alpha(n)$ with $m \leq n \leq k$, such that the translation that is non-uniformly realized by CST with α differs on inputs of length between m and k from all translations that can be non-uniformly realized by M with an advice of size at most f over an alphabet of size b.

Let Φ' be the translation that is non-uniformly realized by CST with the advice function α. It follows that $\Phi'(x) = \Phi(x)$ for all x such that the length of x is not in an interval $[m, k]$ for a tuple (f, b, M). In particular, if the length of x is N or less, then $\Phi'(x) = \Phi(x)$.

Suppose that Φ' can be non-uniformly realized by an ITM/A M with an advice β of size bounded by a function $f \in \mathcal{F}$, over an alphabet $\Theta \in \mathcal{A}$. Then, $|\beta(n)| \leq f(n)$ for every n. Let b be the size of Θ and let $[m, k]$ be the interval corresponding to the tuple (f', b, M). Consider the behavior of Φ' on inputs of length between m and k. By definition, α was chosen such that the translation that is non-uniformly realized by CST with advice function α behaves differently from the translation that is non-uniformly realized by M with the advice function β. This implies that Φ' behaves differently from Φ', which is a clear contradiction. Thus, Φ' cannot be non-uniformly realized by any ITM/A with an advice function bounded by $f \in \mathcal{F}$ over an alphabet $\Theta \in \mathcal{A}$.

□

Theorem 5.42. *Let Σ be an input alphabet of size c and Ω an advice alphabet of size $d > 1$. Let g and f be integer-valued functions such that f is in $o(g)$ and $g(n) \leq c^n / \log d$. Then, there is a translation that can be non-uniformly realized by an ITM/A with advice of size g, but not by any ITM/A with an advice of size f', where f' is a function such that $f'(n) \leq f(n)$ for all but finitely many n.*

Proof. Let $\mathcal{F} = \{\ f' \quad | \quad f'(n) = f(n) \text{ for all but finitely many } n\ \}$. We will show that there are infinitely many intervals $[m, k]$ for which (5.6) holds for all $f' \in \mathcal{F}$ and every advice alphabet Θ. The result then follows from Lemma 5.41. Observe that (5.6) is equivalent to

$$\sum_{n=1}^{k} \log \left(\sum_{i=1}^{f(n)} |\Theta|^i \right) < \sum_{n=m}^{k} g(n) \log d \ . \tag{5.8}$$

Since f is in $o(g)$, there is an m such that $f(n) < \frac{\log d}{2 \log |\Theta| + 3} g(n)$ for all $n \geq m$. There are three distinct cases for $n \geq m$:

- If $f(n) = 0$, then

$$\log \left(\sum_{i=0}^{f(n)} |\Theta|^i \right) = 0 \leq g(n) \log d - 1 \ . \tag{5.9}$$

- If $f(n) \geq 1$ and $|\Theta| = 1$, then $2f(n) + 1 \leq 3f(n) < g(n) \log d$. Thus, the following inequality holds:

$$\log \left(\sum_{i=0}^{f(n)} |\Theta|^i \right) = \log \left(f(n) + 1 \right) \leq 2f(n) < g(n) \log d - 1 \ . \tag{5.10}$$

- If $f(n) \geq 1$ and $|\Theta| \geq 2$, then $2f(n)\log|\Theta|+1 < g(n)\log d$. Thus, the following inequality holds:

$$\log\left(\sum_{i=0}^{f(n)}|\Theta|^i\right) < \log\left(|\Theta|^{f(n)+1}\right) \leq 2f(n)\log|\Theta| < g(n)\log d - 1 \ . \quad (5.11)$$

Let $S = \sum_{n=1}^{m-1}\log\left(\sum_{i=1}^{f(n)}|\Theta|^i\right)$. Then, the following inequality holds:

$$\sum_{n=1}^{k}\log\left(\sum_{i=1}^{f(n)}|\Theta|^i\right) \leq S + \sum_{n=m}^{k}(g(n)\log d - 1) \ . \quad (5.12)$$

Thus, if we choose k such that $k > S + m + 1$, then (5.8) holds.

For every $f' \in \mathcal{F}$ there is an N such that $f'(n) = f(n)$ for all $n \geq N$. Thus, if we let $m > N$, then there is a k such that (5.6) also holds for this f'. Finally, note that we can choose an $m > k$ and get a new, disjoint interval for which (5.6) holds. This procedure can be repeatedly infinitely often, giving us infinitely many disjoint intervals for which (5.6) holds.

□

Theorem 5.43. *Let Σ be an alphabet of size c and Ω an advice alphabet of size d. Let $\mathcal{A}$ be a class of allowed advice alphabets. Suppose that $|\Theta| \leq d$ for every $\Theta \in \mathcal{A}$. Let f and g be integer-valued functions such that $g(n) \leq c^n/\log d$ for all n. If $d > 2$ and $f(n) < g(n)$ for all but finitely many n, or $d = 2$ and $f(n) < g(n) - 1$ for all but finitely many n, then there is a translation that can be non-uniformly realized by an ITM/A with an advice of size g, but not by any ITM/A with an advice of size at most f.*

Proof. Let $\mathcal{F} = \{\ f' \quad | \quad f'(n) = f(n)$ for all but finitely many $n\ \}$. We will show that there are infinitely many intervals $[m,k]$ for which (5.6) holds for all $f' \in \mathcal{F}$ and every $\Theta \in \mathcal{A}$.

Since $|\Theta| \leq d$ and $d > 1$, it follows that

$$\prod_{n=1}^{k}\left(\sum_{i=0}^{f(n)}|\Theta|^i\right) \leq \prod_{n=1}^{k}\left(\sum_{i=0}^{f(n)}d^i\right) = \prod_{n=1}^{k}\frac{d^{f(n)+1}-1}{d-1} \ . \quad (5.13)$$

Let $P_m = \prod_{n=1}^{m-1}\left(d^{f(n)+1}-1\right)/(d-1)$. Note that P_m is a constant if m is fixed. Then the following holds:

$$\prod_{n=1}^{k}\frac{d^{f(n)+1}-1}{d-1} = P_m \cdot \prod_{n=m}^{k}\frac{d^{f(n)+1}-1}{d-1} \ . \quad (5.14)$$

There are two distinct cases:

- If $d \geq 3$ and m is chosen such that $f(n) \leq g(n) - 1$ for all $n \geq m$, then:

$$P_m \cdot \prod_{n=m}^{k}\frac{d^{f(n)+1}-1}{d-1} \leq P_m \cdot \prod_{n=m}^{k}\frac{d^{f(n)+1}-1}{2} < \frac{P_m}{2^{k-m+1}} \cdot \prod_{n=m}^{k}d^{g(n)} \ . \quad (5.15)$$

- If $d = 2$ and m is chosen such that $f(n) \leq g(n) - 2$ for all $n \geq m$, then:

$$P_m \cdot \prod_{n=n}^{k} \frac{d^{f(n)+1} - 1}{d - 1} = P_m \cdot \prod_{n=m}^{k} \frac{d^{f(n)+2} - d}{d} < \frac{P_m}{2^{k-m+1}} \cdot \prod_{n=m}^{k} d^{g(n)} \quad . \quad (5.16)$$

For every m, there is an integer k such that $P_m / \left(2^{k-m+1}\right) \leq 1$, which implies that (5.6) holds in either case. By choosing m large enough, it follows that (5.6) holds for every $f \in \mathcal{F}$ for infinitely many intervals. The result then follows from Lemma 5.41.

□

5.6 Conclusions

In this Chapter, we explored ITM's and the translations they produce as a model for interactive evolving systems. The ITM was first introduced by Van Leeuwen and Wiedermann[28, 30]. An important aspect of interactive systems is their ability to arbitrarily extend computations. We modeled this aspect by using input streams instead of the classical tapes. These streams necessitated the use of symbols to act as placeholders when no information occurred in the stream: the λ-symbols. The λ-symbols introduced two different types of strings, i.e., the inputs (strings without λ -symbols) containing the information and the streams (strings with λ-symbols) that carried the input strings. Some care needed to be given to make sure that different streams carrying the same input actually produced the same output. To achieve this, efficient inputs were introduced. In addition to being useful to show correctness of definitions, they also proved useful in setting up the complexity theory based on ITM's.

A different approach to extending computations that we also explored briefly is to halt a machine when the end of its input is reached, the BTM's. This way, a computation can be easily extended by extending the input and continuing the computation in the configuration the machine was in when it halted. We showed that these two approaches are equivalent in the sense that they produce the same translations. By using BTM's, we don't have to use λ-symbols, which simplifies the definitions a lot. The ideas of BTM's, especially in connection with advice mechanisms, can be explored further. In fact, the BTM can be posed as another model for evolving interactive systems. An important difference is that BTM's are two-way machines, while most other models for evolving interactive systems are one-way machines.

We showed that the translations that are interactively realized by ITM's are the recursive functions operating on infinite domains. Indeed, the main difference between classical Turing machines and ITM's lies in the allowed domains. As a consequence, many of the results for recursive functions also hold for the interactively realizable translations. We showed this for composition and inversion of functions.

In setting up the time and space complexity hierarchies, there was an issue that complicated matters. To determine the amount of resources needed to process an input of length n, the resources needed to process inputs of smaller lengths had

to be included too. This is not a problem for the space complexity hierarchy, since the tapes can be reused. But for the time complexity hierarchy, this implies comparing $\sum_{i=1}^{n} f(i)$ to $g(n)$, instead of $f(n)$ to $g(n)$, for functions f and g. We solved this by using the difference function $n \mapsto g(n) - g(n-1)$ and the concept of time-constructable functions. However, to reasonably deal with time-constructable functions, we need them to be in $O(1) \cup \Omega(n)$. If g is an integer-valued function in $\Omega(n) \cap o(n^2)$, its difference function is in $\Omega(1) \cap o(n)$, which is a problem. It is an open question if we can use other techniques to say anything in this case.

By enhancing ITM's with an advice mechanism, we can use them to model evolving systems. The use of advice functions gives ITM's the ability to "break the Turing barrier" basically for free. The running time and space usage of an ITM/A are defined the same as that of an ITM. Since the advice mechanism uses a different tape to store the advice, this implies that the length of the advice is not included in the input length. Thus, the advice length does not influence the running time and space usage. This is in contrast with the classical model that Karp and Lipton[22] proposed, where the advice length *does* influence the running time and space usage.

The advice functions add a new resource to ITM's and thus a new complexity hierarchy to establish. The method to prove the existence of the advice hierarchy uses the ideas of Chapter 3, but again we have to deal with the resources needed to process inputs of smaller lengths. We solved the issue here by adjusting the interesting cases. In Chapter 3, an interesting case was the combination of machine, advice length and a single input length. Here, we instead look at ranges of lengths. For the case of unbounded advice alphabet sizes, the result then holds unaltered. For the case of bounded alphabets, we needed an advice alphabet size of at least three to prove a result similar to that of Theorem 3.22. We could not show that (5.6) holds when $d = 2$ and $f(n) < g(n)$ for all but finitely many n (the reader is encouraged to try it himself). It is of course possible that because of the added problem of needing to include the resources for inputs of lengths $< n$, the result of Theorem 5.43 cannot be improved for $d = 2$. Either way, this remains an open question. For $d = 1$, the problem is similar to that of $d = 2$. In this case, we can show that the result holds if there is a fixed $\epsilon > 0$ such that $f(n) < (1+\epsilon)g(n)$ for all but finitely many integers n.

CHAPTER 6

Lineages of Automata

In this Chapter, we define lineages of automata, a model designed to capture the evolving aspect of computational systems in a natural way. This model, inspired by notions from evolutionary biology, was initially outlined by Van Leeuwen and Wiedermann[30]. It is based on the idea that systems evolve in stages during their existence, with minimal assumptions about the underlying mechanisms. It turns out that even this simple model is more powerful than classical Turing machines, when cast in computational terms. This was initially observed by Van Leeuwen and Wiedermann[28, 30], when they showed that lineages of automata are equivalent to interactive Turing machines with advice (see also Chapter 7). The latter machines are known to possess super-Turing computing power. Here, we develop the theory of lineages in detail.

A lineage is a sequence of interactive, finite automata with a mechanism of passing information from each automaton to its immediate successor and the potential to process infinite input streams. Every automaton in the sequence can be seen as a temporary instantiation of the modeled system, before it changes into the next automaton. We study the properties of lineages through the translations they realize. Lineages of interactive finite automata (or transducers) have been introduced by Van Leeuwen and Wiedermann[30].

The concept of transducers acting on infinite input streams (ω-transducers) is not new. For example, Thomas[41] gives an overview of the theory of finite devices that operate on infinite objects. In the field of non-uniform complexity theory, sequences of computing devices are common-place (see e.g. Balcázar et al.[1] and Chapters 3 and 4). It is the idea of combining these concepts and allowing some form of communication between the devices in the sequence that is new. It allows for a closer modeling of the evolution of a system. The approach leads to several new fundamental questions that will be settled in this Chapter and the next.

The structure of the Chapter is as follows. First, we define lineages. Next, we give some effective constructions to produce new lineages out of given ones.

Then, we look at the class of translations that can be computed by lineages, and show that this class is much richer than any class that is realized by non-evolving finite-state machines. We give a useful characterization of non-uniformly realizable translations in terms of their domain. Finally, we show that the class of translations is closed under composition and under inversion.

Van Leeuwen and Wiedermann[28, 30, 49, 50] showed that several other well-motivated models are equivalent to interactive Turing machines with advice and thus also equivalent to lineages of automata. This implies that lineages are firmly embedded in the family of new models proposed to fill the gap between the classical Turing machine model and its equivalents on the one hand, and the super-Turing behavior of many real-life applications on the other (see Etesi and Németi[12], Wiedermann and Van Leeuwen[51] and Wegner and Goldin[48] for examples of real-life applications). Between all these new models, lineages stand out because they demonstrate the aspect of evolution directly, rather than indirectly through an advice mechanism. Furthermore, just as (interactive) finite automata are a fundamental model of computation, so are sequences of automata, and hence lineages, a fundamental model of evolving interactive computing.

6.1 Lineages

The building blocks of the model are automata that are a generalization of Mealy automata. These automata process potentially infinite input streams and produce potentially infinite output streams, one symbol at a time. We assume that there is no input tape. Instead, the automaton reads its input from a single input port. One symbol is read from this port at each step. Similarly, the output goes to a single output port, one symbol at a time. In contrast to classical models, the input stream does not have to be known in advance, and can be adjusted at any time by an external agent, based on previous in- and output symbols. This allows the environment to interact with the automaton.

We model the evolutionary aspects by considering sequences of automata. Each automaton in the sequence represents the next evolutionary phase of the system. The way in which this sequence develops need not be described recursively in general. When a transition occurs from one automaton to its successor, the information that the automaton has accumulated over time must be preserved in some way. This is done by requiring that every automaton has a subset of its states in common with its immediate successor.

Definition 6.1. *An automaton is a 6-tuple* $A = (\Sigma, \Omega, Q, I, O, \delta)$*, where* Σ *and* Ω *are non-empty finite alphabets,* Q *is a set of states,* I *and* O *are subsets of* Q*, and* $\delta : Q \times \Sigma \to Q \times \Omega$ *is a (partial) transition function.* Σ *is the input alphabet and* Ω *is the output alphabet. We call* I *the set of* entry states *and* O *the set of* exit states*.*

Definition 6.2. *Let* $\mathcal{A}$ *be a sequence of automata* A_1*,* A_2*,* A_3*, ..., with* $A_i = (\Sigma, \Omega, Q_i, I_i, O_i, \delta_i)$*, such that* $O_i \subseteq I_{i+1}$ *for every* i*. We call* $\mathcal{A}$ *a* lineage of automata*, or a lineage for short.*

The elements in $Q_i - O_i$ are called *local states* (of A_i). The first automaton, A_1, has an initial state $q_{\text{in}} \in I_1$. Usually, I_1 contains only the initial state of A_1, and I_{i+1} equals O_i. See Figure 6.1 for an example.

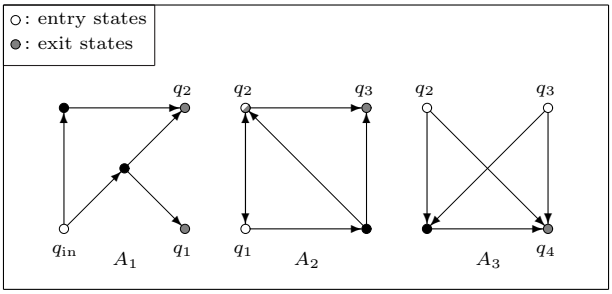

Figure 6.1. Part of a lineage $\mathcal{A}$. The set of exit states of A_1 is a subset of the set of entry states of A_2, the set of exit states of A_2 is a subset of the set of entry states of A_3.

Let $\mathcal{A}$ be a lineage. We assume that $\mathcal{A}$ has one input port and one output port, which is shared between all the automata in the sequence. Assume that we can divide time into discrete time-frames, such that in each time-frame only one symbol appears at the input port and one symbol is produced at the output port. In such a time-frame, exactly one automaton is responsible for processing the input symbol. This automaton is called *active* at this time. Initially, the first automaton in the sequence is active, and it starts processing the first symbol to appear at the input port. Whenever the currently active automaton A_i enters an exit state q, it turns the control over to A_{i+1}, which then becomes active. This is done by letting A_{i+1} start processing the next symbols appearing at the input port, beginning in state q (which is an entry state of A_{i+1} by definition). This is called *updating*, and A_i is the i-th *update* of $\mathcal{A}$.

Again, let $\mathcal{A}$ be a lineage. The entire sequence of symbols that appears at the input port is an *input* for $\mathcal{A}$. Similarly, the sequence of symbols that is generated at the output port form the *output* of $\mathcal{A}$. Note that an input is a string in Σ^∞ and an output is a string in Ω^∞. Inputs and outputs are related as follows: let Q be the union of all Q_i and let $x \in \Sigma^\infty$ be an input to a lineage $\mathcal{A}$. Using simultaneous recursion, we define a sequence of states $(q_j)_{j\geq 1}$ in Q and a sequence of integers $(m_j)_{j\geq 1}$, with m_j representing the index of the active automaton at time j, as follows:

$$\begin{aligned} q_1 &= q_{\text{in}} \ , \\ m_1 &= 1 \ , \\ \\ q_{j+1} &= \pi_1\left(\delta_{m_j}(q_j, x_j)\right) \ , \\ m_{j+1} &= \begin{cases} m_j + 1 & \text{if } q_{j+1} \in O_{m_j} \\ m_j & \text{otherwise} \end{cases} \ . \end{aligned} \tag{6.1}$$

Note that q_{j+1} and m_{j+1} depend on $x_{[1:j]}$. Therefore, we also write $q_{j+1}(x_{[1:j]})$ and $m_{j+1}(x_{[1:j]})$ to emphasize the dependence. If q_j is defined for every $j \leq 1 + |x|$,

then we say that x is a *valid input* to $\mathcal{A}$. In this case, the *output* of $\mathcal{A}$ on x is the string $y \in \Omega^\infty$ such that $y_j = \pi_2\left(\delta_{m_j}(q_j, x_j)\right)$, for every $j \geq 1$.

Let $\mathcal{A}$ be a lineage of automata and let n and m be integers. We say that $\mathcal{A}$ needs less than m updates to process all strings of length n if $m_n(x) \leq m$ for every string x of length n.

Definition 6.3. *Let $\mathcal{A}$ be a lineage. We define the partial function $\Phi^{\mathcal{A}} : \Sigma^\infty \to \Omega^\infty$ by letting $\Phi^{\mathcal{A}}(x)$ be the output of $\mathcal{A}$ on x if x is a valid input and undefined otherwise, for every string x. We say that $\Phi^{\mathcal{A}}$ is* non-uniformly *realized by the lineage $\mathcal{A}$. In general, for a partial function $\Psi : \Sigma^\infty \to \Omega^\infty$, we say that Ψ is* non-uniformly *realizable, if there is a lineage $\mathcal{A}$ such that Ψ equals $\Phi^{\mathcal{A}}$.*

For many lineages $\mathcal{A}$, the translation $\Phi^{\mathcal{A}}$ is not realizable by a single finite-state transducer and not even for a Turing transducer (Proposition 6.14). See also Van Leeuwen and Wiedermann[30].

6.2 Constructions on Lineages

In this section, we give some methods to construct new lineages $\mathcal{B}$ out of a given lineage $\mathcal{A}$ that non-uniformly realize the same translation, i.e., such that $\Phi^{\mathcal{B}}$ equals $\Phi^{\mathcal{A}}$. In fact, we show two extreme cases: a method that postpones updates of automata to arbitrary finite times and a method that updates as often as possible, i.e., after each step.

To distinguish between states of different automata (in a lineage), we let Q^A be the set of states, I^A the set of entry states and O^A the set of exit states of an automaton A.

6.2.1 Merging Two Successive Automata in a Lineage

The first method merges two successive automata A_i and A_{i+1} into one new automaton B_i such that A_i followed by A_{i+1} translates input segments in the same way as B_i. To obtain a new lineage $\mathcal{B}$ that non-uniformly realizes the same translation as $\mathcal{A}$, we let $B_j = A_j$ for all $j < i$, and we let $B_j = A_{j+1}$ for all $j > i$.

Construction 6.1. Let Q^{B_i} be the disjoint union of Q^{A_i} and $Q^{A_{i+1}}$, that is,

$$\begin{aligned} Q^{B_i} = \{\ (q,i) \quad &|\quad q \in Q^{A_i} \quad \} \quad \cup \\ \{\ (q,i+1) \quad &|\quad q \in Q^{A_{i+1}} \ \}\ . \end{aligned} \tag{6.2}$$

Roughly speaking, a state (q,i) corresponds to a state q in A_i, while a state $(q,i+1)$ corresponds to a state q in A_{i+1}. Note that each exit state q of A_i has two copies in Q^{B_i}, namely (q,i) and $(q,i+1)$. If q is not an exit state of A_{i+1}, then both copies can be local states, but if q is an exit state of A_{i+1}, then one (and only one) of the copies is an exit state. In this case we let (q,i) be the exit state. Thus we define

$$\begin{aligned} I^{B_i} &= \{\ (q,i) \quad &|\quad q \in I^{A_i} \quad \}\ , \\ O^{B_i} &= \{\ (q,i+1) \quad &|\quad q \in O^{A_{i+1}} - O^{A_i} \quad \} \quad \cup \\ &\quad \{\ (q,i) \quad &|\quad q \in O^{A_{i+1}} \cap O^{A_i} \quad \}\ . \end{aligned} \tag{6.3}$$

Let δ_i and δ_{i+1} be the transition functions of A_i and A_{i+1}, respectively. We define the transition function γ_i of B_i as follows:
For $q \in Q^{A_i}$ and $a \in \Sigma$, the transition $\gamma_i((q,i),a)$ is defined by the following cases:

- If $\delta_i(q,a) = (r,b)$ and $r \notin O^{A_i}$, then $\gamma_i((q,i),a) = ((r,i),b)$,
- if $\delta_i(q,a) = (r,b)$ and $r \in O^{A_i}$, then $\gamma_i((q,i),a) = ((r,i+1),b)$ and
- if $\delta_i(q,a)$ is undefined, then so is $\gamma_i((q,i),a)$.

For $q \in Q^{A_{i+1}}$ and $a \in \Sigma$, the transition $\gamma_i((q,i+1),a)$ is defined by these cases:

- If $\delta_{i+1}(q,a) = (r,b)$ and $r \notin O^{A_i} \cap O^{A_{i+1}}$, then $\gamma_i((q,i+1),a) = ((r,i+1),b)$,
- if $\delta_{i+1}(q,a) = (r,b)$ and $r \in O^{A_i} \cap O^{A_{i+1}}$, then $\gamma_i((q,i+1),a) = ((r,i),b)$,
- if $\delta_{i+1}(q,a)$ is undefined, then so is $\gamma_i((q,i+1),a)$.

Note that an exit state (r,i) with $r \in O^{A_i} \cap O^{A_{i+1}}$ cannot be entered from a state (q,i). To make sure that the exit states of B_i are entry states of B_{i+1}, we should relabel every state q of B_i as $(q,i+1)$ (unless q is an exit state of both A_i and A_{i+1}, in which case (q,i) is the correct label). The transition function has to be adjusted accordingly. Similarly, every state q of B_{i-1} should be relabeled (q,i). A similar relabeling has to occur for every automaton B_j, with $j \neq i$. See Figure 6.2 for an example.

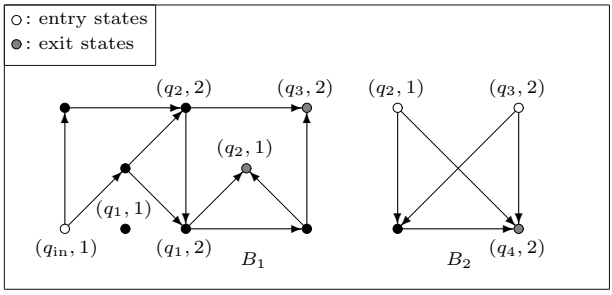

Figure 6.2. In the lineage $\mathcal{A}$ from Figure 6.1, the automata A_1 and A_2 are replaced by the automaton B_1.

Proposition 6.4. *Lineage $\mathcal{B}$ from Construction 6.1 non-uniformly realizes the same translation as $\mathcal{A}$.*

Proof. Since the states of B_j have been relabeled for $j < i$, the automaton B_i starts in an entry state (q,i) iff A_i starts in q.

To see that B_i translates input segments in the same way as A_i and A_{i+1} combined, consider a string x. Suppose A_i starts in a state q_1 and processes a part of x, until it enters an exit state q_3, say after n_i symbols of x. At this point, A_{i+1} processes the remainder of x, starting in q_3. Let q_2 be the state that A_i was in before entering q_3. Now observe the action of B_i on x. It starts in (q_1,i), and processes x in exactly the same way as A_i for $n_i - 1$ steps, ending up in (q_2,i). The next transition goes to $(q_3,i+1)$.

Suppose A_{i+1} enters an exit state q_5 after processing another part of x, say of length n_{i+1}. Let q_4 be the state A_{i+1} was in just before entering q_5. From $(q_3, i+1)$, the automaton B_i ends up in $(q_4, i+1)$ after $n_{i+1} - 1$ steps. If q_5 is also an exit state of A_i, then the next transition goes to (q_5, i), which is an exit state of B_i by definition. Otherwise, the next transition goes to $(q_5, i+1)$, which is also an exit state. Either way, B_i enters the exit state corresponding to q_5. Since the states of B_j have been relabeled for $j > i$, the rest of x is processed correctly.

If either A_i or A_{i+1} does not enter an exit state, then the transitions that occur will be mimicked by B_i (with i or $i+1$ resp. appended to the state). It follows that $\mathcal{B}$ non-uniformly realizes the same translation as $\mathcal{A}$.

□

Construction 6.1 can be applied repeatedly to merge any fixed number of consecutive automata. The intuitive notion behind lineages is that they model the evolution of a system. Applying Construction 6.1 to a lineage can be thought of as "making bigger jumps in the evolution of the system".

6.2.2 Updating the Lineage at Each Step

The next method turns a lineage $\mathcal{A}$ into a lineage $\mathcal{B}$ that non-uniformly realizes the same translation, in such a way that each automaton only processes one input symbol, i.e., after every single step the active automaton is updated to the next one.

Construction 6.2. First, we let the set of states for the lineage $\mathcal{B}$ be

$$\{ \ (q, i) \quad | \quad q \in A_i \ \} \ . \tag{6.4}$$

Now, we recursively construct the automaton B_n. Let q_{in} be the initial state of A_1. Then the initial state of B_1 is the state $(q_{\text{in}}, 1)$, and $I^{B_1} = \{(q_{\text{in}}, 1)\}$. Suppose the set of entry states I^{B_n} of B_n has been constructed. Then the set of output states O^{B_n} consists of all the states that are reachable from a state in I^{B_n} in one step, and we let $Q^{B_n} = I^{B_n} \cup O^{B_n}$. Let's make this more formal. Let (q, i) be a state in I^{B_n}, let $a \in \Sigma$, and let δ_i be the transition function of A_i. Suppose $\delta_i(q, a)$ is defined. Then there is a $b \in \Sigma$ and a state r such that $\delta_i(q, a) = (r, b)$. If r is an exit state of A_i, then we let $i' = i+1$, otherwise $i' = i$. We add the state (r, i') to O^{B_n}, and define $\gamma_n((q, i), a) = ((r, i'), b)$, where γ_n is the transition function of B_n. We do this for each state (q, i) in I^{B_n} and every $a \in \Sigma$. Once O^{B_n} is constructed, we construct the set of entry states of B_{n+1} by defining $I^{B_{n+1}} = O^{B_n}$.

In each automaton, all transitions go from an entry state to an exit state. This means that, after reading one input symbol, the automaton is updated.

Proposition 6.5. *Lineage $\mathcal{B}$ from Construction 6.2 non-uniformly realizes the same translation as $\mathcal{A}$.*

Proof. Let x be an input string. It is left to the reader to prove, using induction, that prior to reading the n-th input symbol, A_i is active in state q iff B_n is active in state (q, i). By inspecting the transition functions, we see that both automata will output the same symbol when they process x_n. So $\mathcal{B}$ non-uniformly realizes the same translation as $\mathcal{A}$.

□

Applying Construction 6.2 to a lineage can be thought of as "taking smaller steps in the evolution of the system".

6.2.3 Reducing the Number of States

The next method works only on lineages that update after every input symbol. From section 6.2.2, we conclude that every lineage is equivalent to one of this type and that in such a lineage, a state is either an entry state, or an exit state, or both. This means that the total number of states of the n-th update is at least $\max\{|I^{A_n}|, |O^{A_n}|\}$. We will alter the lineage such that the total number of states of B_n equals this maximum.

Construction 6.3. Let Q be an infinite set of states. Recursively construct injective functions $f_n : I^{A_n} \to Q$ and $g_n : O^{A_n} \to Q$ such that $f_{n+1}(q) = g_n(q)$ for every state in O^{A_n}, and $|f_n(I^{A_n}) \cup g_n(O^{A_n})| = \max\{|I^{A_n}|, |O^{A_n}|\}$. The actual construction of f_n and g_n is left to the reader. Construct a lineage $\mathcal{B}$ such that the set of entry states of B_n is the set $f_n(I^{A_n})$, and the set of exit states is $g_n(O^{A_n})$. Let $q_{\text{in}} \in I^{A_1}$ be the initial state of A_1. Then $f_1(q_{\text{in}})$ is the initial state of B_1.

Let q be a state in I^{A_n} and $a \in \Sigma$, and let δ_n be the transition function of A_n. If $\delta_n(q, a)$ is defined, then there is an exit state r and a $b \in \Sigma$ such that $\delta_n(q, a) = (r, b)$. Now define $\gamma_n(f_n(q), a) = (g_n(r), b)$, where γ_n is the transition function of B_n. Since f_n and g_n are injective, this definition is unambiguous.

Proposition 6.6. *Lineage $\mathcal{B}$ from Construction 6.3 non-uniformly realizes the same translation as $\mathcal{A}$.*

Proof. Let x be an input string. It is left to the reader to prove that, prior to reading the n-th input symbol, A_n is in state q iff B_n is in state $f_n(q)$. Just as in the previous method, we can conclude from this fact that $\mathcal{B}$ non-uniformly realizes the same translation as $\mathcal{A}$.

□

We summarize the last two constructions in the following result:

Proposition 6.7. *If a translation is non-uniformly realizable, then it can be non-uniformly realized by a lineage $\mathcal{B}$ with the property that its automata B_i update after every step and have precisely* $\max\{|I^{B_i}|, |O^{B_i}|\}$ *states each.*

6.3 Properties of Non-uniformly Realizable Translations

In this section, we show the basic properties of the class of translations that are non-uniformly realized by lineages. First, we give a useful characterization of non-uniformly realizable translations in terms of their domains. Next, we show that the class is uncountable and contains non-recursive translations. Then, we show that the class is closed under composition and inversion.

6.3.1 A Characterization of Non-uniformly Realizable Translations

It is possible to characterize the translations that are non-uniformly realized by a lineage without actually constructing lineages, by specializing the theory of continuous mappings (see Staiger[39]). This very useful characterization depends on the domain on which the translation is defined and the relation it specifies between input and output pairs.

Theorem 6.8. *A translation Φ with domain D can be non-uniformly realized by a lineage $\mathcal{A}$ iff*

- $|\Phi(x)| = |x|$ *for all* $x \in D$,
- *if* u *is a prefix of* $x \in D$, *then* $u \in D$ *and* $\Phi(u)$ *is a prefix of* $\Phi(x)$,
- D *is closed.*

We prove the Theorem with the following Propositions.

Proposition 6.9. *Let Φ be a non-uniformly realizable translation with domain D. Then $|\Phi(x)| = |x|$ for all $x \in D$.*

Proof. This follows directly from (6.1) and Definition 6.3.

□

Proposition 6.10. *Let Φ be a non-uniformly realizable translation with domain D. If u is a prefix of $x \in D$, then $u \in D$ and $\Phi(u)$ is a prefix of $\Phi(x)$.*

Proof. Let Φ be non-uniformly realized by the lineage $\mathcal{A}$. Let $x \in D$. Then x is a valid input to $\mathcal{A}$. If u is a prefix of x, then u is also a valid input. The result then follows from (6.1) and Definition 6.3.

□

Proposition 6.11. *Let Φ be a non-uniformly realizable translation with domain D. Then D is closed.*

Proof. Let $\mathcal{A}$ be a lineage that non-uniformly realizes Φ. Let $x \notin D$ be a string and consider a run of $\mathcal{A}$ on x. Because x is not in the domain of Φ, there is a finite prefix u of x, that is not processed by $\mathcal{A}$.

Let y be a string in $\mathbb{B}(u)$ and consider a run of $\mathcal{A}$ on y. We conclude that y cannot be processed by $\mathcal{A}$. It follows that $\mathbb{B}(u)$ does not intersect D, which implies that D is a closed set.

□

The previous Propositions showed that a non-uniformly realizable translation fulfills the conditions of Theorem 6.8. The next Proposition constructs a lineage for a translation that satisfies the conditions. For a domain D, denote the set $D \cap \Sigma^n$ by D_n.

Proposition 6.12. *Let Φ be a translation with domain D. Suppose that*

- $|\Phi(x)| = |x|$ *for all* $x \in D$,
- *if* u *is a prefix of* $x \in D$, *then* $u \in D$ *and* $\Phi(u)$ *is a prefix of* $\Phi(x)$,
- D *is closed.*

Then Φ can be non-uniformly realized by a lineage $\mathcal{A}$ that updates after every step, such that A_n has $|D_{n-1}|$ entry states and $|D_n|$ exit states.

Proof. Define the set of states of A_n for $n \geq 1$ by

$$\begin{aligned} I_n &= \{\ [\![u]\!] \quad | \quad u \in D_{n-1}\ \}\ , \\ O_n &= \{\ [\![u]\!] \quad | \quad u \in D_n\ \}\ , \end{aligned} \tag{6.5}$$

and let $Q_n = I_n \cup O_n$. The initial state of A_1 is $[\![\epsilon]\!]$. The transition function δ_n is defined by

$$\delta_n([\![u]\!], a) = \begin{cases} ([\![ua]\!], (\Phi(ua))_n) & \text{if } ua \in D_n \\ \text{undefined} & \text{otherwise} \end{cases}\ . \tag{6.6}$$

The transition function is well-defined, since D_n is a subset of D and $|\Phi(x)| = |x|$ for all x. Using induction and the fact that $\Phi(u)$ is a prefix of $\Phi(ua)$ for $u, ua \in D$, one can show that $\mathcal{A}$ produces $\Phi(x)$ on input $x \in D$.

Suppose on the other hand that $x \notin D$. Since D is closed, there is a basis set $\mathbb{B}(u)$ that contains x, which does not intersect D. It follows that $u \notin D$. Since the transition functions are not defined on u, we see that u (and therefore x) is not a valid input to $\mathcal{A}$. Hence $\mathcal{A}$ non-uniformly realizes Φ.

□

For an example of a translation that cannot be non-uniformly realized by a lineage, consider the translation from Example 6.13.

Example 6.13. Let $\Sigma = \{0, 1\}$, and $\Omega = \{a, b, c, d\}$. Define the help-function ψ by:

$$\begin{aligned} \psi(00) &= aa\ , \\ \psi(01) &= bb\ , \\ \psi(10) &= cc\ , \\ \psi(11) &= dd\ . \end{aligned} \tag{6.7}$$

Now, we define the translation $\Phi : \Sigma^\infty \to \Omega^\infty$ by

$$\Phi(x) = \psi(x_1x_2)\psi(x_3x_4)\dots\ , \tag{6.8}$$

for strings x of infinite length. If x is a finite string, then $\Phi(x)$ is undefined.

Since Φ is not defined on finite prefixes, it is not non-uniformly realizable. Furthermore, it can not be embedded into a non-uniformly realizable translation. To see this, suppose that there is a lineage $\mathcal{A}$ which behaves like Φ on infinite inputs. Consider the possibilities when A_1 is given the input 0. It must produce an output. If a is produced, then $\mathcal{A}$ fails to correctly process strings that start with 01, but if it doesn't produce a, strings which start with 00 are not properly processed. We conclude that Φ cannot be embedded in a non-uniformly realizable translation.

6.3.2 The Number of Non-uniformly Realizable Translations

Observe that any translation that is realized by a finite-state transducer can also be non-uniformly realized by a lineage (just take infinitely many copies of the transducer that realizes the translation). On the other hand, the class of non-uniformly

realizable translations contains uncountably many translations that cannot be realized by a finite-state transducer. We deduce that the class of non-uniformly realizable translations is uncountable, whereas the class of translations that are realized by finite-state transducers is countable.

Proposition 6.14. *Let Ω be an alphabet with at least two elements. There are uncountably many non-uniformly realizable translations from Σ (any Σ) to Ω that cannot be realized by a finite-state transducer.*

Proof. Pick two elements of Ω, call them 0 and 1. Let N be a set of positive integers that is not recursively enumerable and let $(n_i)_{i\geq 1}$ be an enumeration of N. Consider the infinite string $y = 1^{n_1}01^{n_2}01^{n_3}0\ldots$. Define the translation Φ by letting $\Phi(x)$ be a prefix of y of length $|x|$, for every x in Σ^∞. Since the domain is all of Σ^∞, it fulfills the conditions of Proposition 6.12, so Φ is non-uniformly realizable.

Suppose that Φ can be realized by an automaton A. Let a be a letter in Σ and let M be a Turing machine, on input $i \in \mathbb{N}$ in unary, simulates A on input $a^\star$ until A has written i zeroes. Then M outputs the last sequence of ones in A's output. We see that M computes n_i in unary. It follows that M enumerates N, which is a contradiction. Because there are uncountably many sets that are not recursively enumerable, we have the desired result.

□

If in the proof we replace the automaton A by a Turing machine, the proof is still valid. We conclude that lineages possess super-Turing computing power.

6.3.3 Composition of Non-uniformly Realizable Translations

The general class of translations is closed under the operations of composition and inverse. A natural question that arises, is whether the class of non-uniformly realizable translations is also closed under these operations. This question is answered in this and the next subsection.

Proposition 6.15. *Let $\Phi^{\mathcal{A}} : \Sigma^\infty \to \Omega^\infty$ and $\Phi^{\mathcal{B}} : \Omega^\infty \to \Theta^\infty$ be translations non-uniformly realized by lineages $\mathcal{A}$ and $\mathcal{B}$ respectively. Then a lineage $\mathcal{C}$ exists such that $\Phi^{\mathcal{C}} = \Phi^{\mathcal{B}} \circ \Phi^{\mathcal{A}}$.*

Proof. Given lineages $\mathcal{A}$ and $\mathcal{B}$, we construct a lineage $\mathcal{C}$ by defining for every automaton C_i its set of states, its initial state and its transition function as follows. The set of states of C_i is defined by:

$$Q^{C_i} = \{\ (q,k,r,l) \quad | \quad k,l \leq i+1,\ q \in Q^{A_k} \wedge r \in Q^{B_l}\ \}\ . \tag{6.9}$$

When C_i is in state (q,k,r,l), this simulates the fact that A_k has entered state q and B_l has entered state r. So, $\mathcal{C}$ simulates the transitions of $\mathcal{A}$ in the first two components of its states and the transitions of $\mathcal{B}$, with the output of $\mathcal{A}$ as input, in the last two components. Special care must be taken with exit states. If A_k enters the exit state q, then A_{k+1} will start in state q. So, the corresponding state will be $(q, k+1, \_, \_)$. A similar thing holds for $\mathcal{B}$.

Let q_{in} be the initial state of A_1 and r_{in} the initial state of B_1. The initial state of C_1 is $(q_{\text{in}}, 1, r_{\text{in}}, 1)$. A state (q, k, r, l) is an exit state if q is an exit state of A_{k-1} (and $k > 1$) or r is an exit state of B_{l-1} (and $l > 1$). The state is an entry state if q is an entry state of A_k or r is an entry state of B_l.

Now, we define the transition function τ_i of C_i. Let (q, k, r, l) be a state in C_i, with $k, l \leq i$, and let a be a letter. Let δ_k be the transition function of A_k and γ_l the transition function of B_l. Suppose that $\delta_k(q, a) = (q', b)$ and $\gamma_l(r, b) = (r', c)$. The transition function τ_i simulates the changes in $\mathcal{A}$ and $\mathcal{B}$. In $\mathcal{A}$, the parameters q and k change to q' and k', with $k' = k + 1$ if q' is an exit state, or $k' = k$. Similarly, in $\mathcal{B}$, the parameters r and l change to r' and l'. Thus, we let $\tau_i((q, k, r, l), a) = ((q', k', r', l'), c)$. In all other cases we let τ_i be undefined.

Proving that $\mathcal{C}$ non-uniformly realizes the translation $\Phi^{\mathcal{B}} \circ \Phi^{\mathcal{A}}$ is similar to the proof of Proposition 6.5. If x is an input to $\mathcal{A}$ and $y = \Phi^{\mathcal{A}}(x)$ is an input to $\mathcal{B}$ then, just before reading the n-th input symbol, C_i is in state (q, k, r, l) iff A_k is in state q and B_l is in state r. Inspecting the transition functions, we see that in this case $\Phi^{\mathcal{B}}(\Phi^{\mathcal{A}}(x)) = \Phi^{\mathcal{C}}(x)$. If, on the other hand, either x or y is not valid for $\mathcal{A}$ or $\mathcal{B}$ respectively, then there comes a time when either $\delta_k(q, x_n)$ or $\gamma_l(r, y_n)$ is undefined. In either case, $\tau_i((q, k, r, l), x_n)$ is also undefined. Hence the domains match and the two functions are equal.

□

The set of states Q_{C_i} in the given proof can be taken much smaller. In fact, we don't need the states $(q, i + 1, \_, \_)$ unless q is an exit state of A_i. Likewise we can do without the states $(\_, \_, r, i + 1)$ if r is not an exit state of B_i.

6.3.4 The Inverse of a Non-uniformly Realizable Translation

Much of the theory presented up until now, remains valid when we ignore finite inputs and consider translations as functions from infinite strings to infinite strings only. The results in this section however, can not be carried over. Consider Example 6.16.

Example 6.16. Let $\Sigma = \{a, b, c, d\}$, and $\Omega = \{0, 1\}$. Define the help-function ψ by:

$$\begin{aligned} \psi(aa) &= 00 \ , \\ \psi(bb) &= 01 \ , \\ \psi(cc) &= 10 \ , \\ \psi(dd) &= 11 \ . \end{aligned} \tag{6.10}$$

The output of ψ is undefined for all other inputs. Now, we define the translation $\Phi : \Sigma^\omega \to \Omega^\omega$ by

$$\Phi(x) = \psi(x_1 x_2)\psi(x_3 x_4)\psi(x_5 x_6)\dots \ . \tag{6.11}$$

This translation is injective. Although it does not fit the conditions of Proposition 6.12, it can be embedded into a non-uniformly realizable translation. Thus, when one ignores finite inputs, one might say that Φ is an injective non-uniformly realizable translation. The inverse of Φ is the translation from Example 6.13. We see that although Φ can be embedded into a non-uniformly realizable translation, its inverse cannot.

Note that the non-uniformly realizable translation used for the embedding is not an injection, since $\Phi(a) = \Phi(b) = 0$. The following Proposition shows that for injective non-uniformly realizable translations (where finite inputs *are* taken into account), their inverses are non-uniformly realizable.

Proposition 6.17. *Let Φ be an injective non-uniformly realizable translation with domain D. Then the translation Φ^{-1} with domain $\Phi(D)$ can be non-uniformly realized.*

Proof. Let $\mathcal{A}$ be a lineage that non-uniformly realizes $\mathcal{A}$. We will transform $\mathcal{A}$ into a lineage $\mathcal{B}$ that non-uniformly realizes Φ^{-1}. Consider the automaton A_k, with transition function δ_k. First, we remove all states that cannot be reached from the initial state.

Suppose $\delta_k(q, a) = (r, b)$ and $\delta_k(q, a') = (r', b)$. Let u be a string such that A_k enters q after processing u, with output v. Then the output belonging to ua is vb, and the output belonging to ua' is also vb. Since Φ is injective, it follows that $a = a'$ (and $r = r'$). Therefore the function γ_k, defined by

$$\gamma_k(q, b) = \begin{cases} (r, a) & \text{if } \delta_k(q, a) = (r, b) \\ \text{undefined} & \text{otherwise} \end{cases} , \tag{6.12}$$

is well-defined. The automaton B_k is defined by taking A_k, with δ_k replaced by γ_k.

By inspecting the transition functions, we see that for strings $y = \Phi(x)$, the lineage $\mathcal{B}$ gives x as output.

Let y be a string not in $\Phi(D)$. Then there is a largest prefix v of y, such that $\mathcal{A}$ gives v as output, on an input u. Suppose $\mathcal{A}$ enters a state q after processing u. There is no transition from q that gives $y_{|v|+1}$ as output. It follows that there is no transition from q with input $y_{|v|+1}$ in $\mathcal{B}$. Then, $\mathcal{B}$ enters q after processing v, but it cannot process $y_{|v|+1}$, thus y is not a valid input to $\mathcal{B}$. Hence the domain of $\mathcal{B}$ is $\Phi(D)$. We conclude that $\mathcal{B}$ non-uniformly realizes Φ^{-1}.

□

Most translations can be made injective by just restricting their domain. If a translation is a bijection, then it follows that the input and output alphabets need to be of the same size. Propositions 6.15 and 6.17 show that the class of non-uniformly realizable translations is closed under composition and inversion.

6.4 Conclusions

In this Chapter, lineages of automata were introduced. This model was introduced in a slightly different form by Van Leeuwen and Wiedermann[30]. This model is different from other sequence-based models such as the ones in Chapter 4 by the fact that the automata in a lineage have a method of passing information to their immediate successor. By passing along enough information, the computation of an automaton can then be continued by its successor. Another way of looking at it is that the automaton changes or evolves into its immediate successor.

Since we are dealing with automata, all the information has to be stored in the finite control. Information is passed along by using sets of states that are shared between two neighboring automata in a sequence. The entry states of an automaton in lineage can be compared to initial states of a classical finite automaton. The exit states basically force an automaton to halt. The next automaton can then resume the computation. Thus, the exit states can be seen as another halting criterion.

We gave several constructions to produce new lineages out of given ones. It is important to note that all these constructions were recursive procedures. The translations that are non-uniformly realized by lineages can be characterized in terms of their domains and the relations between inputs and outputs. This characterization does not use the lineages at all, so one can describe the translations without constructing a lineage. It proves much easier to test if a translation is non-uniformly realizable by using this characterization then it is to define a lineage that non-uniformly realizes the translation.

CHAPTER 7

The Complexity of Lineages

In Chapter 6, we introduced lineages of automata as a model for evolving interactive systems and studied the basic properties of lineages. We have looked at the translations that are non-uniformly realizable by lineages. We have seen that there are non-uniformly realizable translations that cannot be realized by Turing machines, and translations that cannot be non-uniformly realized at all.

In this Chapter, we will make a finer separation between translations, based on the processing power of the lineages that non-uniformly realize them. For this, we need to define a measure of processing power. In the case of ordinary finite automata, the number of states is a good measure. An automaton with more states is able to distinguish among a greater number of different situations. It can apply different actions to each situation it can recognize, thus adding more diversity to a computation. Likewise, the number of states of every automaton in a lineage serves as a good measure of the computational power of the lineage. We measure the "speed of growth" (i.e., "growth complexity") of a lineage by a function that relates the index of each automaton in the lineage to its size. That is, the complexity of a lineage is a function g such that the n-th automaton in the sequence has $g(n)$ states. Using this measure, we can divide the translations computed by evolving systems into classes based on the complexity of the lineages that non-uniformly realize them. We will show in this Chapter that this division is non-trivial, i.e., for every positive, non-decreasing function g, there is a translation that can be realized by a lineage with complexity g, but not by any lineage with a lower complexity.

The structure of the Chapter is as follows. First, we define the complexity measure for lineages and the complexity classes that are based on it. We give a fundamental result about the highest possible complexity, showing that no extra computational power is gained beyond exponentially sized advices. Next, we consider some closure properties of complexity classes. Then, we define a translation of a given complexity (under suitable restrictions), and show that it cannot be non-

uniformly realized more efficiently. We use this translation to show that adding just one extra state to an automaton in a lineage can give more processing power to the lineage. Finally, we compare lineages to ITM/A's. In particular, we examine the complexity of an ITM/A simulating a lineage and of a lineage simulating an ITM/A.

7.1 A Complexity Measure for Lineages

For a lineage, which is a sequence of automata, the number of states of each of the constituent automata contributes to the computing power of the lineage. Therefore, we use a function to describe the complexity of a lineage.

Definition 7.1. *The* complexity *of a lineage $\mathcal{A}$ is a function g such that for every n, the n-th automaton of $\mathcal{A}$ has $g(n)$ states. We say that a translation Φ is of complexity g if there is a lineage $\mathcal{A}$ of complexity g that non-uniformly realizes Φ. We define the complexity class SIZE(g) as the class of non-uniformly realizable translations of complexity g.*

Remark 7.2. The automata of a lineage with $\leq g(n)$ states per automaton can be augmented with unreachable dummy states, to obtain a lineage with $g(n)$ states per automaton. Thus, if a translation is of complexity f and $f(n) \leq g(n)$ for every n, then the translation is of complexity g.

An important subclass of lineages are the lineages that update after every step. Recall that any lineage can be turned into a lineage that updates after every step by applying Construction 6.2.

Proposition 7.3. *Let $\mathcal{A}$ be a lineage over an alphabet of size c. After applying Construction 6.2 and Construction 6.3 to $\mathcal{A}$, the resulting lineage is of complexity g for a function g such that $g(1) \leq c$ and $g(n+1) \leq c \cdot g(n)$ for every n.*

Proof. By applying Construction 6.2, all unreachable states are removed. Suppose A_n has $g(n)$ states. Then A_{n+1} has at most $g(n)$ entry states, so at most $c \cdot g(n)$ states can be reached in one step. Thus, A_{n+1} has at most $c \cdot g(n)$ exit states. By applying Construction 6.3, it follows that A_{n+1} has at most $c \cdot g(n)$ states. Hence, $g(n+1) \leq c \cdot g(n)$. Similarly, A_1 has one entry state, namely the initial state. Thus, A_1 has at most $c \cdot 1$ states, so $g(1) \leq c$.

□

Corollary 7.4. *Let Φ be a non-uniformly realizable translation over an alphabet of size c. Then Φ can be non-uniformly realized by a lineage of size at most c^n.*

7.1.1 Properties of Complexity Classes

In Chapter 6, some closure properties of the class of translations were examined. Here, we will consider the same properties of the complexity classes.

First, we look at composition. Let $\mathcal{C}$ be a lineage that non-uniformly realizes the composition of two non-uniformly realizable translations, one of complexity g,

and one of complexity f, respectively. From the proof of Proposition 6.15, we can conclude that the complexity of $\mathcal{C}$ is given by the function

$$g'(n) = \sum_{i,j \leq n+1} g(i) \cdot f(j) \ . \tag{7.1}$$

Although not every complexity class is closed under composition, many interesting classes are. As an example, we give the following Corollary.

Corollary 7.5. *The composition of two non-uniformly realizable translations with polynomially bounded complexity has polynomially bounded complexity too.*

Next, we take a look at inversion. The lineage that is constructed in the proof of Proposition 6.17 has the same complexity as the lineage that we started with (after removing the redundant states). It follows that any two non-uniformly realizable translations that are each others inverse belong to the same complexity class.

Corollary 7.6. *If a non-uniformly realizable translation has an inverse, then both translations have the same complexity.*

Also in Chapter 6, some constructions on lineages were given. These constructions alter the way a lineage behaves with regards to updating. We will examine the effect of the constructions on the complexity of the lineages.

The merging of the i-th and $(i+1)$-st automaton in a lineage (Construction 6.1) will usually increase its complexity. Given a complexity of g, the new complexity becomes

$$g'(n) = \begin{cases} g(n) & \text{if } n < i \\ g(i) + g(i+1) & \text{if } n = i \\ g(n+1) & \text{if } n > i \end{cases} \ . \tag{7.2}$$

Many interesting classes are closed under a finite number of merges.

Corollary 7.7. *Merging a finite number of automata in a lineage of polynomially bounded complexity yields a lineage of polynomially bounded complexity.*

Altering a lineage such that it updates after every step (Construction 6.2) will also usually increase its complexity. Given a lineage of complexity g, the new lineage will have a complexity that is bounded by

$$g'(n) \leq \sum_{i=1}^{n} g(i) \ . \tag{7.3}$$

This is also not too bad, as illustrated by the following Corollary.

Corollary 7.8. *Altering a lineage of polynomial complexity so that it updates after every step results in a lineage of polynomial complexity.*

As a consequence of these results, for many complexity classes the update behavior of a lineage does not really influence the growth order of the complexity of the translation that is non-uniformly realized by it.

7.2 Complexity Classes

We have expressed the complexity of an evolving interactive system by a positive integer-valued function. However, not every positive integer-valued function corresponds naturally to a complexity class, e.g. the super-exponential functions do not (Corollary 7.4). The *growth-rate* of an integer-valued function g is defined by the function that maps n to $g(n+1)/g(n)$. In this section, we will show that non-decreasing functions with a bounded growth-rate correspond to non-empty complexity classes. In particular, if a non-decreasing function g has a bounded growth-rate, we can define a translation of complexity g. Furthermore, we will show that this translation does not have a lower complexity. We will use these functions to prove a more general hierarchy result for complexity classes, where we no longer need the bounded growth-rate.

For any given function $g : \mathbb{N} \to \mathbb{N}$, we construct a new function g_c with growth-rate bounded by c, such that g_c is bounded by the original function g. The function $g_c(n)$ is defined by

$$\begin{array}{ll} g_c(1) & = \min\{\ g(1) \quad\ , c \quad\ \ \} \ , \\ g_c(n+1) & = \min\{\ g(n+1)\ , c \cdot g_c(n)\ \} \ . \end{array} \tag{7.4}$$

It follows that for every n,

- $g_c(n) \leq g(n)$,
- $g_c(n) \leq g_c(n+1)$, and
- $g_c(n+1) \leq c \cdot g_c(n)$.

Thus, the function g_c is bounded by g and the growth-rate of g_c is bounded by c. Our main aim in this section is to show that there is a translation $\Phi^{g,c}$ of complexity g_c that cannot be non-uniformly realized by any lineage of complexity f, for any function f such that $f(n) < g_c(n)$ for at least one integer n.

First, we establish the domain of $\Phi^{g,c}$. Let Σ be an alphabet of size c. For every n, we choose $g_c(n)$ strings of length n, such that they are prefixes of the $g_c(n+1)$ strings of length $n+1$. The details are given in Construction 7.1.

Construction 7.1. Label the letters from Σ as a_1 through a_c, and let D_n be the chosen subset of size $g_c(n)$ of Σ^n. We proceed recursively.

$$D_1 = \{\ a_i \quad | \quad i \leq g_c(1)\ \} \ . \tag{7.5}$$

Assume D_n is chosen. Using integer division, we write $g_c(n+1) = l \cdot g_c(n) + m$, for unique integers l and m, with $0 \leq m < g_c(n)$. It follows that $1 \leq l \leq c$ (if $l = c$, then $m = 0$). Let $u_1, \ldots, u_m$ be m different strings in D_n. Now, take

$$D_{n+1} = \{\ u a_i \quad | \quad u \in D_n \ \wedge\ i \leq l\ \} \ \cup\ \{\ u_j a_{l+1} \quad | \quad j \leq m\ \} \ . \tag{7.6}$$

It is left to the reader to verify that D_{n+1} contains $g_c(n+1)$ strings of length $n+1$.

Note that $ua_1 \in D_{n+1}$ for every n and every $u \in D_n$. If u is a string in D_m, then $u_{[1:n]} \in D_n$ for every $n \leq m$. Similarly, we define the set D_ω of infinite strings x such that $x \in D_\omega$ iff $x_{[1:n]} \in D_n$ for every n. The translation $\Phi^{g,c}$ will be defined on the domain

$$D = D_\omega \cup \bigcup_{n \geq 1} D_n \ . \tag{7.7}$$

Construction 7.2. Consider the family of functions $f_{k,l,m} : \Sigma^{\geq m} \to \Sigma^{1+l}$, defined by

$$f_{k,l,m}(x) = \begin{cases} x_k^{1+l} & \text{if } 0 < k \leq m \\ x_1^{1+l} & \text{otherwise} \end{cases} , \tag{7.8}$$

for $k, l, m \in \mathbb{N}$. Let $\psi : \mathbb{N} \to \mathbb{N} \times \mathbb{N}$ be a surjective function that attains each value infinitely often. For $x \in D$, consider the string

$$y = f_{\psi(1),1}(x) f_{\psi(2),2}(x) \dots f_{\psi(|x|),|x|}(x) \ . \tag{7.9}$$

The length of y is at least $|x|$. Now, define the translation Φ by letting $\Phi(x)$ be the prefix of length $|x|$ of y. Finally, we define the translation $\Phi^{g,c}$ by

$$\Phi^{g,c}(x) = \begin{cases} \Phi(x) & \text{if } x \in D \\ \text{undefined} & \text{otherwise} \end{cases} . \tag{7.10}$$

Observe that $|\Phi(x)| = |x|$ for every $x \in D$. Furthermore, if u is a prefix of $x \in D$, then $u \in D$ and $\Phi(u)$ is a prefix of $\Phi(x)$. The following Lemma shows that D is closed, which means that $\Phi^{g,c}$ can be non-uniformly realized by a lineage of automata.

Lemma 7.9. *D is a closed set.*

Proof. Suppose $x \notin D$. Then there is a finite prefix u of x such that $u \notin D$. Let y be a string in $\mathbb{B}(u)$. Since u is a prefix of y, it follows that $\Phi(y)$ is undefined, so $y \notin D$. We conclude that $\mathbb{B}(u)$ does not intersect D, hence D is closed.

□

Combining Lemma 7.9 and Proposition 6.12, we conclude that $\Phi^{g,c}$ can be non-uniformly realized. Next, we will examine the complexity of $\Phi^{g,c}$. Proposition 7.10 shows that $\Phi^{g,c}$ is of complexity g_c, while Proposition 7.12 tells us that any lineage with a complexity less than g_c cannot non-uniformly realize $\Phi^{g,c}$.

Proposition 7.10. *The translation $\Phi^{g,c}$ can be non-uniformly realized by a lineage $\mathcal{A}$ that updates at every step, such that A_n has $g_c(n)$ states.*

Proof. Let $\mathcal{B}$ be the lineage from Proposition 6.12. Since $D \cap \Sigma^n$ equals D_n, we see that B_n has $g_c(n-1)$ entry states and $g_c(n)$ exit states. Remember that $g_c(n-1) \leq g_c(n)$. Using Construction 6.3, we can transform $\mathcal{B}$ into a lineage $\mathcal{A}$ with $g_c(n)$ states, that updates at every step.

□

For the proof of Proposition 7.12, we need the following Lemma.

Lemma 7.11. *Let k, l and $n \geq 1$ be integers such that $k, l \geq n$. Let x and y be infinite strings such that $x_n \neq y_n$. Then there is an $i > 0$ such that*

$$(\Phi(x))_{k+i} \neq (\Phi(y))_{l+i} \ . \tag{7.11}$$

Proof. Assume $k \geq l$. Let $t = k - l$. Choose an integer $m > l$ such that $\psi(m) = (n, t)$. Then $f_{\psi(m),m}(x) = f_{n,t,m}(x) = x_n^{1+t}$, since $n \leq m$. It follows that $\Phi(x)$ contains a string x_n^{1+t}, starting at an index $i_m \geq m$. Similarly, $\Phi(y)$ contains a string y_n^{1+t}, starting at the same index, see Figure 7.1. But then

$$(\Phi(x))_{i_m+t} \neq (\Phi(y))_{i_m} \quad , \tag{7.12}$$

since $x_n \neq y_n$. Now $i_m \geq m > l$, so $i_m = l + i$ for a certain $i > 0$, and $i_m + t = l + i + (k - l) = k + i$. Therefore

$$(\Phi(x))_{k+i} \neq (\Phi(y))_{l+i} \quad . \tag{7.13}$$

□

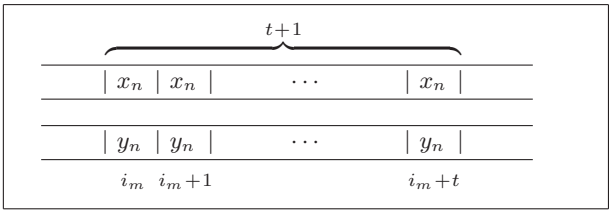

Figure 7.1. Part of the outputs of Φ, on the inputs x and y. Starting in position i_m, the outputs contain a sequence of x_n's and y_n's respectively, of size $t + 1$ each. As a result, $(\Phi(x))_{i_m+t} = x_n \neq y_n = (\Phi(y))_{i_m}$.

Let u' and v' be two different strings in D_n. Let u and v be strings in D such that u' is a prefix of u and v' is a prefix of v. Finally, let $x = (a_1)^\omega$. It follows that ux and vx are elements of D. Since $u' \neq v'$, it follows that there is an $n' \leq n$, such that $(ux)_{n'} \neq (vx)_{n'}$. By Lemma 7.11, there is an $i > 0$ such that

$$(\Phi^{g,c}(ux))_{|u|+i} \neq (\Phi^{g,c}(vx))_{|v|+i} \quad . \tag{7.14}$$

See Figure 7.2 for a visual explanation. In fact, $x = (a_1)^i$ is long enough.

Proposition 7.12. *Let $\mathcal{A}$ be a lineage that non-uniformly realizes $\Phi^{g,c}$. Suppose $\mathcal{A}$ needs less than m updates to process all strings of length n. Then A_m has at least $g_c(n)$ states.*

Proof. Suppose A_m has less than $g_c(n)$ states. Then there are two different strings u' and v' in D_n, with strings u and v in D that extend u' and v' respectively, such that A_m enters the same state r after processing either u or v, see Figure 7.3.

Then there is an $i > 0$ that satisfies (7.14). Suppose $\mathcal{A}$ is in the m-th update (or the $(m + 1)$-st, if r was an exit state), in state r. Now we give $x = (a_1)^i$ as further input to $\mathcal{A}$. After i steps, $\mathcal{A}$ enters a state r' with a certain output b. These last i steps are independent of the steps that $\mathcal{A}$ took to reach r. In other words,

$$(\Phi^{g,c}(ux))_{|u|+i} = (\Phi^{g,c}(vx))_{|v|+i} = b \quad , \tag{7.15}$$

which contradicts (7.14). It follows that A_m must have at least $g_c(n)$ states.

□

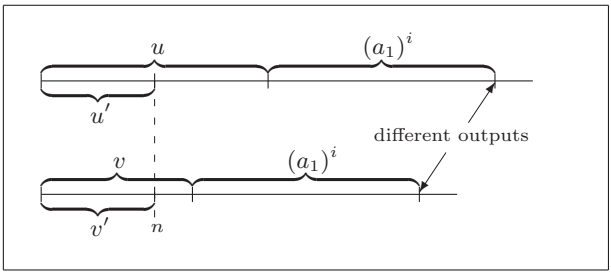

Figure 7.2. Two different finite strings u' and v' in D_n are extended to strings u and v in D respectively. The strings u and v are extended with the infinite string $x = (a_1)^\omega$. Then there is an integer $i > 0$ such that $(\Phi^{g,c}(ux))_{|u|+i} \neq (\Phi^{g,c}(vx))_{|v|+i}$.

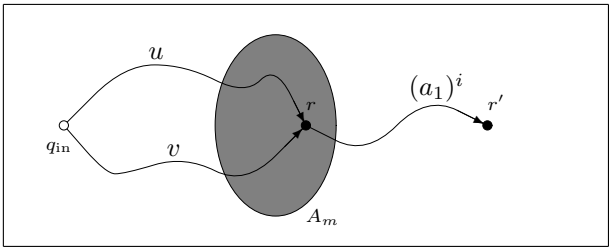

Figure 7.3. The paths of two valid input-prefixes u and v. After processing u or v, A_m enters the state r. Then the remainder of the input is processed, which equals $(a_1)^i$ in both cases. The rest of the path only depends on r and $(a_1)^i$, so after i steps, both paths enter r' and the same output symbol is generated.

Proposition 7.13. *Let $\mathcal{A}$ be a lineage that non-uniformly realizes $\Phi^{g,c}$. Then A_n has at least $g_c(n)$ states.*

Proof. Since each active automaton must read at least one symbol before $\mathcal{A}$ can update, by the time $\mathcal{A}$ is ready to update to the $(n+)$1-st automaton, at least n symbols have been read. The result now follows from Proposition 7.12.

□

We conclude that for any non-decreasing function g and any integer $c > 1$, the complexity class $\text{SIZE}(g_c)$ contains the translation $\Phi^{g,c}$. Furthermore, if f is any function such that $f(n) < g_c(n)$ for a certain n, then $\text{SIZE}(f)$ does not contain $\Phi^{g,c}$.

7.3 A Hierarchy Result for Complexity Classes

For clarity, we repeat the results of the preceding section in one Proposition.

Proposition 7.14. *Let c be a positive integer and g a positive, non-decreasing function. Let f be a function such that $f(m) < g_c(m)$ for at least a certain m. Then $SIZE(g_c) - SIZE(f)$ is non-empty.*

Proof. Combine Proposition 7.10 and Proposition 7.13.

□

When we are free to choose c, we can show that for any positive non-decreasing function g, translations exist that cannot be non-uniformly realized by any lineage that has less than $g(n)$ states in its n-th automaton, for any n. Observe that this is a stronger claim than before; we no longer require the growth-rate to be bounded.

Theorem 7.15. *Let g be a positive, non-decreasing function and let f be a function such that $f(m) < g(m)$ for at least a certain m. Then SIZE(g) − SIZE(f) is non-empty.*

Proof. Let m be an integer such that $f(m) < g(m)$. Let $c \geq g(1)$ be an integer such that $g(n+1) \leq c \cdot g(n)$ for every n smaller than m. Then $g_c(n) = g(n)$ for all $n \leq m$. It follows that $f(m) < g_c(m)$. The translation $\Phi^{g,c}$ can be non-uniformly realized by a lineage of size g_c (Proposition 7.10). Hence $\Phi^{g,c} \in \text{SIZE}(g)$.

Any lineage $\mathcal{A}$ that non-uniformly realizes $\Phi^{g,c}$ must have at least $g_c(n)$ states in its n-th automaton (Proposition 7.13). Since $f(m) < g_c(m)$, it follows that $\mathcal{A}$ is not of size f. Hence $\Phi^{g,c} \notin \text{SIZE}(f)$.

□

Corollary 7.16. *Let g and h be positive non-decreasing functions such that $f(n) \leq g(n)$ for all n. If the inequality is strict for a certain n, then SIZE(f) is a proper subset of SIZE(g).*

Proof. By the observations in Remark 7.2, any translation of complexity f is in SIZE(g). By Theorem 7.15, not every translation of complexity g is in SIZE(f).

□

This means that every extra state of a lineage can be used to gain more potential computing power. As a final Corollary, we observe that the roles of g and f can be interchanged.

Corollary 7.17. *Let g and f be positive non-decreasing functions such that $f(n) < g(n)$ for a certain n and $g(m) < f(m)$ for a certain m. Then the classes SIZE(g) and SIZE(f) are incomparable, both contain translations that do not occur in the other.*

While lineages are powerful enough to decide translations that lie well beyond the range of classical Turing machines, this power can be kept in check by imposing bounds on the sizes of the automata in the lineages. This way, the complexity theory slices out infinitely many subclasses of the translations that can be non-uniformly realized by lineages. This places the theory firmly beside the classical Turing machine based complexity theory.

7.4 Lineages and Interactive Turing Machines with Advice

Van Leeuwen and Wiedermann[30] showed that any lineage of automata can be simulated by an ITM/A, and that any ITM/A can be simulated by a lineage. In this section, we examine what the consequences of this relation are for the complexity classes.

Theorem 7.18. *Let $\Phi : \Sigma^\infty \to \Omega^\infty$ be a translation. Suppose Φ is non-uniformly realized by a lineage of automata of complexity g. Then, Φ can be realized by an ITM/A of space complexity $O(\log g)$ and advice of size $O(g \log g)$.*

Proof. Suppose Φ is non-uniformly realized by a lineage $\mathcal{A}$ of complexity g. Let $\alpha : \mathbb{N} \to \Sigma^\star$ be an advice function such that $\alpha(n)$ is the description of A_n. Since A_n has $g(n)$ states, the length of $\alpha(n)$ is $O(g(n) \log g(n))$. We construct an ITM/A M that uses the advice value $\alpha(k)$ to simulate the automaton A_k.

Suppose that after processing n input symbols, the lineage is in its k-th update, in state q. Suppose M is in an external configuration, after processing the first n non-λ symbols of the input. At this time, the advice tape contains the first n values of the advice function. In other words, the tape contains the descriptions of the first n automata of the lineage, including A_k. The head of the advice tape is positioned at the beginning of the description of q in the description of A_k. A separate update tape contains a 1 if the lineage performs an update before processing the next input symbol, and is empty otherwise. Observe that the initial configuration of M corresponds to the lineage before it begins processing the input.

Now, when M reads the next non-λ input symbol, the next internal phase is started. If the update tape contains a 1, then the description of q is written to a work tape. The advice head moves over to the description of the next automaton and, using the work tape, locates the description of q. Since an update takes place, q was an exit state of A_k and hence an entry state of A_{k+1}. Thus, a description of q can be found within the description of the next automaton. Note also that $k+1 \leq n+1$, thus the description of A_{k+1} exists on the advice tape. If the update tape is empty, then this step can be skipped. Next, M looks up the transition from q corresponding to the input symbol on the advice tape. The description of the destination state is written to the work tape and the output symbol is remembered for now. Then, the head moves to the next state, using the work tape again. If this state is an exit state, then a 1 is written to the update tape. Otherwise, the update tape is erased. Finally, the output symbol is written to the output port and the ITM/A enters an external configuration, signaling its readiness to the environment.

It is left to the reader to verify that the translation realized by M matches the translation from $\mathcal{A}$. We see that the advice function has size $O(g \log g)$, and the work space uses at most $O(\log g)$ cells to store the states. Moving from state to state, and updating the automaton can all be done in finite time, so the internal phases are finite, and M is a valid ITM/A.

□

Theorem 7.19. *Let $\Phi : \Sigma^\infty \to \Omega^\infty$ be a translation. Suppose Φ is realized by an ITM/A with k tapes, with a space complexity g and advice of size f. Then Φ can be non-uniformly realized by a lineage of automata of complexity $O\left((\sum f)c^{kg}g^k\right)$, where c is the size of Σ.*

Proof. Suppose Φ is realized by an ITM/A M. We will simulate M with a lineage $\mathcal{A}$. Every state of A_n will correspond to an external configuration of M, with the contents of the advice tapes included. Each state is an exit state.

On a prefix of length n, the first n values of the advice have appeared on the advice tape of M. So, for the n-th automaton, which processes the prefixes of

length n, the contents of the advice tape are uniquely defined and have a length of $\sum_{i=1}^{n} f(i)$. The head of the advice tape can be in one of the $\sum_{i=1}^{n} f(i)$ positions. Since the work space of M is bounded by $g(n)$, each of the remaining tapes can have at most $c^{g(n)}$ different contents and the heads can be in one of the $g(n)$ positions. It follows that there are at most $O\left(\left(\sum_{i=1}^{n} f(i)\right) c^{kg(n)} g(n)^k\right)$ possible configurations. Now the simulation takes place by moving from external configuration to external configuration, following the transition function of M.

External λ's cannot be simulated, since the lineage does not have λ's in its input and output alphabets. Fortunately, they do not contribute to the computation and can safely be ignored by simulating only efficient inputs (as defined in Chapter 5).

Similarly, internal phases cannot be simulated. For this reason, there are no states corresponding to internal configurations. In the computation of an ITM/A, an internal phase can be viewed as computing a matching output symbol to the given non-λ input symbol. Such a computation is finite and deterministic, so it can be replaced with a single transition in the simulating lineage. An internal phase starts in an external configuration q, reads a non-λ symbol a, is followed by a finite sequence of internal configurations, with accompanying λ's at both ports and is concluded by an external configuration r, where a non-λ symbol b is produced. This phase is simulated by the single transition $\delta(q, a) = (r, b)$.

□

When we compare the results of Theorems 7.18 and 7.19, we see that an ITM/A can be more efficient than a lineage of automata. As an interesting application, the Theorems show that a lineage of polynomially bounded complexity can be simulated by an ITM/A with logarithmic space and polynomially bounded advice. Conversely, an ITM/A with logarithmic space and polynomially bounded advice can be simulated a lineage of polynomially bounded complexity.

7.5 Conclusions

The complexity hierarchy proved in this Chapter again uses a counting argument. The specifics are different from the ones used in Chapters 3 and 5 though. Indeed, the outcome of the results is different too. In Chapters 3 and 5, we showed that in the best case the hierarchy exists for functions f and g such that $f(n) < g(n)$ holds for infinitely many integers n. In this Chapter, we showed that $f(n) < g(n)$ for just one integer is already enough to establish that $SIZE(f)$ and $SIZE(g)$ are different classes. Of course, since the result can also be applied with the roles of f and g reversed, they are not subsets of each other. Thus, $SIZE(f)$ is a proper subset of $SIZE(g)$ if $f(n) \leq g(n)$ for every integer n and the inequality is strict for at least one integer.

The difference can be partially explained by the fact that the size of a lineage is a different resource than the length of an advice function. This difference also becomes clear when lineages are compared to ITM/A's: in the statement of Theorem 7.19, the size of the lineage depends on the space usage and the advice length of the ITM/A.

We conclude that ITM/A's and lineages highlight different aspects of evolving interactive systems. The advantage of lineages is that they model the evolutionary

aspect in a rather direct way. A disadvantage is that there is no distinction between the different kinds of resources that evolving systems can use. For this, ITM/A's are a better model.

CHAPTER 8

The Complexity of Evolving Systems

Evolving systems are systems that change over time, by internal factors or through adaptation to their environment. Examples of evolving systems are living organisms, communities of small cooperating organisms, stand-alone computers, or networks of linked computers. Exactly how these systems change is not of prime concern, what is interesting is the effect on the system. Up to now, several models have been introduced that capture this aspect of change. During the study of the complexity issues of these models, it became apparent that the models shared some fundamentally common structures. For instance, a number of complexity results involved techniques that could be used for all models. It is the intent of this Chapter to capture these common structures and generalize them into a framework for evolving systems. The aim is to make the framework abstract enough that a whole class of models can be described using the framework, yet on the other hand making it rigorous enough that we can apply common proof-techniques to instances of the framework. As a result, when a model for an evolving system is based on this framework, we automatically gain a toolbox of proof-methods that we can apply.

The structure of this Chapter is as follows. First, we give the general model of evolving systems. Then, we explore the topological aspects of the framework. Next, we apply the model to several instances, to illustrate the use of common proof-techniques, such as counting arguments and diagonalizing arguments. The instances are lineages of automata, advice functions (used by Turing machines) and recursive languages.

Lineages of automata form a straight-forward model of an evolving system and can be easily described using the framework. Advice functions can be viewed as sequences of strings and in this sense, the framework is a useful way to describe advice functions. The last instance shows that the framework is abstract enough to be useful for problems that do not fit the paradigm of evolving systems at first glance. Here, the construction of certain sequences of recursive languages is described as an example of the framework.

8.1 A New Framework

Based on the metaphor of evolution and some concepts from topology, a unifying framework to express evolving systems is presented. Let U be a universe of objects. These objects are the possible instances of an evolving system. Suppose a topology $\mathcal{T}$ on U is given.

An evolving system can be seen as a sequence of objects in U. We start with one object. This object represents the system in its current state. Eventually, an update takes place. This update, however, is limited in some way. The limiting effect is modeled by using sets of candidates. Given a starting object, a set of candidates for the next object contains all objects which are allowed to be the next object after an update takes place. From these objects, one candidate is chosen. Once the update has taken place, we choose a new set of candidates. The sequence of objects that we obtain this way is closely related to the sequence of sets of candidates: at any given time, the current set of candidates in the sequence depends on the object that is chosen in the preceding set of candidates, and the next object in the sequence depends on the availability of objects in the current set of candidates. The rate of evolution is then determined by the sizes of the sets of candidates.

There are several methods to choose the successor object in a set of candidates. An easy method is to take a random object, a more natural method is to choose an object that is better suited to its task (or more *fit*). This implies that we need a way to order the objects. The canonical way to do this is to map the sets of candidates to an ordered set, e.g., (a subset of) the real numbers $\mathbb{R}$. For every set of candidates C in the sequence, we define a *fitness* function $f : C \to R$, where R is an ordered set. This function somehow describes the fitness of the members of C.

The framework is set up to be as abstract as possible, to ensure that it can be deployed in a wide range of problems. The set of candidates can be constructed in ways to guarantee that the objects satisfy all kinds of restrictions. On the other hand, the fitness function can be used to rate the fitness of objects in a set of candidates on any number of desirable properties.

The concept of evolution as often applied in computer science is heavily connected with the concept of optimization; a system tries to evolve to an instance that is (locally) optimal with respect to the fitness function. In the terminology of optimization problems (especially local search variants), the objects become the feasible instances that we want to optimize, the fitness function is the cost function, the objects in the sequence are the candidates that have been chosen in executing the local search operation and the sets of candidates in the sequence are the neighborhoods of the objects in the execution. In this sense, an evolving system can be viewed as the result of a particular execution of a local search algorithm.

8.1.1 Complexity in the Framework

In this Chapter, the focus lies on the complexity of evolving systems. In particular, the instances that we analyze in detail show how to obtain results that demonstrate the existence of complexity hierarchies in evolving systems. These results

are usually achieved by showing the existence of objects of a given complexity that are demonstrably not of lower complexity.

We can construct the candidate sets in a way that guarantees that the objects that are chosen from the candidate sets have a complexity above a given threshold. This way, each object chosen from a candidate set is provably more complex than the threshold. Thus, a complexity hierarchy can be established by taking an object from a certain candidate set, since the threshold ensures that the object is of a given complexity, but not of a lower complexity.

The fitness function can be a direct measure of complexity, favoring objects of lower complexity. By choosing objects that optimize the fitness function within the candidate sets, the established hierarchy becomes finer and finer.

8.2 Topological Ideas in the Framework

Setting up the framework, we will also use notions from Topology. This has a number of advantages. For instance, we can use the concept of open sets to convey a sense of locality: in more abstract implementations of the framework where the universe has no meaningful distance functions, it is hard to imagine objects being related to each other. When we have a topology for the universe, we can use membership of the same open set as a measure of relatedness instead. If we choose large open sets, then many objects will be related to each other. Using smaller open sets leads to weaker relationships. This proves to be very useful and in many instances of the framework the sets of candidates will be (subsets of) open sets.

Topology serves as a basis for the study of (continuous) functions. So, we can use the wealth of knowledge in Topology as a library in which to browse for suitable results to apply to the fitness functions. We will give an example in this section. Another useful concept is that of (converging) sequences. In the rest of this section, the concept of convergence will be further explored.

8.2.1 Finding Optima within the Sets of Candidates

An important part of the framework is finding an object that optimizes the fitness function in a set of candidates. Depending on the characteristics of the set of candidates, this can be a non-trivial task. In fact, an optimal object does not even have to exist in all cases. However, under certain circumstances, we can use classical topological results to our advantage to ensure that an optimal object does exist.

Given a set of candidates C, let $f : C \to R$ be a fitness function from C to an ordered set R. If C is a compact set, R has the order topology and f is a continuous function, then a classical result in topology theory says that there is an object $x \in C$ for which $f(x)$ is an optimum in $f(C)$. This result is known as the maximum value theorem of calculus. Now, a recipe for selecting the next object is easy to give: given a compact set of candidates, choose an object which optimizes the fitness function and get a new set of candidates. In this set, we repeat the process.

We may not be so fortunate to have a continuous fitness function or a compact set of candidates. Even when we do, the optima may still be too hard to compute or even approximate in practical cases. However, there may be other means to ensure the existence of optimal elements. For instance, if the set of candidates is finite, an optimal element is guaranteed to exist. Otherwise, we can impose artificial bounds to choose the successor, e.g., "in the n-th stage, we choose an object from the top $100/n$ percent".

8.2.2 Convergence of Sequences

Another important part of the framework is the concept of sequence. A sequence is constructed using a fitness function and a sequence of sets of candidates, choosing at each step an object with the highest fitness value. While the optimization that takes place during the construction of a sequence is entirely local, it does have a global effect. We will explore the relationship between local optimizations and global effects on optimality. Let's clarify the situation first:

Definition 8.1. *Given a function f and a set S, an object x in S is* optimal *in S (with respect to f) if there is no object in y in S with $f(y) > f(x)$. An object x in S is* locally optimal *in S if there is a set O open in S such that x is optimal in O.*

Thus, an object can be optimal in a set of candidates, it can be locally optimal in the universe and it can be optimal in the universe. Note that an optimal object is locally optimal, but the converse does not hold. The underlying idea of the framework is that by choosing optimal in every set of candidates, the object that the sequence converges to will be optimal in the universe. It turns out that this is not always true. The sequence can diverge, in which case no optimal value is reached. Even when it converges, the resulting limit is not always locally optimal in the universe. Next, we give a sufficient condition on the sequence of sets of candidates, such that the sequence of objects converges to a limit that is locally optimal in the universe.

For now, we will assume that it is possible to find an object from every set of candidates C which is optimal in C. Furthermore, we assume that at the $(n+1)$-st stage we can choose an object that is at least as fit as the object that was chosen in the n-th stage, for every n. This implies that the sequence of fitness values is monotone.

To check if an object is locally optimal, we need to test a small open neighborhood containing the solution. Therefore, the union of the sets of candidates from which we are allowed to sample, should be an open set. On the other hand, it may very well be possible that a sequence converges to a solution outside of the union. But then we still can't tell whether the solution is optimal or not. Therefore, the union should be closed as well.

Theorem 8.2. *Let x_1, x_2,... be a sequence of objects that is constructed using a continuous fitness function f and a sequence of sets of candidates. Suppose that x_n is chosen optimally in the n-th set of candidates, for each n. Suppose that the sequence of objects converges. Let X be the union of the sets of candidates. If X is open and closed, then the limit is locally optimal in the universe.*

Proof. Suppose the sequence converges to an object x. Since f is continuous, the sequence $f(x_1)$, $f(x_2)$,... converges to $f(x)$. Since the sequence is monotonic, we know that $f(x) \geq f(x_n)$.

The set X is closed, so x belongs to it. X is also open, so X contains an open neighborhood O of x. Suppose now that x is not a local optimum. Then O contains an element y such that $f(y)$ is bigger than $f(x)$. As O is contained in X, one of the sets of candidates, say the n-th, contains y. But then $f(y) > f(x) \geq f(x_n)$, which contradicts the choice of x_n.

□

8.3 Instances of Evolving Systems

In this section, we show that the topological framework for evolving systems is suitable for the cases we have described in earlier Chapters. The focus can either be on a single object in the sequence, or on the sequence as a whole. The framework is flexible: it can be applied in either case, whether single objects model instances of the system, or the sequence itself is the model.

Working with different models of evolving systems, it becomes apparent that the same types of arguments are used again and again. Instead of having to find the right way to formulate the arguments correctly, it is much more efficient to recognize the common framework that lies at the base of the different models. The instances given in this section illustrate how to use the framework to apply diagonalizing arguments or counting arguments.

In each instance, we will define the universe, its topology and give sets of candidates with a corresponding fitness function.

8.3.1 Lineages of Automata as Evolving Systems

In this subsection, we redefine lineages of automata as a model of evolving systems, this time using the framework. Again, the focus is on the entire sequence. To illustrate common techniques for the framework, we restate Theorem 7.15 and prove it using the tools the framework gives us.

We take as our universe U the class of finite transducers. For each automaton, two subsets of its states are singled out: its entry states and its exit states (see Definition 6.1). A sequence of automata is seen as an entity by allowing a computation started by an automaton to be continued by the next automaton in the sequence, provided that the first automaton enters an exit state on its last step, and the next automaton continues in an entry state that corresponds to the exit state.

A sequence of objects $(A_i)_{i\geq 1}$ from U is a *lineage* when the set of exit states of A_i is a subset of the set of entry states of A_{i+1} (see Definition 6.2). A lineage computes a *translation*; a function that translates infinite input streams into output streams (see Definition 6.3). When A_i reaches an exit state during the computation, A_{i+1} resumes the computation in the corresponding entry state.

The efficiency with which lineages compute translations is measured by the number of states of the automata in a lineage (see also Chapter 7). Therefore, the

topology is related to the size of automata. It is defined as follows by giving the open sets:

$$B_n = \{\ A \quad | \quad A \text{ has } \leq n \text{ states } \}\ . \tag{8.1}$$

Let Φ be a translation with domain $D \subseteq \Sigma^\infty$. We try to construct a lineage that non-uniformly realizes Φ by choosing at each stage an automaton with the highest fitness value out of the available automata. If a part of the sequence has already been constructed, say $A_1, \ldots, A_n$, then we can define the fitness function f_n as follows: let A be an automaton such that its entry states contain all of the exit states of A_n. Then the fitness value $f_n(A)$ is the largest integer k such that all strings $u \in D$ of length at most k are valid inputs for the sequence $A_1, \ldots, A_n$, A. If the set of entry states of A does not contain the exit states of A_n, then the sequence $A_1, \ldots, A_n$, A cannot be turned into a lineage, thus we let $f_n(A) = -1$. Indeed, we see that automata with a higher fitness are capable of computing more of Φ.

Note that these functions are not continuous. However, we can still be sure that an optimum is achieved within compact sets.

Lemma 8.3. *Let $\Phi : \Sigma^\infty \to \Omega^\infty$ be a translation. Suppose a sequence $A_1, \ldots, A_n$ has been constructed to compute part of Φ. Then every non-empty compact set contains an element with optimal fitness with respect to f_n.*

Proof. A set is compact in the given topology if the number of states of an automaton in the set is bounded, say by M. Assume that A_n has $m \leq M$ exit states. Let $k = M - m$ and let $q_1, \ldots, q_k$ be states that are not exit states of A_n.

Consider a subset S of the compact set. An automaton A is contained in S if the following conditions are met:

- the input alphabet is Σ, the output alphabet is Ω,
- the entry states of A are the exit state of A_n,
- the remaining states form a subset of $\{q_1, \ldots, q_k\}$.

Under these conditions, there are only finitely many possibilities to finish the description of A. Therefore, S is a finite set.

Now, let B be an arbitrary automaton in the compact set. We can transform B into an automaton B' with input alphabet Σ and output Ω by restricting the transition function to transitions for which the input symbol belongs to Σ and the output symbol belongs to Ω. Note that B has the same fitness value as B'.

Assume that the set of entry states of B' contains all the exit states of A_n (otherwise, the fitness value of B' is -1). If B' contains entry states that are not exit states of A_n, then it can be transformed into an automaton B'' such that these superfluous entry states of B' are not entry states of B''. Then B' has the same fitness value of B''.

There is an isomorphism from B'' to an automaton B''' in S, such that the entry states are not relabeled. The fitness value of B'' is the same as that of its isomorphic copy B'''. We conclude that for an arbitrary automaton B, there are only finitely many possibilities for its fitness value. Thus, the compact set must contain an automaton with an optimal fitness value.

□

Given a function g, the sets of candidates are chosen as follows:

$$C_n = \{\ A \quad | \quad A \text{ has } \leq g(n) \text{ states } \}\ . \tag{8.2}$$

Note that the sets of candidates are compact sets. Now, we will use the framework to give an alternative proof of Theorem 7.15 We restate the Theorem here.

Theorem 8.4. *Let g be a non-decreasing function, and let f be a function such that $f(m) < g(m)$ for a certain integer m. Then there is a translation that can be non-uniformly realized by a lineage of size g, but not by any lineage of size f.*

Proof. We use the function g_c that is defined in (7.4). Let c be an integer such that $g_c(n) = g(n)$ for all $n \leq m$. We try to construct a lineage for the translation $\Phi^{g,c}$ with domain D (see Construction 7.1 and 7.2).

Suppose we are constructing a lineage $\mathcal{B}$ of size bounded by h. Consider the m-th automaton B_m. It has at most $f(m)$ states. Then there are two different finite strings u' and v' in D of length m, with strings u and v in D that extend u' and v' respectively, such that B_m enters the same state r after processing either u or v (see Figure 7.3). Combining (7.14) and (7.15), we conclude that there is a string x such that ux and vx belong to D, but $\mathcal{B}$ produces the wrong output for either ux or vx. Let k be the length of the longest of the strings. Then the fitness of B_n is less than k, for every $n \geq m$. Thus, a lineage with size bounded by h cannot non-uniformly realize $\Phi^{g,c}$.

On the other hand, for the lineage $\mathcal{A}$ that was constructed in Proposition 7.10, the n-th automaton A_n has a fitness value of n. Thus, $\mathcal{A}$ can non-uniformly realize $\Phi^{g,c}$.

□

Remark 8.5. Note that the lineage $\mathcal{A}$ that is used to compute $\Phi^{g,c}$ is not necessarily optimal for every n. There is no guarantee that choosing an optimal automaton at each stage even leads to a solution. If an optimal solution is required, we can modify the fitness function f_n to $f'_n(A) = 1$ if $f_n(A) > f_{n-1}(A_{n-1})$ and $f'_n(A) = 0$ otherwise. The lineage $\mathcal{A}$ is optimal if we use the fitness function f'_n.

8.3.2 Evolving Systems in Non-uniform Complexity Theory

In this subsection, we view advice functions as instances of an evolving system. In this case, the focus is on the entire sequence. We restate a simplified version of Theorem 3.22 and prove it in the context of the framework.

We let the universe U be the set of finite binary strings. The topology is given by the open sets

$$B_n = \{\ w \in U \quad | \quad \text{length of } w \text{ is } \leq n\ \}\ . \tag{8.3}$$

A sequence of strings can be seen as a function from $\mathbb{N}$ to the set of strings. Thus, a sequence of strings corresponds to an *advice function* (see Definition 3.1). A Turing machine may use an advice function to consult some extra information when it needs to decide whether an input should be accepted or not. For an advice

function α, the string $\alpha(n)$ contains helpful information for all the strings of length n. See Chapter 3 for a more formal introduction to the matter.

Given a function $g : \mathbb{N} \to \mathbb{N}$, the sets of candidates are defined as:

$$C_n = \{\ w \in U \quad | \quad \text{length of } w \text{ is } = g(n)\ \}\ . \tag{8.4}$$

Observe that an advice function is of size g if its strings are taken from the sets of candidates.

In this particular case, we will show that there is a language that can be decided by a particular Turing machine with an advice of length g, but not by any Turing machine with any advice of length less than g. Of course, we know from Chapter 3 that this only makes sense if $g(n) \leq c^n$, for an integer $c > 1$.

Let Σ be an alphabet of size c. For a particular machine M and a string w, we denote by $L_n(M, w)$ the subset of Σ^n that is decided by M with the advice string w. Recall the CS problem from Definition 3.5. Let M be a machine that decides CS_Σ. Consider an enumeration of all Turing machines $M_1, M_2, \ldots$ We use the machine M to define the fitness function: for a string w, we let $f_n(w) = 0$ if $L_n(M, w)$ equals $L_n(M_n, v)$ for a certain binary string v of length less than $g(n)$. Otherwise, $f_n(w) = 1$.

Theorem 8.6. *There is a language that can be decided by a Turing machine with an binary advice of length g, but not by any Turing machine with any binary advice of length less than g.*

Proof. Since the fitness functions have a finite image, each domain has an optimum. In the sets of candidates, this optimum is 1, since the number of different advice strings of length less than $g(n)$ equals

$$\sum_{i=0}^{g(n)-1} 2^i = 2^{g(n)} - 1\ . \tag{8.5}$$

By Proposition 3.12, M can decide $2^{g(n)}$ different subsets of Σ^n with advices of size g. Therefore, at least one of the advice strings of size $g(n)$ helps M to decide a subset that is not decided by M_n with any of the available $2^{g(n)} - 1$ advice strings. Thus C_n, which contains all advice strings of length $g(n)$, has an element with a fitness of 1.

With the sets of candidates given and the fitness function defined, we can choose an advice function α by maximizing in each compact set. The language L that M decides with advice α cannot be decided by any machine with advice of length less than g, as we will see.

Suppose that L is decided by a machine M_n with advice of size less than g. Let v be the n-th advice string. The length of v is less than $g(n)$. Consider the advice string $\alpha(n)$. Since it is optimal, its fitness is 1. On the other hand, the set of strings that M decides with the advice $\alpha(n)$ is the same set that M_n decides with v. This implies that the fitness of $\alpha(n)$ is 0. This is a contradiction, so our assumption was incorrect: L cannot be decided by any machine with an advice of length less than g.

□

8.3.3 Evolving Systems of Recursive Languages

In this subsection, we consider the construction of a recursive language as an evolving system. In this case, the focus lies on the union of the languages in the sequence; the sequence itself is just the means of constructing the desired language. In particular, we try to construct languages that can be accepted as efficiently as possible while satisfying certain constraints. To illustrate the usage of the framework, we restate a well-known classical result in complexity theory, namely that there are languages that can be accepted by a Turing machine in time $g \log g$, but not in time g, a result due to Hennie and Stearns[17]. This is done by constructing a language that can be accepted by a Turing machine as efficient as possible, under the constraint that it may not be accepted within time g. It turns out that the method employed by Hennie and Stearns[17] can be easily embedded in the framework. The method is based on the fact that any k-tape Turing machine running in time g can be simulated by a two-tape Turing machine running in time $O(g \log g)$, for any integer k. Similarly, the framework is based on this fact. The good thing about this is, if a more efficient means of simulating k-tape Turing machines for arbitrary integers k is discovered, this result can be plugged into the framework, improving the separation result.

The Universe and the Topology

First, we define the universe U. The universe consists of all the languages over the alphabet $\{0,1\}$. Thus, $U = \mathcal{P}(\{0,1\}^\star)$. We partition the set $\{0,1\}^\star$ into the following sets:

$$\begin{aligned} W_{-1} &= \{\ 0^i \quad \mid \quad i \in \mathbb{N}\ \}\ , \\ W_u &= \{\ 0^i 1 u \quad \mid \quad i \in \mathbb{N}\ \}\ , \end{aligned} \tag{8.6}$$

for $u \in \{0,1\}^\star$.

We will see later that we cannot guarantee that the fitness function gives us optimal solutions from the candidate sets. For this reason, the actual topology that we use is not important. Thus, we give U the trivial topology.

The Set of Candidates

Let g be a function and let $M_1, M_2, \ldots$ be an enumeration of all Turing machines. At the n-th stage in the evolution, the set containing the possible choices for the next language depends on the language L that has been constructed thus far. Given the binary description $\langle M_n \rangle$, let C_n be a subset of $\mathcal{P}\left(W_{\langle M_n \rangle}\right)$.

Two things are important when we decide which subsets of $W_{\langle M_n \rangle}$ to use for the n-th set of candidates. First, the union of an infinite sequence of recursive languages is not necessarily a recursive language. Using Proposition 8.7, we restrict the collections C_n such that the generated sequences will have a recursive union. Second, there must not be a Turing machine that decides the limit within time g. Again, we modify the collections C_n, this time ensuring that the union cannot be decided within time g. For this, we use Proposition 8.8.

Proposition 8.7. *Suppose L is the union of a sequence of languages L_1, L_2,..., constructed using the framework. Let N_1, ..., N_k be a partition of $\mathbb{N}$. For every $1 \leq i \leq k$, let M_i' be a Turing machine such that for all n, the machine M_i' decides L_n as a subset of $W_{\langle M_n \rangle}$ iff $n \in N_i$. Then L is recursive iff the complement of N_i is recursively enumerable for all $1 \leq i \leq k$.*

Proof. Note that a language is recursively enumerable iff there is a Turing machine that halts only for inputs that belong to the language.

Suppose L is recursive. Let M be a Turing machine that decides L. Note that M decides L_n as a subset of $W_{\langle M_n \rangle}$, for every n. Thus, $n \in N_i$ iff M_i' decides the same subset of $W_{\langle M_n \rangle}$ as M does.

We construct a Turing machine M_i'' that takes an integer n as input. Given n, it enumerates all strings in $W_{\langle M_n \rangle}$. It compares the output of M and M_i' on each of these strings. If the outputs differ for a string, then M_i'' halts. It follows that M_i'' halts only for strings in the complement of N_i.

For the converse, suppose M_i'' is a Turing machine that halts only for strings in the complement of N_i, for every $1 \leq i \leq k$. We construct a Turing machine M that takes a finite string x as input. Given x, it rejects x if it is not of the form $0^\star 1 \langle M' \rangle$ for a Turing machine M'. Otherwise, it finds the integer n such that $\langle M_n \rangle = \langle M' \rangle$. Thus, x belongs to $W_{\langle M_n \rangle}$.

Next, it runs the machines M_1'', ..., M_k'' simultaneously on n. Note that $n \in N_i$ for exactly one integer i. Thus, all the machines except M_i'' will eventually halt. When the correct integer i is found, M runs M_i' on x and accepts iff M_i' accepts x. Note that M_i' decides L_n as a subset of $W_{\langle M_n \rangle}$. It follows that M decides L, so L is recursive.

□

The proof of Proposition 8.8 illustrates that diagonalizing arguments can be easily given using the framework.

Proposition 8.8. *Let L be constructed using the framework. Suppose $L(M_n) \cap W_{\langle M_n \rangle}$ is not in C_n for every n. Then L cannot be decided within time g.*

Proof. Let n be an integer and let $S = L \cap W_{\langle M_n \rangle}$. It follows that $S \in C_n$, so by the assumptions of the Proposition, $S \neq L(M_n) \cap W_{\langle M_n \rangle}$. Therefore, S is not decided by M_n as a subset of $W_{\langle M_n \rangle}$ within time g. But then, M_n cannot decide L within time g. It follows that there is no Turing machine that can decide L within time g.

□

Let $\overline{M}$ be a Turing machine that takes as input a string u. If u is not of the form $0^\star 1 \langle M \rangle$ for a Turing machine M, then $\overline{M}$ rejects u. Otherwise, $\overline{M}$ simulates M on u and accepts iff M rejects u within time g. The machine $\overline{M}$ runs in time $O(g \log g)$, see Hennie and Stearns[17] for details. Note that $\overline{M}$ decides a subset of $W_{\langle M \rangle}$ that cannot be decided by M.

Choose an integer $k \geq 1$. Basically, the framework involves choosing for each n a Turing machine that decides a subset of $W_{\langle M_n \rangle}$. The limit L is then the union of these subsets. Following Proposition 8.7, we allow only finitely many different machines, distributed over the integers in a way that induces a partition of $\mathbb{N}$ into

sets with recursively enumerable complements. Furthermore, the machine chosen for an integer n should decide a different subset of $W_{\langle M_n \rangle}$ than the machine M_n does (see Proposition 8.8). We will do this by partitioning the integers into at most k disjoint intervals, each with its own corresponding machine. Observe that every interval of $\mathbb{N}$ is a recursive set, so their complements are recursively enumerable .

Now, we are ready to finish the definition of the candidate sets. For any n, the class of subsets of $W_{\langle M_n \rangle}$ depends on the choices that have been made previously, i.e., it depends on the parameters k, l (with $l \leq k$) and p, where k is the number of different machines that are allowed, l is the number of machines used so far and M is the machine that was used in the previous stage. For an arbitrary machine M, let $L_n(M)$ be short-hand for $L(M) \cap W_{\langle M_n \rangle}$. Let $\mathcal{M}$ be the class of Turing machines. Then the class C_n, with parameters k,l and M, is defined as

$$C_n(k,l,M) = \begin{cases} \{\ L_n(M') \quad | \quad M' \in \mathcal{M}, L_n(M') \neq L_n(M_n)\ \} & \text{if } l < k-1 \\ \{\ L_n(M), L_n(\overline{M})\ \} - \{\ L_n(M_n)\ \} & \text{if } l = k-1 \\ \{\ L_n(\overline{M})\ \} & \text{if } l = k \end{cases}, \tag{8.7}$$

Observe that $L_n(\overline{M}) \neq L_n(M_n)$, so C_n never contains $L_n(M_n)$.

Now, the construction of the sequence begins with the set of candidates $C_1(k, 0, \overline{M})$. Suppose the language L has been chosen from the set $C_n(k,l,M)$. If $L = L_n(M)$, then the next set will be $C_{n+1}(k,l,M)$. Otherwise, the next set is of the form $C_{n+1}(k,l,M')$, where M' is a Turing machine that decides L, under the restriction that $M' = \overline{M}$ if $l = k-1$.

The Fitness Function

Next, we need a fitness measure to select a language at each stage. Since our interest lies in running times, the fitness function could be a map from languages to functions. There are two problems with this approach. First, we know from Blum's Speed-up Theorem[4] that there are languages for which a most efficient algorithm does not exist. Thus, we cannot simply map a language to the smallest time in which a machine can decide the language. Second, the usual ordering on functions is not a total ordering; there are incomparable functions.

We partially solve the problem by considering not the set of functions, but instead the class of subsets of functions. Let L be a language and define the set of functions $T(L)$ such that $h \in T(L)$ iff there is a machine that decides L within time h, for any function h.

We define a partial ordering on the class of sets of functions by letting $X \leq Y$ iff Y is a subset of X. This ordering corresponds to our intuition; a language L can be decided as efficiently as a language L' iff $T(L')$ is a subset of $T(L)$ iff $T(L) \leq T(L')$. The ordering is not total, but it yields a lattice with a minimum element and a maximum element.

The fitness functions for the candidate sets depend on the previous choices. If the languages $L_1, \ldots, L_n$ were chosen in the first n steps, then the fitness function f_{n+1} for C_{n+1} is defined by $f_{n+1}(L) = T(L_1 \cup \cdots \cup L_n \cup L)$. We cannot guarantee that each set of candidates contains a language with optimal fitness. However, we can give a lower bound on the fitness of the languages in the $(n+1)$-st set of candidates based on the choice in the n-th set.

Proposition 8.9. *Let L_1, L_2, ... be a sequence of languages that is constructed using the framework. Then $f_n(L_n) \leq f_{n+1}(L_{n+1})$.*

Proof. We need to show that $T(L_1 \cup \cdots \cup L_{n+1})$ is a subset of $T(L_1 \cup \cdots \cup L_n)$. Observe that $L_i \cap L_j = \emptyset$ if $i \neq j$.

Let M be a Turing machine for $L_1 \cup \cdots \cup L_{n+1}$, running in time $h \in T(L_1 \cup \cdots \cup L_{n+1})$. Suppose M does not have enough time to check if an input u is in $W_{\langle M_i \rangle}$ for a machine M_i, with $1 \leq i \leq n+1$. Since u is not in $L_1 \cup \cdots L_{n+1}$ if it is not in $W_{\langle M_i \rangle}$ for a certain $1 \leq i \leq n+1$, it follows that u should be rejected in this case. We conclude that we can check for all $u \in L_1 \cup \cdots \cup L_{n+1}$ if u is in $W_{\langle M_i \rangle}$ for a certain $1 \leq i \leq n+1$ in $O(h)$ time.

Consider the Turing machine M', which works on inputs u as follows: first, M' checks, using at most $O(h)$ time, if u is in $W_{\langle M_{n+1} \rangle}$. If this is the case, or if M' doesn't have enough time to finish the check, then M' rejects u. Otherwise, M' uses M to process u and accepts iff M accepts u. This takes again $O(h)$ time. It follows that M' is a Turing machine that decides $L_1 \cup \cdots \cup L_n$, with a running time of $O(h)$. We conclude that $h \in T(L_1 \cup \cdots \cup L_n)$.

□

An interesting consequence of Proposition 8.9 is that choosing optimally in the set of candidates will not improve upon the choices that have been made earlier in the sequence. This is somewhat counter-intuitive. The explanation lies in the fact that the sequence is defined with the purpose of eventually satisfying the constraint, instead of just finding better and better solutions.

Theorem 8.10. *For every function g, there is a language L that can be decided within time $O(g \log g)$, but not within time $O(g)$.*

Proof. Let L be generated using the framework. By Proposition 8.8, L cannot be decided within time g. By Proposition 8.7, L is recursive.

During the construction of the sequence, at most k different machines were used. Now, a machine for L can use these k machines and a mechanism to decide which machine to use. Since we use intervals, the machine from the proof of Proposition 8.7 can be simplified as follows: Given input u, check if u is in $W_{\langle M_i \rangle}$ for an integer i. Then, check which of the k intervals contains i. Finally, use the machine corresponding to this interval to accept or reject u. If $k = 1$, then the checks can be done in linear time, so the running time of this machine is the running time of the slowest of the k machines. In this case, we get a running time of $O(g \log g)$.

□

While the $O(g \log g)$ bound is not a new one, the possibility of using k faster machines might lead to a better solution and should therefore be our aim. At first thought, it might seem reasonable to choose simple sets such as zero- or one-element sets, since they can be decided very efficiently. However, it follows from Proposition 8.8, that at least one of the machines should run in time $\omega(g)$. It is therefore smarter to choose all the machines with a complexity between $\omega(g)$ and $O(g \log g)$.

Another possibility for improvement lies in the choice for the machine $\overline{M}$. If a more efficient machine of $\overline{M}$ can be found, then the worst-case result of Theorem 8.10 can be improved to the running time of the new $\overline{M}$. In fact, such an

improvement has already been made. Before Hennie and Stearns[17] proved their result, it was known that any k-tape Turing machine running in time g can be simulated by a one-tape Turing machine in time g^2 (see Hartmanis and Stearns[16]). Using this fact, it was proved that there were languages that could be recognized in time $O(g^2)$, but not by any Turing machine running in time $O(g)$. Hennie and Stearns[17] improved this result, but the methodology for establishing the hierarchy remained the same and is abstracted in the framework of this Chapter.

8.4 Conclusions

The essential ingredients of an evolving system consist of a universe and a sequence of objects taken from this universe. A sequence is constructed by choosing objects from the universe to add to the sequence. By defining a topology on the universe, we can use the resulting compact sets to restrict the choices to just the objects in a compact set. By using compact sets of different shapes and sizes, we can control the rate of evolution of the resulting sequence. As we hinted multiple times in the earlier Chapters, the rate of evolution is a resource that an evolving system has available. As such, it is the basis of a complexity hierarchy. In practice, every evolving system that we encounter consists of countably many parts. However, there is no reason to forbid uncountable systems in theory. Thus, the framework and its tools apply equally well to uncountable sequences.

Fitness functions add another degree of control on the evolving system. They are used to optimize the goal that underlies the evolving system. Here, we assume that every evolving system is constructed for a reason, i.e., every system has a goal. In a biological sense, evolution is a random process without any goals. In computer science however, evolution is more seen as a means to an end, it is a tool with a purpose. If we use an evolving system to model a modern computing system, this view has some merit: we do not want to keep this big expensive interactive multitasking machine running without a purpose, do we?

The reason that we chose compact sets to control the rate of evolution is that they guarantee the existence of local optima when combined with continuous functions. If the fitness function is not continuous, it is thus not really necessary to use compact sets.

While having an abstract framework has many advantages, we must not forget that they are not the holy grail of science. In Chapter 3, we argued that sometimes improvements can be achieved by delving into the details that abstract frameworks try to hide. The amount of detail required to solve a problem varies with the problem itself. The more specific a problem is, the less likely it is that it is solvable in the context of an abstract framework. The complexity hierarchies for advice functions and lineages of automata that were established in Chapters 3 and 7 are clear examples of problems that can be solved using a framework. This is recognizable by the fact that they use the same techniques. Another problem that uses similar techniques is the complexity hierarchy for recursive languages.

Bibliography

1. J.L. Balcázar, J. Díaz, and J. Gabarró. *Structural Complexity I*, 2nd Edition. Springer–Verlag, Berlin, 1995.
2. J.L. Balcázar, J. Díaz, and J. Gabarró. *Structural Complexity II*. Springer–Verlag, Berlin, 1990.
3. L. Blum, F. Cucker, M. Shub, S. Smale. *Complexity and Real Computation*. Springer–Verlag, New York, 1997.
4. M. Blum. A Machine-Independent Theory of the Complexity of Recursive Functions, in: *Journal of the ACM*, Vol. 14, Nr. 2. New York, 1967, pp. 322–336.
5. M. Burgin. How We Know what Technology Can Do, in: *Communications of the ACM*, Vol. 44, Nr. 11. ACM Press, New York, 2001, pp. 83–88.
6. L. Cardelli. Global Computation, in: *ACM SIGPLAN Notices*, Vol. 32, Nr. 1. ACM Press, New York, 1997, pp. 66–68.
7. C.S. Calude, B. Pavlov. Coins, Quantum Measurements, and Turing's Barrier, Technical Report CDMTCS-170, Centre for Discrete Mathematics and Theoretical Computer Science, University of Auckland, 2001.
8. R.S. Cohen, A.Y. Gold. ω-Computations on Turing Machines, in: M. Nivat, M.S. Paterson (Eds.), *Theoretical Computer Science*, Vol. 6. North-Holland, Amsterdam, 1978, pp. 1–23.
9. S.A. Cook. Characterizations of Pushdown Machines in Terms of Time-Bounded Computers, in: *Journal of the ACM*, Vol. 18, Nr. 1. New York, 1971, pp. 4–18.
10. C. Damm, M. Holzer. Automata that Take Advice, in: J. Wiedermann, P. Hájek (Eds.), *Mathematical Foundations of Computer Science 1995*, Proceedings on the 20th Conference on Mathematical Foundations of Computer Science, Lecture Notes in Computer Science, Vol. 969. Springer–Verlag, Berlin, 1995, pp. 149–158.
11. P. van Emde Boas. Machine Models and Simulations, in: J. van Leeuwen (Ed.), *Handbook of Theoretical Computer Science*, Vol. A, Algorithms and Complexity. Elsevier Science, Amsterdam, 1990, pp. 1–66.
12. G. Etesi, I. Németi. Non-Turing Computations via Malament-Hogarth Space-Times, in: *International Journal of Theoretical Physics*, Vol. 41, Nr. 2. Springer Science, 2002, pp. 341–370.
13. D. Goldin. Persistent Turing Machines as a Model of Interactive Computation, in: K.-D. Schewe, B. Thalheim (Eds.), *Foundations of Information and Knowledge Systems*: First International Symposium, FOIKS 2000, Lecture Notes in Computer Science, Vol. 1762. Springer–Verlag, Berlin, 2000, pp.116–135.

14. D. Goldin, P. Wegner. Persistence as a Form of Interaction, Technical Report CS-98-07. Department of Computer Science, Brown University, 1998.
15. G.H. Hardy. *Orders of Infinity*, Cambridge Tracts in Mathematics and Mathematical Physics, Vol. 12. Cambridge Universal Press, London, 1924.
16. J. Hartmanis, R.E. Stearns. On the Computational Complexity of Algorithms, in: *Transactions of the AMS*, Vol. 117. 1965, pp. 285–306.
17. F.C. Hennie, R.E. Stearns. Two-Tape Simulation of Multitape Turing Machines, in: *Journal of the ACM*, Vol. 13, Nr. 4. New York, 1966, pp. 533–546.
18. M. Hermo, E. Mayordomo. A Note on Polynomial-Size Circuits with Low Resource-Bounded Kolmogorov Complexity, in: *Mathematical Systems Theory*, Vol. 27. Springer–Verlag, New York, 1994, pp. 347–356.
19. J.E. Hopcroft, R. Motwani, J.D. Ullman. *Introduction to Automata Theory, Languages and Computation*, 2nd Edition. Addison–Wesley, Boston, 2001.
20. O.H. Ibarra. On Two-Way Multihead Automata, in: *Journal of Computer and System Sciences*, Vol. 7. Academic Press, New York, 1973, pp. 28–36.
21. O.H. Ibarra, B. Ravikumar. Sublogarithmic-Space Turing Machines, Nonuniform Space Complexity, and Closure Properties, in: *Mathematical Systems Theory*, Vol. 21. Springer–Verlag, New York, 1998, pp. 1–17.
22. R.M. Karp, R.J. Lipton. Some Connections between Non-Uniform and Uniform Complexity Classes, in: *Proc. 12th Annual ACM Symposium on the Theory of Computing* (STOC'80). 1980, pp. 302–309.
23. R.M. Karp, R.J. Lipton. Turing Machines that Take Advice, in: *L'Enseignement Mathématiques*, II^e Série, Tome XXVIII. 1982, pp. 191–209.
24. S.C. Kleene. *Introduction to Metamathematics*. Van Nostrand, New York, 1952.
25. S. Kosub. Persistent Computations, Technical Report 217. Institut für Informatik, Julius–Maximilians–Universität Würzburg, 1998.
26. L.H. Landweber. Decision Problems for ω-Automata, in: *Mathematical Systems Theory*, Vol. 3, Nr. 4. Springer–Verlag, New York, 1969, pp. 376–384.
27. J. van Leeuwen, J. Wiedermann. On Algorithms and Interaction, in: M. Nielsen, B. Rovan (Eds.), *Mathematical Foundations of Computer Science 2000*, 25th International Symposium, Lecture Notes in Computer Science, Vol. 1893. Springer–Verlag, Berlin, 2000, pp. 99–113.
28. J. van Leeuwen, J. Wiedermann. The Turing Machine Paradigm in Contemporary Computing, in: B. Enquist, W. Schmidt (Eds.), *Mathematics Unlimited – 2001 and Beyond*. Springer–Verlag, Berlin, 2001, pp. 1139–1156.
29. J. van Leeuwen, J, Wiedermann. A Computational Model of Interaction in Embedded Systems, in: Technical Report UU-CS-2001-02. Institute of Information and Computing Sciences, Utrecht University, 2001.
30. J. van Leeuwen, J. Wiedermann. Beyond the Turing Limit: Evolving Interactive Systems, in: L. Pacholski, P. Ružička (Eds.), *SOFSEM 2001: Theory and Practice of Informatics*, 28th Conference on Current Trends in Theory and Practice of Informatics, Lecture Notes in Computer Science, Vol. 2234. Springer–Verlag, Berlin, 2001, pp. 90–109.
31. M. Li, P. Vitányi. *An Introduction to Kolmogorov Complexity and its Applications*, 2nd Edition. Springer–Verlag, New York, 1997.
32. R. Milner. Elements of Interaction, in: *Communications of the ACM*, Vol. 36, Nr. 1. ACM Press, New York, 1993, pp. 78–89.
33. M.L. Minsky. *Computation: Finite and Infinite Machines*, 2nd Print. Prentice–Hall, London, 1974.
34. J. Munkres. *Topology a First Course*. Prentice–Hall, Englewood Cliffs, New Jersey 07632, 1975.
35. P. Odifreddi. *Classical Recursion Theory*. North-Holland, Amsterdam, 1989.

36. P. Orponen. An Overview of the Computational Power of Recurrent Neural Networks, in: *Proceedings of the Finnish AI Conference*, Vol. 3, "AI of Tomorrow". Finnish AI Society, Vaasa, 2000, pp. 89–96.
37. H. Petersen. The Head Hierarchy for Oblivious Finite Automata with Polynomial Advice Collapses, in: L. Brim, J, Gruska, J. Zlatuška (Eds.), *Mathematical Foundations of Computer Science 1998*, 23rd International Symposium, MFCS '98, Lecture Notes in Computer Science, Vol. 1450. Springer–Verlag, Berlin, 1998, pp. 296–304.
38. M. Prasse, P. Rittgen. Why Church's Thesis Still Holds. Some Notes on Peter Wegner's Tracts on Interaction and Computability, in: *Computer Journal*, Vol. 41, Nr. 6. 1998, pp. 357–362.
39. L. Staiger. ω-Languages, in: G. Rozenberg, A. Salomaa (Eds.), *Handbook of Formal Languages*, Vol. 3, Beyond Words. Springer–Verlag, Berlin, 1997, pp. 339–387.
40. S. Stepney, S.L. Braunstein, J.A. Clark, A. Tyrrell, A. Adamatzky, R.E. Smith, T. Addis, C. Johnson, J. Timmis, P. Welch, R. Milner, D. Partridge. Journeys in Non-Classical Computation I: A Grand Challenge for Computing Research, in: *International Journal of Parallel, Emergent and Distributed Systems*, Vol. 20, Nr. 1. 2005, pp. 5–19.
41. W. Thomas. Automata on Infinite Objects, in: J. van Leeuwen (Ed.), *Handbook of Theoretical Computer Science*, Vol. B, Formal Methods and Semantics. Elsevier Science, Amsterdam, 1990, pp. 134–191.
42. A. Turing. On Computable Numbers with an Application to the 'Entscheidungsproblem', in: *Proceedings of the London Mathematical Society*, Vol. 2. 1936, pp. 230–265.
43. P. Verbaan, J. van Leeuwen, J. Wiedermann. Lineages of Automata, in: J. Karhumäki, H. Maurer, G. Păun, G. Rozenberg (Eds.), *Theory is Forever*, Essays Dedicated to Arto Salomaa on the Occasion of His 70th Birthday, Lecture Notes in Computer Science, Vol. 3113. Springer–Verlag, Berlin, 2004, pp. 268–281.
44. K. Wagner, G. Wechsung. *Computational Complexity*. VEB Deutscher Verlag der Wissenschaften, Berlin, 1986.
45. P. Wegner. Interaction as a Basis for Empirical Computer Science, in: *ACM Computing Surveys*, Vol. 27. 1995, pp. 45–48.
46. P. Wegner. Why Interaction is More Powerful than Algorithms, in: *Communications of the ACM*, Vol. 40, Nr. 5. ACM Press, New York, 1997, pp. 80–91.
47. P. Wegner. Interactive Foundations of Computing, in: *Theoretical Computer Science*, Vol. 192 Elsevier Science, Amsterdam, 1998, pp. 315–351.
48. P. Wegner, D. Goldin. Computations beyond Turing Machines, in: *Communications of the ACM*, Vol. 46., Nr. 4. ACM Press, New York, 2003, pp. 100–102.
49. J. Wiedermann, J. van Leeuwen. Emergence of a Super-Turing Computational Potential in Artificial Living Systems, in: J. Kelemen, P. Sosík (Eds.), *Advances in Artificial Life*, 6th European Conference (ECAL 2001), Lecture Notes in Artificial Intelligence, Vol. 2159. Springer–Verlag, Berlin, 2001, pp. 55–65.
50. J. Wiedermann, J. van Leeuwen. The Emergent Computational Potential of Evolving Artificial Living Systems, in: *AI Communications*, Vol. 15, Nr. 4. IOS Press, Amsterdam, 2002, pp. 205–215.
51. J. Wiedermann, J. van Leeuwen. Relativistic Computers and Non-Uniform Complexity Theory, in: C.S. Calude, M.J. Dinneen, F. Peper (Eds.), *Unconventional Models of Computation*: Proceedings of the Third International Conference, UMC 2002, Lecture Notes in Computer Science, Vol. 2509. Springer–Verlag, Berlin, 2002, pp. 287–299.

Acknowledgements

I would like to take this opportunity to say a few words of thanks to those who have been so helpful over the past few years.

First, I would like to thank my promotor Jan van Leeuwen, who has been my daily supervisor. Although he has a *very* busy schedule, he still found time for me whenever I needed his advice. I have very much enjoyed our discussions, both on technical issues as on matters of a more philosophical nature. These discussions never failed to motivate me when I needed motivation the most. His many suggestions and improvements have helped shape this thesis.

Second, I would like to thank my promotor Jiří Wiedermann. His suggestions for improvements have been very helpful. I appreciate the fact that he has managed to read large parts of my thesis in very short periods of time, due to ever present time-constraints.

I also mention Jaap van Oosten, who supervised my masters thesis. He convinced me to pursue a PhD degree when I was uncertain about my future career.

Next, I would like to thank the members of the reading committee, Peter van Emde Boas, John-Jules Meyer, Jaap van Oosten, Grzegorz Rozenberg and Cees Witteveen, for reading my thesis and for their helpful suggestions and discussions.

I would like to thank Guido Diepen, Jan van Leeuwen, Peter Lennartz, Iris Reinbacher, Marinus Veldhorst, Jiří Wiedermann and Thomas Wolle for valuable technical discussions and suggestions. Furthermore, I would like to thank Marjan van den Akker for her moral support. The stories about her own PhD research have been truly inspiring. A word of thanks goes to all my colleagues at the Department of Information and Computing Sciences of Utrecht University. Lastly, I would like to thank my friends and my family, especially my mother, for their endless support. Thank you all!

Samenvatting

Al voor de uitvinding van de eerste elektronische rekenmachines in het midden van de vorige eeuw was er belangstelling voor de theoretische grenzen aan het rekenen door mens of machine. Met de ontwikkeling van computers ontstond een theorie van *berekenbaarheid* die een antwoord bood op de vraag naar de mogelijkheden van op zichzelf staande, geprogrammeerde rekenmachines. Om uitspraken over berekenbaarheid te doen, moet men beschikken over een passend wiskundig model van computersystemen. Voor de modellering van klassieke computers wordt meestal gekozen voor het Turing machine model[42], genoemd naar de wiskundige Alan Turing, of voor de "random access" machine of RAM, die nauwer aansluit bij de bekende Von Neumann architectuur.

De efficiëntie waarmee berekeningen door computers uitgevoerd kunnen worden, vergt een hele studie op zichzelf, de zogeheten *(berekenings-) complexiteitstheorie*. Ook deze theorie laat zich goed vormgeven met de genoemde machine modellen. De efficiëntie van een berekening wordt bepaald door te meten hoeveel tijd of geheugen een computersysteem gebruikt om een berekening uit te voeren. De efficiëntie van een algoritme wordt weergegeven met behulp van een functie die aangeeft hoeveel rekentijd of geheugen nodig is, afhankelijk van de omvang van de invoer. In de complexiteitstheorie worden typisch resultaten verkregen die, voor paren functies f en g waarbij f op een geschikte manier kleiner is dan g, aangeven dat er problemen zijn die wel binnen complexiteit $g(n)$ oplosbaar zijn maar niet binnen complexiteit $f(n)$.

De klassieke machine modellen en hun berekeningen lijken minder geschikt om de kenmerken van hedendaagse computersystemen weer te geven. Moderne computers staan zelden meer op zichzelf: ze zijn verbonden in netwerken, ze werken samen met andere computersystemen in voortdurend wisselende verbanden, ze zijn dag en nacht operatief en hun rekengedrag kan door wijzigingen in hun software in de loop der tijd veranderen. Meerdere onderzoekers (bijv. Wegner[45], Van Leeuwen en Wiedermann[27] en Stepney e.a.[40]) hebben hier de laatste jaren op gewezen en hebben betoogd dat andere modellen nodig zijn en een ruimer begrip van berekening nodig is dan voorheen in de klassieke modellen werd gebruikt. Men spreekt

hier soms van super-Turing modellen en van "hypercomputation". Daarbij moet opgemerkt worden dat er al heel wat eerdere berekeningsmodellen en -theorieën bestaan die aan die beschrijving voldoen, bijvoorbeeld de theorie van effectieve berekeningen op reële getallen en berekeningen met hybride neurale netwerken. Ook de modellering van oneindige berekeningen treft men al aan, bijvoorbeeld in de theorie van ω-automaten (zie Staiger[39]). De theorieën in dit proefschrift richten zich voornamelijk op het modelleren van systemen die zich aanpassen aan hun omgeving of anderszins wijzigen in de tijd, in dit proefschrift *evoluerende* systemen genoemd.

Het voornaamste aspect van evoluerende systemen is het feit dat deze systemen in de loop van een berekening kunnen veranderen. Dit aspect wordt op twee manieren gemodelleerd. Een eerste, directe methode gebruikt een *rij* van systemen om een evoluerend systeem te modelleren. De rij bestaat uit alle momentopnames van het evoluerende systeem, gerangschikt in de tijd. Aangezien er niet noodzakelijk een manier is om de veranderingen van een evoluerend systeem te sturen, is er niet a priori een effectieve beschrijving die alle systemen in de rij omvat. Zo'n rij wordt daarom een *niet-uniforme* rij genoemd, in analogie met eerdere modellen in de zogenaamde niet-uniforme complexiteitstheorie. Een tweede methode om evoluerende systemen te modelleren maakt gebruik van zogenaamde *adviesfuncties*, naar Karp en Lipton[22] die deze functies om andere redenen bestudeerden. Het idee van advies functies is dat een systeem wijzigt als gevolg van extra informatie "van buitenaf" die alleen afhangt van de grootte van de invoer, en dat die informatie door middel van aanroep van de adviesfunctie spontaan wordt ingezet (als bij een orakel). De combinatie van (vast) systeem en adviesfunctie is op deze manier op te vatten als een evoluerend systeem. De gebruikte adviesfuncties zijn in het algemeen weer niet-uniform.

Het is duidelijk dat er een verband is tussen rijen van systemen en het model met adviesfuncties. In Hoofdstuk 4 wordt dit verband uitgewerkt tot een algemeen equivalentieresultaat, dat een stelling van Karp en Lipton[22] generaliseert. Het resultaat wordt vervolgens gebruikt om rijen van eindige automaten en van push-downautomaten, al of niet met meerdere leeskoppen, te vergelijken met diverse andere modellen (zie Stellingen 4.25 en 4.42). Het blijkt onder andere dat de klasse van rijen van eindige automaten equivalent is aan $LOGSPACE/poly$, terwijl de klasse van rijen van push-downautomaten equivalent is aan $P/poly$. Meerdere resultaten blijken goed in dit equivalentieresultaat in te passen.

Vervolgens komt de analyse van "oneindige" modellen aan de orde, in het bijzonder van interactieve Turing machines. De interactieve Turing machine (ITM) werd voor het eerst beschreven door Van Leeuwen en Wiedermann[28, 30]. Het is een rekenmodel voor een reactief (of interactief) systeem, dat wil zeggen voor een model dat communiceert met zijn omgeving. Een ITM vertaalt een rij invoersymbolen naar een rij uitvoersymbolen. Een belangrijk aspect van interactieve systemen is dat hun berekeningen altijd voortgezet kunnen worden. Dit betekent dat ITM's oneindige berekeningen kunnen uitvoeren. In Hoofdstuk 5 wordt een complexiteitstheorie opgezet, speciaal gericht op dergelijke interactieve systemen. Het blijkt dat de meeste resultaten uit de klassieke complexiteitstheorie kunnen worden overgezet naar de nieuwe theorie, met de juiste definitie van ITM's. Als voorbeeld wordt een hiërarchieresultaat bewezen (Proposities 5.31 en 5.32): als f

en g niet-dalende functies zijn zodat $f \log f \in o(g)$ (of $f \in o(g)$), dan zijn er berekeningen die uitgevoerd kunnen worden door een ITM in tijd (of ruimte) $O(g)$, maar niet in tijd (of ruimte) $O(f)$.

Gezien de eerdere behandeling van evoluerende systemen, lijkt het voor de hand te liggen deze te modelleren door ITM's met een adviesmechanisme. Dit stelt ons in staat een eerste (niet-uniforme) complexiteitstheorie te ontwikkelen voor evoluerende systemen, zoals gezocht. Ook hier kunnen veel resultaten van de klassieke niet-uniforme complexiteitstheorie worden overgezet naar de nieuwe theorie, wat de geschiktheid van de gekozen modellering lijkt te ondersteunen. Als voorbeeld wordt wederom een hiërarchieresultaat bewezen. Dit resultaat is gebaseerd op de ideeën uit Hoofdstuk 3 (zie Stelling 5.42): als f en g functies zijn zodat $f \in o(g)$ en $g(n) \leq c^n$ voor een zekere $c \geq 1$, dan zijn er berekeningen die uitgevoerd kunnen worden met een advies van formaat $O(g)$, maar niet met een advies van formaat $O(f)$.

Een *stamlijn* ("lineage") van automaten is een model voor evoluerende systemen, gebaseerd op rijen van eindige automaten. In een stamlijn wordt niet alleen een rij van systemen weergegeven, maar ook een vorm van informatieoverdracht tussen opvolgende evoluties van een systeem. Net als ITM's met advies kunnen ook stamlijnen van eindige automaten gebruikt worden om een complexiteitstheorie van evoluerende systemen te ontwikkelen. De theorie van stamlijnen biedt een interessante, niet-conventionele generalisatie van de theorie van eindige automaten. Ook voor stamlijnen blijkt een hiërarchieresultaat te bewijzen (zie Stelling 7.15): als f en g functies zijn zodat $f(m) < g(m)$ voor tenminste een zekere m, dan zijn er berekeningen die uitgevoerd kunnen worden met een stamlijn van formaat g, maar niet met een stamlijn van formaat f. Verder worden een aantal resultaten gegeven die aantonen dat de beide complexiteitstheorieën, die met ITM's met advies en die gebaseerd op stamlijnen van eindige automaten, op een aanvaardbare manier gerelateerd zijn.

De methoden die gebruikt zijn om de klassieke resultaten over te zetten naar de nieuwe theorieën, maken gebruik van een aantal eigenschappen die de onderzochte modellen van evoluerende systemen gemeen hebben. In Hoofdstuk 8 wordt een algemeen raamwerk ontwikkeld waarin dergelijke systemen passen, gebruikmakend van concepten uit de wiskundige topologie. In dit raamwerk worden de modellen van evoluerende systemen gezien als rijen van objecten. Bij elk object in een rij hoort een verzameling objecten die de mogelijke opvolgers van het object representeren. Samen met een mechanisme om de opvolger uit zo'n verzameling te selecteren, vormen rijen objecten met bijbehorende verzameling het skelet voor een evoluerende systeem. In Hoofdstuk 8 wordt dit raamwerk nader uitgewerkt. Het blijkt geschikt voor een reeks van toepassingen, niet alleen beperkt tot evoluerende systemen. Het raamwerk kan mogelijk van nut zijn in andere contexten waarin vormen van evolutionaire systemen gemodelleerd dienen te worden.

Uit de resultaten van dit proefschrift blijkt dat het zeer goed mogelijk is om een complexiteitstheorie voor evoluerende systemen op te zetten. Methoden die gebruikt worden in de klassieke complexiteitstheorie van berekeningen kunnen, met de nodige aanpassingen, in de nieuwe kaders worden gebruikt en leiden tot generalisaties die een goed antwoord geven op de vraag hoe de rekenkracht van evoluerende systemen onderscheiden kan worden, in termen van geschikte maten

op de evolutie van dergelijke systemen. Voor praktisch relevante modelleringen zullen uiteindelijk de systemen met sterk beperkte complexiteitsmaten het meest interessant zijn.

Curriculum Vitae

Peter Rudolf Alexander Verbaan was born on September 21st, 1976 in Tegelen, the Netherlands. From 1989 to 1995, he received his preparatory education (Atheneum-level) at the Thomas College in Venlo. From 1995 to 2001, he studied Mathematics at Utrecht University. He passed his propaedeutic exam in June 1998 and graduated in June 2001, in both cases with distinction. His Masters thesis, entitled "Primes and their Residue Rings in Models of Open Induction", was completed under the supervision of Dr. Jaap van Oosten. During these years, he also followed several Computer Science courses, which resulted in passing the propaedeutic exam in Computer Science in July 2000, also with distinction. In July 2001, he started as a PhD student (AIO) at the Department of Information and Computing Sciences of Utrecht University under the supervision of Prof. dr. Jan van Leeuwen. In December 2005, he completed his PhD thesis.

Titles in the IPA Dissertation Series

J.O. Blanco. *The State Operator in Process Algebra*. Faculty of Mathematics and Computing Science, TUE. 1996-01

A.M. Geerling. *Transformational Development of Data-Parallel Algorithms*. Faculty of Mathematics and Computer Science, KUN. 1996-02

P.M. Achten. *Interactive Functional Programs: Models, Methods, and Implementation*. Faculty of Mathematics and Computer Science, KUN. 1996-03

M.G.A. Verhoeven. *Parallel Local Search*. Faculty of Mathematics and Computing Science, TUE. 1996-04

M.H.G.K. Kesseler. *The Implementation of Functional Languages on Parallel Machines with Distrib. Memory*. Faculty of Mathematics and Computer Science, KUN. 1996-05

D. Alstein. *Distributed Algorithms for Hard Real-Time Systems*. Faculty of Mathematics and Computing Science, TUE. 1996-06

J.H. Hoepman. *Communication, Synchronization, and Fault-Tolerance*. Faculty of Mathematics and Computer Science, UvA. 1996-07

H. Doornbos. *Reductivity Arguments and Program Construction*. Faculty of Mathematics and Computing Science, TUE. 1996-08

D. Turi. *Functorial Operational Semantics and its Denotational Dual*. Faculty of Mathematics and Computer Science, VUA. 1996-09

A.M.G. Peeters. *Single-Rail Handshake Circuits*. Faculty of Mathematics and Computing Science, TUE. 1996-10

N.W.A. Arends. *A Systems Engineering Specification Formalism*. Faculty of Mechanical Engineering, TUE. 1996-11

P. Severi de Santiago. *Normalisation in Lambda Calculus and its Relation to Type Inference*. Faculty of Mathematics and Computing Science, TUE. 1996-12

D.R. Dams. *Abstract Interpretation and Partition Refinement for Model Checking*. Faculty of Mathematics and Computing Science, TUE. 1996-13

M.M. Bonsangue. *Topological Dualities in Semantics*. Faculty of Mathematics and Computer Science, VUA. 1996-14

B.L.E. de Fluiter. *Algorithms for Graphs of Small Treewidth*. Faculty of Mathematics and Computer Science, UU. 1997-01

W.T.M. Kars. *Process-algebraic Transformations in Context*. Faculty of Computer Science, UT. 1997-02

P.F. Hoogendijk. *A Generic Theory of Data Types*. Faculty of Mathematics and Computing Science, TUE. 1997-03

T.D.L. Laan. *The Evolution of Type Theory in Logic and Mathematics*. Faculty of Mathematics and Computing Science, TUE. 1997-04

C.J. Bloo. *Preservation of Termination for Explicit Substitution*. Faculty of Mathematics and Computing Science, TUE. 1997-05

J.J. Vereijken. *Discrete-Time Process Algebra*. Faculty of Mathematics and Computing Science, TUE. 1997-06

F.A.M. van den Beuken. *A Functional Approach to Syntax and Typing*. Faculty of Mathematics and Informatics, KUN. 1997-07

A.W. Heerink. *Ins and Outs in Refusal Testing*. Faculty of Computer Science, UT. 1998-01

G. Naumoski and W. Alberts. *A Discrete-Event Simulator for Systems Engineering*. Faculty of Mechanical Engineering, TUE. 1998-02

J. Verriet. *Scheduling with Communication for Multiprocessor Computation*. Faculty of Mathematics and Computer Science, UU. 1998-03

J.S.H. van Gageldonk. *An Asynchronous Low-Power 80C51 Microcontroller*. Faculty of Mathematics and Computing Science, TUE. 1998-04

A.A. Basten. *In Terms of Nets: System Design with Petri Nets and Process Algebra*. Faculty of Mathematics and Computing Science, TUE. 1998-05

E. Voermans. *Inductive Datatypes with Laws and Subtyping – A Relational Model*. Faculty of Mathematics and Computing Science, TUE. 1999-01

H. ter Doest. *Towards Probabilistic Unification-based Parsing*. Faculty of Computer Science, UT. 1999-02

J.P.L. Segers. *Algorithms for the Simulation of Surface Processes*. Faculty of Mathematics and Computing Science, TUE. 1999-03

C.H.M. van Kemenade. *Recombinative Evolutionary Search*. Faculty of Mathematics and Natural Sciences, UL. 1999-04

E.I. Barakova. *Learning Reliability: a Study on Indecisiveness in Sample Selection*. Faculty of Mathematics and Natural Sciences, RUG. 1999-05

M.P. Bodlaender. *Scheduler Optimization in Real-Time Distributed Databases*. Faculty of Mathematics and Computing Science, TUE. 1999-06

M.A. Reniers. *Message Sequence Chart: Syntax and Semantics*. Faculty of Mathematics and Computing Science, TUE. 1999-07

J.P. Warners. *Nonlinear approaches to satisfiability problems*. Faculty of Mathematics and Computing Science, TUE. 1999-08

J.M.T. Romijn. *Analysing Industrial Protocols with Formal Methods*. Faculty of Computer Science, UT. 1999-09

P.R. D'Argenio. *Algebras and Automata for Timed and Stochastic Systems*. Faculty of Computer Science, UT. 1999-10

G. Fábián. *A Language and Simulator for Hybrid Systems*. Faculty of Mechanical Engineering, TUE. 1999-11

J. Zwanenburg. *Object-Oriented Concepts and Proof Rules*. Faculty of Mathematics and Computing Science, TUE. 1999-12

R.S. Venema. *Aspects of an Integrated Neural Prediction System*. Faculty of Mathematics and Natural Sciences, RUG. 1999-13

J. Saraiva. *A Purely Functional Implementation of Attribute Grammars*. Faculty of Mathematics and Computer Science, UU. 1999-14

R. Schiefer. *Viper, A Visualisation Tool for Parallel Program Construction*. Faculty of Mathematics and Computing Science, TUE. 1999-15

K.M.M. de Leeuw. *Cryptology and Statecraft in the Dutch Republic*. Faculty of Mathematics and Computer Science, UvA. 2000-01

T.E.J. Vos. *UNITY in Diversity. A stratified approach to the verification of distributed algorithms*. Faculty of Mathematics and Computer Science, UU. 2000-02

W. Mallon. *Theories and Tools for the Design of Delay-Insensitive Communicating Processes*. Faculty of Mathematics and Natural Sciences, RUG. 2000-03

W.O.D. Griffioen. *Studies in Computer Aided Verification of Protocols*. Faculty of Science, KUN. 2000-04

P.H.F.M. Verhoeven. *The Design of the MathSpad Editor*. Faculty of Mathematics and Computing Science, TUE. 2000-05

J. Fey. *Design of a Fruit Juice Blending and Packaging Plant*. Faculty of Mechanical Engineering, TUE. 2000-06

M. Franssen. *Cocktail: A Tool for Deriving Correct Programs*. Faculty of Mathematics and Computing Science, TUE. 2000-07

P.A. Olivier. *A Framework for Debugging Heterogeneous Applications*. Faculty of Natural Sciences, Mathematics and Computer Science, UvA. 2000-08

E. Saaman. *Another Formal Specification Language*. Faculty of Mathematics and Natural Sciences, RUG. 2000-10

M. Jelasity. *The Shape of Evolutionary Search Discovering and Representing Search Space Structure*. Faculty of Mathematics and Natural Sciences, UL. 2001-01

R. Ahn. *Agents, Objects and Events a computational approach to knowledge, observation and communication*. Faculty of Mathematics and Computing Science, TU/e. 2001-02

M. Huisman. *Reasoning about Java programs in higher order logic using PVS and Isabelle*. Faculty of Science, KUN. 2001-03

I.M.M.J. Reymen. *Improving Design Processes through Structured Reflection*. Faculty of Mathematics and Computing Science, TU/e. 2001-04

S.C.C. Blom. *Term Graph Rewriting: syntax and semantics*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2001-05

R. van Liere. *Studies in Interactive Visualization*. Faculty of Natural Sciences, Mathematics and Computer Science, UvA. 2001-06

A.G. Engels. *Languages for Analysis and Testing of Event Sequences*. Faculty of Mathematics and Computing Science, TU/e. 2001-07

J. Hage. *Structural Aspects of Switching Classes*. Faculty of Mathematics and Natural Sciences, UL. 2001-08

M.H. Lamers. *Neural Networks for Analysis of Data in Environmental Epidemiology: A Case-study into Acute Effects of Air Pollution Episodes*. Faculty of Mathematics and Natural Sciences, UL. 2001-09

T.C. Ruys. *Towards Effective Model Checking*. Faculty of Computer Science, UT. 2001-10

D. Chkliaev. *Mechanical verification of concurrency control and recovery protocols*. Faculty of Mathematics and Computing Science, TU/e. 2001-11

M.D. Oostdijk. *Generation and presentation of formal mathematical documents*. Faculty of Mathematics and Computing Science, TU/e. 2001-12

A.T. Hofkamp. *Reactive machine control: A simulation approach using χ*. Faculty of Mechanical Engineering, TU/e. 2001-13

D. Bošnački. *Enhancing state space reduction techniques for model checking*. Faculty of Mathematics and Computing Science, TU/e. 2001-14

M.C. van Wezel. *Neural Networks for Intelligent Data Analysis: theoretical and experimental aspects*. Faculty of Mathematics and Natural Sciences, UL. 2002-01

V. Bos and J.J.T. Kleijn. *Formal Specification and Analysis of Industrial Systems*. Faculty of Mathematics and Computer Science and Faculty of Mechanical Engineering, TU/e. 2002-02

T. Kuipers. *Techniques for Understanding Legacy Software Systems*. Faculty of Natural Sciences, Mathematics and Computer Science, UvA. 2002-03

S.P. Luttik. *Choice Quantification in Process Algebra*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2002-04

R.J. Willemen. *School Timetable Construction: Algorithms and Complexity*. Faculty of Mathematics and Computer Science, TU/e. 2002-05

M.I.A. Stoelinga. *Alea Jacta Est: Verification of Probabilistic, Real-time and Parametric Systems*. Faculty of Science, Mathematics and Computer Science, KUN. 2002-06

N. van Vugt. *Models of Molecular Computing*. Faculty of Mathematics and Natural Sciences, UL. 2002-07

A. Fehnker. *Citius, Vilius, Melius: Guiding and Cost-Optimality in Model Checking of Timed and Hybrid Systems*. Faculty of Science, Mathematics and Computer Science, KUN. 2002-08

R. van Stee. *On-line Scheduling and Bin Packing*. Faculty of Mathematics and Natural Sciences, UL. 2002-09

D. Tauritz. *Adaptive Information Filtering: Concepts and Algorithms*. Faculty of Mathematics and Natural Sciences, UL. 2002-10

M.B. van der Zwaag. *Models and Logics for Process Algebra*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2002-11

J.I. den Hartog. *Probabilistic Extensions of Semantical Models*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2002-12

L. Moonen. *Exploring Software Systems*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2002-13

J.I. van Hemert. *Applying Evolutionary Computation to Constraint Satisfaction and Data Mining*. Faculty of Mathematics and Natural Sciences, UL. 2002-14

S. Andova. *Probabilistic Process Algebra*. Faculty of Mathematics and Computer Science, TU/e. 2002-15

Y.S. Usenko. *Linearization in μCRL*. Faculty of Mathematics and Computer Science, TU/e. 2002-16

J.J.D. Aerts. *Random Redundant Storage for Video on Demand*. Faculty of Mathematics and Computer Science, TU/e. 2003-01

M. de Jonge. *To Reuse or To Be Reused: Techniques for component composition and construction*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2003-02

J.M.W. Visser. *Generic Traversal over Typed Source Code Representations*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2003-03

S.M. Bohte. *Spiking Neural Networks*. Faculty of Mathematics and Natural Sciences, UL. 2003-04

T.A.C. Willemse. *Semantics and Verification in Process Algebras with Data and Timing*. Faculty of Mathematics and Computer Science, TU/e. 2003-05

S.V. Nedea. *Analysis and Simulations of Catalytic Reactions*. Faculty of Mathematics and Computer Science, TU/e. 2003-06

M.E.M. Lijding. *Real-time Scheduling of Tertiary Storage*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2003-07

H.P. Benz. *Casual Multimedia Process Annotation – CoMPAs*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2003-08

D. Distefano. *On Modelchecking the Dynamics of Object-based Software: a Foundational Approach*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2003-09

M.H. ter Beek. *Team Automata – A Formal Approach to the Modeling of Collaboration Between System Components*. Faculty of Mathematics and Natural Sciences, UL. 2003-10

D.J.P. Leijen. *The λ Abroad – A Functional Approach to Software Components*. Faculty of Mathematics and Computer Science, UU. 2003-11

W.P.A.J. Michiels. *Performance Ratios for the Differencing Method*. Faculty of Mathematics and Computer Science, TU/e. 2004-01

G.I. Jojgov. *Incomplete Proofs and Terms and Their Use in Interactive Theorem Proving*. Faculty of Mathematics and Computer Science, TU/e. 2004-02

P. Frisco. *Theory of Molecular Computing – Splicing and Membrane systems*. Faculty of Mathematics and Natural Sciences, UL. 2004-03

S. Maneth. *Models of Tree Translation*. Faculty of Mathematics and Natural Sciences, UL. 2004-04

Y. Qian. *Data Synchronization and Browsing for Home Environments*. Faculty of Mathematics and Computer Science and Faculty of Industrial Design, TU/e. 2004-05

F. Bartels. *On Generalised Coinduction and Probabilistic Specification Formats*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2004-06

L. Cruz-Filipe. *Constructive Real Analysis: a Type-Theoretical Formalization and Applications*. Faculty of Science, Mathematics and Computer Science, KUN. 2004-07

E.H. Gerding. *Autonomous Agents in Bargaining Games: An Evolutionary Investigation of Fundamentals, Strategies, and Business Applications*. Faculty of Technology Management, TU/e. 2004-08

N. Goga. *Control and Selection Techniques for the Automated Testing of Reactive Systems*. Faculty of Mathematics and Computer Science, TU/e. 2004-09

M. Niqui. *Formalising Exact Arithmetic: Representations, Algorithms and Proofs*. Faculty of Science, Mathematics and Computer Science, RU. 2004-10

A. Löh. *Exploring Generic Haskell*. Faculty of Mathematics and Computer Science, UU. 2004-11

I.C.M. Flinsenberg. *Route Planning Algorithms for Car Navigation*. Faculty of Mathematics and Computer Science, TU/e. 2004-12

R.J. Bril. *Real-time Scheduling for Media Processing Using Conditionally Guaranteed Budgets*. Faculty of Mathematics and Computer Science, TU/e. 2004-13

J. Pang. *Formal Verification of Distributed Systems*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2004-14

F. Alkemade. *Evolutionary Agent-Based Economics*. Faculty of Technology Management, TU/e. 2004-15

E.O. Dijk. *Indoor Ultrasonic Position Estimation Using a Single Base Station*. Faculty of Mathematics and Computer Science, TU/e. 2004-16

S.M. Orzan. *On Distributed Verification and Verified Distribution*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2004-17

M.M. Schrage. *Proxima - A Presentation-oriented Editor for Structured Documents*. Faculty of Mathematics and Computer Science, UU. 2004-18

E. Eskenazi and A. Fyukov. *Quantitative Prediction of Quality Attributes for Component-Based Software Architectures*. Faculty of Mathematics and Computer Science, TU/e. 2004-19

P.J.L. Cuijpers. *Hybrid Process Algebra*. Faculty of Mathematics and Computer Science, TU/e. 2004-20

N.J.M. van den Nieuwelaar. *Supervisory Machine Control by Predictive-Reactive Scheduling*. Faculty of Mechanical Engineering, TU/e. 2004-21

E. Ábrahám. *An Assertional Proof System for Multithreaded Java -Theory and Tool Support-*. Faculty of Mathematics and Natural Sciences, UL. 2005-01

R. Ruimerman. *Modeling and Remodeling in Bone Tissue*. Faculty of Biomedical Engineering, TU/e. 2005-02

C.N. Chong. *Experiments in Rights Control - Expression and Enforcement*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2005-03

H. Gao. *Design and Verification of Lock-free Parallel Algorithms*. Faculty of Mathematics and Computing Sciences, RUG. 2005-04

H.M.A. van Beek. *Specification and Analysis of Internet Applications*. Faculty of Mathematics and Computer Science, TU/e. 2005-05

M.T. Ionita. *Scenario-Based System Architecting - A Systematic Approach to Developing Future-Proof System Architectures*. Faculty of Mathematics and Computing Sciences, TU/e. 2005-06

G. Lenzini. *Integration of Analysis Techniques in Security and Fault-Tolerance*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2005-07

I. Kurtev. *Adaptability of Model Transformations*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2005-08

T. Wolle. *Computational Aspects of Treewidth - Lower Bounds and Network Reliability*. Faculty of Science, UU. 2005-09

O. Tveretina. *Decision Procedures for Equality Logic with Uninterpreted Functions*. Faculty of Mathematics and Computer Science, TU/e. 2005-10

A.M.L. Liekens. *Evolution of Finite Populations in Dynamic Environments*. Faculty of Biomedical Engineering, TU/e. 2005-11

J. Eggermont. *Data Mining using Genetic Programming: Classification and Symbolic Regression*. Faculty of Mathematics and Natural Sciences, UL. 2005-12

B.J. Heeren. *Top Quality Type Error Messages*. Faculty of Science, UU. 2005-13

G.F. Frehse. *Compositional Verification of Hybrid Systems using Simulation Relations*. Faculty of Science, Mathematics and Computer Science, RU. 2005-14

M.R. Mousavi. *Structuring Structural Operational Semantics*. Faculty of Mathematics and Computer Science, TU/e. 2005-15

A. Sokolova. *Coalgebraic Analysis of Probabilistic Systems*. Faculty of Mathematics and Computer Science, TU/e. 2005-16

T. Gelsema. *Effective Models for the Structure of pi-Calculus Processes with Replication*. Faculty of Mathematics and Natural Sciences, UL. 2005-17

P. Zoeteweij. *Composing Constraint Solvers*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2005-18

J.J. Vinju. *Analysis and Transformation of Source Code by Parsing and Rewriting*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2005-19

M.Valero Espada. *Modal Abstraction and Replication of Processes with Data*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2005-20

A. Dijkstra. *Stepping through Haskell*. Faculty of Science, UU. 2005-21

Y.W. Law. *Key management and link-layer security of wireless sensor networks: energy-efficient attack and defense*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2005-22

E. Dolstra. *The Purely Functional Software Deployment Model*. Faculty of Science, UU. 2006-01

R.J. Corin. *Analysis Models for Security Protocols*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2006-02

P.R.A. Verbaan. *The Computational Complexity of Evolving Systems*. Faculty of Science, UU. 2006-03