Opleiding Informatica

Tape-quantifying Turing machines

in the arithmetical hierarchy

Simon Heijungs

Supervisors:

H.J. Hoogeboom & R. van Vliet

BACHELOR THESIS

Leiden Institute of Advanced Computer Science (LIACS)

www.liacs.leidenuniv.nl

25/08/2017

Abstract

An important subject in theoretical computer science is the distinction between decidable and undecidable problems. Decidability can be defined in many ways, but it is most popular to base the definition on Turing machines; problems that a Turing machine cannot solve are considered undecidable. Sometimes, it is useful to further classify the undecidable problems. There are systems that can distinguish different degrees of undecidability, but they are mostly based on logic. In this thesis, we introduce a new class of automata similar to Turing machines that can work on certain undecidable problems. This gives a more algorithmic way to look at those problems, which may be more intuitive to those with a computer science background.

Contents

1 Introduction **1**
1.1 The Origin of Computability Theory . . . 1
1.2 Setup of this Thesis . . . 2
2 Preliminaries **3**
2.1 Formal Languages . . . 3
2.2 Vectors . . . 3
2.3 Turing Machines . . . 3
2.4 Total Turing Machines . . . 6
2.5 Multi-tape Turing Machines . . . 6
3 Tape-quantifying Turing Machines **8**
4 The Arithmetical Hierarchy **11**
5 Properties of Tape-quantifying Turing Machines **14**
5.1 Position in the Arithmetical Hierarchy . . . 14
5.2 Closure Properties . . . 15
6 Discussion and Conclusions **18**
Bibliography **19**

Chapter 1

Introduction

1.1 The Origin of Computability Theory

Set theory was formulated by Cantor in 1895. Because sets can be used to define many concepts in mathematics, it would become the foundation of mathematics.

In 1897, the first paradox in set theory was found by Cesare Burali-Forti. More paradoxes followed soon and in in 1901, Bertrand Russell found a very fundamental paradox that would become the most famous. This caused a crisis in mathematics.

In an attempt to recover from the crisis, Hilbert attempted to create an axiomatic system that would contain all of mathematics such that it would be possible to prove that no paradoxes could arise and everything could be proven. He also looked for an algorithm that, given a formula, would either prove the formula itself or its negation. This is known as the *Entscheidungsproblem*.

Hilbert's idea turned out not to be possible. Gödel proved that no axiomatic system containing arithmetic could be both complete and consistent (first incompleteness theorem) and that no axiomatic system could prove its own consistency unless it was inconsistent (in which case it could prove both its own consistency and its own inconsistency).

The *Entscheidungsproblem* remained an important problem. Even though Hilbert's axiomatic system was not possible, could it perhaps be solved for other axiomatic systems? First order logic seemed like a good candidate because it was proven to be complete. The problem was that this was difficult to reason about without a formal definition of algorithms. In 1936, Church and Turing independently proved that no algorithm for solving the *Entscheidungsproblem* of first order logic existed. They both made their own definition of an algorithm. Although their definitions looked very different, they would turn out to be equivalent. They formulated the Church-Turing thesis, which states that no machine or process could do any computation outside of their definitions. Although this cannot be mathematically proven, it is widely believed to be true. This led to the

birth of computability theory, which revolves around the question what can and what cannot be computed in different *models of computation*. [Rob15]

1.2 Setup of this Thesis

In this thesis, we introduce a novel variation on Turing machines we call tape-quantifying Turing machines that is intended to be more powerful than normal Turing machines. We will formally define this model and show that it is indeed more powerful in Chapter 3. We will discuss a framework used to classify the computational power of different models in Chapter 4 and we will show where tape-quantifying Turing machines fit into this framework in Chapter 5.

Chapter 2

Preliminaries

2.1 Formal Languages

The automata described in this thesis all work with formal languages, which we shall henceforth simply call languages. Given a finite set of symbols Σ, called an *alphabet*, a set of finite sequences of symbols in Σ is called a language over Σ. The language of *all* finite strings over Σ is denoted Σ^*. The empty string is denoted λ.

Computability theory can be set up either based on formal languages or based on the natural numbers. In this thesis, we use strings at the fundamental levels but we often encode natural numbers as strings. We encode a number n by repeating a symbol (normally "1") n times (unary).

2.2 Vectors

We will use vectors of symbols and vectors of strings in this thesis. They are denoted by bold letters. $\boldsymbol{\lambda}$ will denote a vector where every element is the empty string and $\boldsymbol{\Delta}$ will denote a vector where every element is the empty square.

2.3 Turing Machines

Structure Turing machines comprise the most used model of computation. They can intuitively be seen as abstract machines that have an infinite tape on which they can go back and forth and store and read information. They are often depicted as in Figure 2.1.

More formally Turing machine is a 5-tuple $T = (Q, \Sigma, \Gamma, q_0, \delta)$ where:

- Q is a finite set of states.

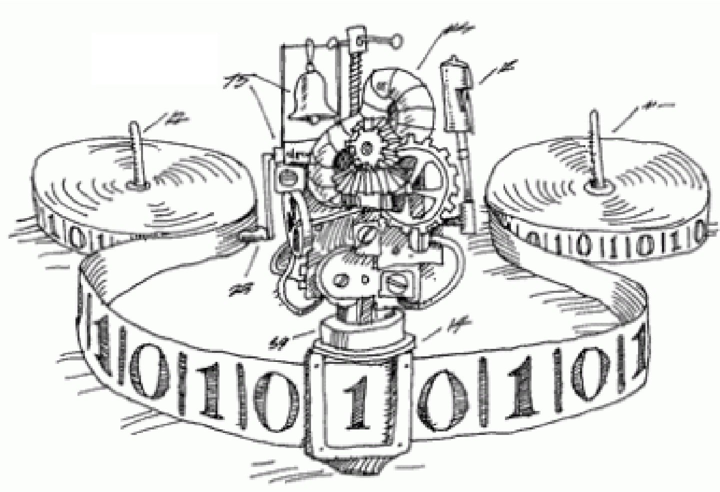

Figure 2.1: Artist's representation of a Turing machine.
(Adapted from https://nocloudinthesky.wordpress.com/2013/01/25/alan-turing-car-production-and-machines/)

- Σ is a finite set of symbols that can appear in the input of the machine.
- $\Gamma \supseteq \Sigma$ is a finite set of symbols that can appear on the tape. "Δ" is a designated symbol in $\Gamma \setminus \Sigma$ denoting an empty square.
- $q_0 \in Q$ is the initial state.
- $\delta : Q \times \Gamma \to Q \cup \{q_a, q_r\} \times \Gamma \times \{L, S, R\}$ is the transition function. q_a and q_r are special states called the accepting state and the rejecting state respectively.

The definition above only gives a static structure that contains the information pertaining to the Turing machine itself. The goal is to have it accept a language. To formalize this, we have to describe what configurations of a Turing machine are and how it can move from one configuration to another.

Configurations A configuration describes the conditions that a Turing machine can be in. It consists of the state in which it is, the contents of the tape and the position of the tape head on the tape. Formally, a configuration of such a machine is defined as a 4-tuple (q, u, γ, v) where:

- $q \in Q \cup \{q_a, q_r\}$ is the current state.
- $u, v \in \Gamma^*$ are the sequences before and after the tape head respectively, up to a point where only empty squares follow.
- $\gamma \in \Gamma$ is the symbol under the tape head.

The step relation The step relation describes how a Turing machine can move from one configuration to another. It is denoted as $\Rightarrow$. The Turing machine looks in which state it is and which symbol is under the tape head. It then uses its transition function to choose to what state to go, what to write and in which direction to

move the tape head. More formally, it is the minimal relation such that for every transition $\delta(q, \boldsymbol{\gamma}) = (q', \mathbf{w}, \mathbf{d})$, for all $\mathbf{u}, \mathbf{v}$, $(q, \mathbf{u}, \boldsymbol{\gamma}, \mathbf{v}) \Rightarrow (q', \mathbf{u}', \boldsymbol{\gamma}', \mathbf{v}')$ where:

if $d = S$	then $u' = u$, $\gamma' = w$ and $v' = v$.
if $d = L$ and $u = \lambda$	then $u' = \lambda$, $\gamma' = \Delta$ and $v' = wv$.
if $d = L$ and $u = ux$ for some $u \in \Gamma^*$ and $x \in \Gamma$	then $u' = u$, $\gamma' = x$ and $v' = wv$.
if $d = R$ and $v = \lambda$	then $u' = uw$, $\gamma' = \Delta$ and $v' = \lambda$.
if $d = R$ and $v = xv$ for some $v \in \Gamma^*$ and $x \in \Gamma$	then $u' = uw$, $\gamma' = x$ and $v' = v$.

Reachability Reachability, denoted $\Rightarrow^*$ is defined as the reflexive and transitive closure of the step relation.

Accepting a language The main goal of a Turing machine is to accept a language. This means it has to accept exactly the strings that are in the language. Informally, it starts with the string on the tape right after the tape head in the initial state and acccepts if it can eventually reach the accepting state by repeatedly applying the step relation. In more formal notation, the machine accepts a string $S \in \Sigma^*$ iff $(q_0, \lambda, \Delta, S) \Rightarrow^* (q_a, u, \gamma, v)$ for some u, γ, v.

Accepting tuples It is also possible to make a Turing machine to accept tuples of strings. The strings in the tuple are all initially placed on the tape one after another, separated by a Δ.

Diagrams To make them easier to understand, Turing machines are often given as diagrams. The states are drawn as circles with an identifier in the middle. If $\delta(q_A, \gamma_A) = (q_B, \gamma_B, d)$, an arrow is drawn from q_A to q_B with a label "$\gamma_A / \gamma_B, d$". The initial state q_0 gets an arrow from the word "start" pointing to it. The input alphabet Σ and the tape alphabet Γ are left implicit in the diagram, but may be given in a caption when needed. Transitions to q_r can be omitted; when no transition is drawn for a given (q, γ), it can be assumed that $\delta(q, \gamma) = (q_r, \gamma, S)$. When several transitions exist that share both their states, they are represented by one arrow with several labels in separate text lines. Figure 2.2 shows an example of such a diagram.

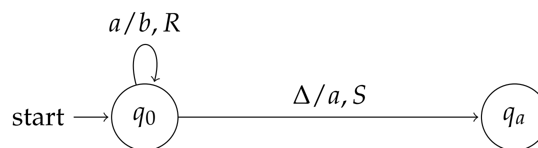

Figure 2.2: Example of a diagram for a Turing machine.

Encoding Turing machines work only on strings. Sometimes it is useful to let it solve a problem about other mathematical objects. This makes it necessary to translate those problems to languages.

We will be considering those problems where for any input, the answer "yes" or "no" has to be given, for example, the problem may be "Is the input number prime?" or "Does the input graph have a Hamiltonian path?" Those problems are called *decision problems*.

In this thesis, we will use unary encoding to encode natural numbers. A natural number n is represented as a symbol repeated n times.

It is also possible to represent graphs, vectors and even Turing machines as strings. Although we will not discuss a specific encoding, the possibility to make a Turing machine reason about other Turing machines will be important in this thesis.

Possibilites and limitations The class of languages that can be accepted by Turing machines is called the *recursively enumerable languages*, denoted RE. Many important decision problems can be solved by Turing machines. However, there are a number of problems, mostly relating to properties of Turing machines themselves that they cannot solve. For example: if the input is an encoded Turing machine, they cannot determine if the language it accepts is finite or not.

2.4 Total Turing Machines

A total Turing machine is a Turing machine that reaches either q_a or q_r for every legal input. The class of languages that can be accepted by total Turing machines is called the class of *recursive languages*, denoted REC. Decision problems that can be solved by total Turing machines are called decidable.

The class REC is a strict subset of RE. One important problem in $RE \setminus REC$ is the halting problem: given an encoding of a Turing machine and an input string for it, the question whether the Turing machine ever stops either in q_a or in q_r is not decidable, but is in RE. Likewise, the decision problem whether a given Turing machine accepts a given input, which is called *accepts*, is in $RE \setminus REC$ as well. However, if an upper bound n on the number of steps is given, the question whether a Turing machine accepts an input within n steps *is* decidable.

2.5 Multi-tape Turing Machines

Multi-tape Turing machines are an important variation on Turing machines. Intuitively, they can be thought of as Turing machines that have more than one tape. The first tape is initialized with the input and the other tapes are initialized empty. More formally, a multi-tape machine with n tapes is defined like Turing machines with the following changes:

Structure δ is now a function $Q \times \Gamma^n \to Q \cup \{q_a, q_r\} \times \Gamma^n \times \{L, S, R\}^n$.

Configurations A configuration is now a tuple $(q, \mathbf{u}, \boldsymbol{\gamma}, \mathbf{v})$ where $\mathbf{u}$ and $\mathbf{v}$ are now n-dimensional vectors of strings containing the symbols before and after the tape head respectively for each tape and $\boldsymbol{\gamma}$ is an n-dimensional vector of symbols containing the symbol under each tape head.

The step relation The step relation is now defined as the minimal relation such that for every transition $\delta(q, \boldsymbol{\gamma}) = (q', \mathbf{w}, \mathbf{d})$, for all $\mathbf{u}, \mathbf{v}$, $(q, \mathbf{u}, \boldsymbol{\gamma}, \mathbf{v}) \Rightarrow (q', \mathbf{u}', \boldsymbol{\gamma}', \mathbf{v}')$ iff for all $i \leq n$ the following is true:

if $d_i = S$ then $u'_i = u_i$, $\gamma'_i = w_i$ and $v'_i = v_i$.

if $d_i = L$ and $u_i = \lambda$ then $u'_i = \lambda$, $\gamma'_i = \Delta$ and $v'_i = w_i v_i$.

if $d_i = L$ and $u_i = ux$ for some $u \in \Gamma^*$ and $x \in \Gamma$ then $u'_i = u$, $\gamma'_i = x$ and $v'_i = w_i v_i$.

if $d_i = R$ and $v_i = \lambda$ then $u'_i = u_i w_i$, $\gamma'_i = \Delta$ and $v'_i = \lambda$.

if $d_i = R$ and $v_i = xv$ for some $v \in \Gamma^*$ and $x \in \Gamma$ then $u'_i = u_i w_i$, $\gamma'_i = x$ and $v'_i = v$.

Accepting a language The machine accepts a string $S \in \Sigma*$ iff $(q_0, \boldsymbol{\lambda}, \boldsymbol{\Delta}, \mathbf{S}) \Rightarrow^* (q_a, \mathbf{u}, \boldsymbol{\gamma}, \mathbf{v})$ for some $\mathbf{u}, \boldsymbol{\gamma}, \mathbf{v}$ where $\mathbf{S} = [S, \lambda, \lambda, \ldots, \lambda]$.

Diagrams Diagrams for multi-tape Turing machines are similar to those of normal Turing machines. The states are drawn in the same way and transitions are still drawn as arrows between them. However, the labels on the transition now give tuples instead of single elements for the letter read, the letter written and the direction of movement, for each tape as in Figure 2.3.

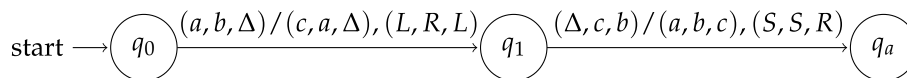

Figure 2.3: Example of a diagram for a multi-tape Turing machine with 3 tapes.

Chapter 3

Tape-quantifying Turing Machines

Tape-quantifying Turing machines are the main object of study in this thesis. Their definition is similar to that of a multi-tape Turing machine. The key difference is in the definition of accepting a string: the machine needs to reach the accepting state from an infinite set of initial configurations, rather than a single one. Since tape-quantifying Turing machines are such a central subject in this thesis, we give a full definition rather than just describing the differences.

Structure A tape-quantifying Turing machines has the same structure as multi-tape Turing machine with two tapes: a 5-tuple $T = (Q, \Sigma, \Gamma, q_0, \delta)$ where:

- Q is a finite set of states.
- Σ is a finite set of symbols that can appear in the input of the machine.
- $\Gamma \supseteq \Sigma$ is a finite set of symbols that can appear on the tape. "Δ" and "I" are designated symbols in $\Gamma \setminus \Sigma$.
- $q_0 \in Q$ is the initial state.
- $\delta : Q \times \Gamma^2 \to Q \cup \{q_a, q_r\} \times \Gamma^2 \times \{L, S, R\}^2$ is the transition function. q_a and q_r are special states called the accepting state and the rejecting state respectively.

Configurations A configuration describes the conditions that A tape-quantifying Turing machine can be in. It consists of the state in which it is, the contents of the tapes and the positions of the tape heads on the tapes. Formally, a configuration of such a machine is defined as a 4-tuple $(q, (u_1, u_2), (\gamma_1, \gamma_2), (v_1, v_2))$ where:

- $q \in Q \cup \{q_a, q_r\}$ is the current state.
- $u_i, v_i \in \Gamma^*$ are the sequences before and after the tape head on tape i respectively, up to a point where only empty squares follow.
- $\gamma_i \in \Gamma$ is the symbol under the the tape head of tape i.

The step relation The step relation, like in multi-tape machines the minimal relation such that for every transition $\delta(q, \boldsymbol{\gamma}) = (q', \mathbf{w}, \mathbf{d})$, for all $\mathbf{u}, \mathbf{v}$, $(q, \mathbf{u}, \boldsymbol{\gamma}, \mathbf{v}) \Rightarrow (q', \mathbf{u}', \boldsymbol{\gamma}', \mathbf{v}')$ iff for $i \in \{1, 2\}$ the following is true:

if $d_i = S$	then $u'_i = u_i$, $\gamma'_i = w_i$ and $v'_i = v_i$.
if $d_i = L$ and $u_i = \lambda$	then $u'_i = \lambda$, $\gamma'_i = \Delta$ and $v'_i = w_i v_i$.
if $d_i = L$ and $u_i = ux$ for some $u \in \Gamma^*$ and $x \in \Gamma$	then $u'_i = u$, $\gamma'_i = x$ and $v'_i = w_i v_i$.
if $d_i = R$ and $v_i = \lambda$	then $u'_i = u_i w_i$, $\gamma'_i = \Delta$ and $v'_i = \Delta$.
if $d_i = R$ and $v_i = vx$ for some $v \in \Gamma^*$ and $x \in \Gamma$	then $u'_i = u_i w_i$, $\gamma'_i = x$ and $v'_i = v$.

Reachability Reachability, denoted $\Rightarrow^*$ is defined as the reflexive and transitive closure of the step relation.

Accepting a language This is distinguishing feature of tape-quantifying Turing machines. Whereas multi-tape Turing machines with 2 tapes always start with an empty second tape, tape-quantifying Turing machines consider the result of starting with any finite string of I symbols (including the empty string) on the second tape. They accept iff *all* of those initialisations allow the accepting state to be reached.

More formally, a tape-quantifying Turing machine accepts a string $S \in \Sigma*$ iff for all c, $(q_0, \boldsymbol{\lambda}, \boldsymbol{\Delta}, \mathbf{S}_c) \Rightarrow^* (q_a, \mathbf{u}, \boldsymbol{\gamma}, \mathbf{v})$ for some $\mathbf{u}, \boldsymbol{\gamma}, \mathbf{v}$ where $\mathbf{S}_c = [S, I^c]$.

Diagrams Diagrams of tape-quantifying Turing machines look the same as diagrams for multi-tape Turing machines with two tapes. States are represented by circles with arrows between them representing the transitions. If $\delta(q_A, (\gamma_{A,1}, \gamma_{A,2})) = (q_B, (\gamma_{B,1}, \gamma_{B,2}), (d_1, d_2))$, an arrow is drawn from q_A to q_B with a label "$(\gamma_{A,1}, \gamma_{A,2})/(\gamma_{B,1}, \gamma_{B,2}), (d_1, d_2)$". The initial state q_0 gets an arrow from the word "start" pointing to it. If the destination of a transition is the rejecting state, it may be omitted in the diagram. When several transitions exist that share both their states, they are represented by one arrow with several labels in seperate text lines.

Terminology We will call the machine with a specific initialisation of the second tape a *count instance* of that tape-quantifying Turing machine and call the number of Is in its initialisation its *count*. A count instance accepts if it reaches the accepting state by following the rules of a multi-tape machine so a tape-quantifying Turing machine accepts an input iff all of its count instances accept it.

Examples To illustrate how tape-quantifying Turing machines work, we give an example in Figure 3.1. It accepts a string S iff S is a prime number encoded in unary.

The main computation happens in the states q_5 and q_6. Those states check whether the input is dividable by the count. Here the input is divided by the count and accepted iff there is a remainder. The rest of the states take care of the edge cases: the facts that a prime number is allowed to be dividable by 1 and itself, that we cannot divide a number by 0 and that 0 and 1 are not prime. The whole machine accepts iff for each count instance, either the count is equal to 0, 1 or the input, or the input is not dividable by the count and the input is not 0 or 1.

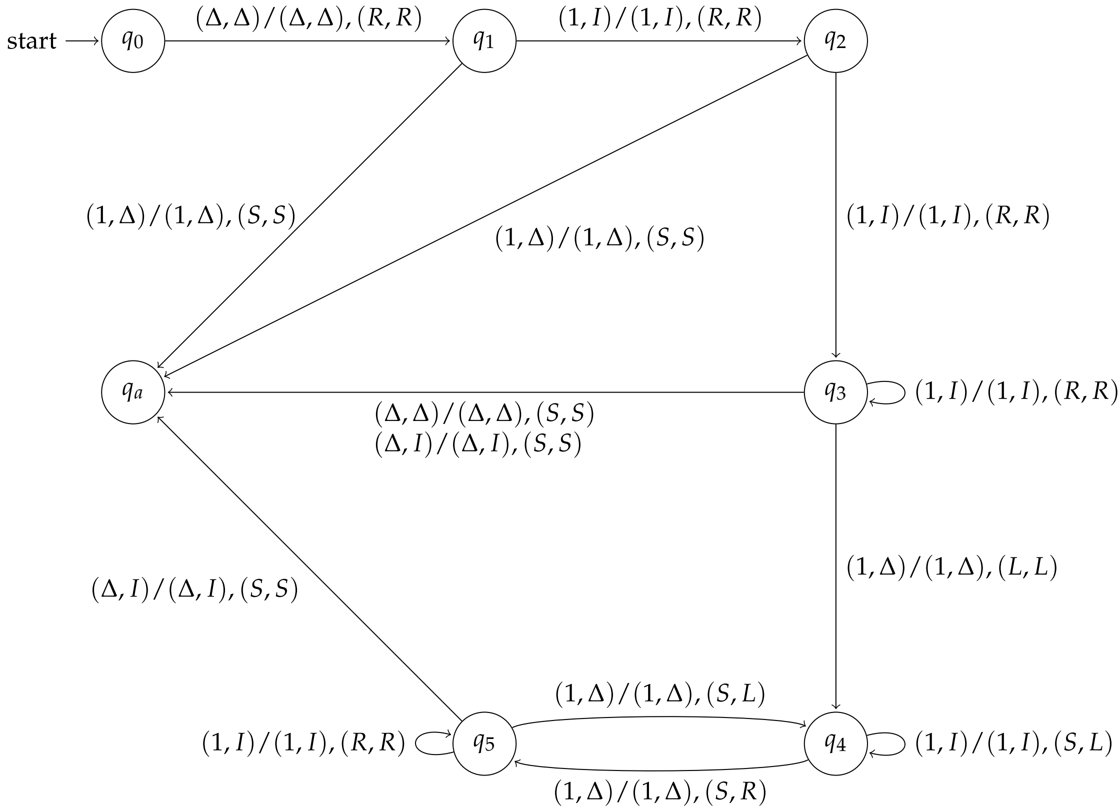

Figure 3.1: Example of a tape-quantifying Turing machine. It uses input alphabet $\{1\}$ and tape alphabet $\{1, I, \Delta\}$. It accepts a string S iff $S = 1^n$ where n is a prime number.

Tape-quantifying Turing machines are more powerful than normal Turing machines. To illustrate this, we show how they can accept the complements of all recursively enumerable languages. Ordinary Turing machines cannot do this.

Theorem 1. *Tape-quantifying Turing machines can accept the complement of recursively enumerable languages.*

Proof. Given a Turing machine T that accepts a recursively enumerable language L, construct a tape-quantifying Turing machine T' with a state q' for each state q in T. For each transition $q_1 \times \gamma_1 \mapsto q_2 \times \gamma_2 \times d$ in T, make a transition $q'_1 \times (\gamma_1, I) \mapsto q'_2 \times (\gamma_2, I) \times (d, R)$ in T'. Add a new initial state q_0 and make a transition $q_0 \times (\Delta, \Delta) \mapsto q'_0 \times (\Delta, \Delta) \times (S, R)$ in T'. Swap the accepting and the rejecting states and make a transition $q'_1 \times (\gamma_1, \Delta) \mapsto q_a \times (\gamma_1, \Delta) \times (S, S)$ for each γ_1 and q_1.

T' simulates T while counting the steps it takes until the step count equals the number of Is that was initially on its second tape. For the tape-quantifying Turing machine to accept, this has to be the case for any number of Is. We can thus conclude that T' accepts iff T does not accept after any number of steps. Therefore, T' accepts the complement of L. □

Chapter 4

The Arithmetical Hierarchy

The arithmetical hierarchy is a system for describing the level of complexity of sets based on the formulas that can describe them [HR67]. Since languages are sets, it can be used to classify them as well. The hierarchy consists of infinitely many classes of the form Σ_n^0, Π_n^0 and Δ_n^0. The reason for the 0 in superscript is that this hierarchy is part of a bigger hierarchy called the analytical hierarchy where other superscripts are used. The definition of the analytical hierarchy is beyond the scope of this thesis.

A set of is in Σ_n^0 iff it can be described by a formula of the form $\{x : \exists y_1 \forall y_2 \exists y_3 \ldots \forall y_n P(x, y_1, y_2, y_3 \ldots y_n)\}$ or $\{x : \exists y_1 \forall y_2 \exists y_3 \ldots \exists y_n P(x, y_1, y_2, y_3 \ldots y_n)\}$ depending on whether n is even or odd, where P is a decidable predicate.

Similarly, a set is in Π_n^0 iff it can be described by a formula of the form $\{x : \forall y_1 \exists y_2 \forall y_3 \ldots \exists y_n P(x, y_1, y_2, y_3 \ldots y_n)\}$ or $\{x : \forall y_1 \exists y_2 \forall y_3 \ldots, \forall y_n P(x, y_1, y_2, y_3 \ldots y_n)\}$ depending on whether n is even or odd, where P is a decidable predicate.

Finally, Δ_n^0 is defined as $\Sigma_n^0 \cap \Pi_n^0$.

The classes Σ_0^0 and Π_0^0 are both defined as sets of the form $\{x : P(x)\}$ where P is decidable. Δ_0^0 and Δ_1^0 turn out to be equal to this as well.

Remark. *Some books (e.g., [HR67]) define the arithmetical hierarchy based on complements and projections. Projection is similar to existential quantification and can be combined with complement to form universal quantification.*

Note that this hierarchy is only concerned with alternation of quantifiers. Several consecutive instances of the same quantifier are always equivalent to just a single quantifier because it is possible to encode a vector as a single number and quantify over that to simulate quantifying over every element of the vector.

The classes we discussed before, RE and REC, both have their place in the arithmetical hierarchy. REC is in definition equal to Σ_0^0 and Π_0^0. The position of RE is less obvious.

Theorem 2. *RE is the class Σ_1^0 in the arithmetical hierarchy.*

Proof. We will first show that $RE \subseteq \Sigma_1^0$ and then that $\Sigma_1^0 \subseteq RE$.

A language L is in RE iff there is a Turing machine T that accepts it. A word w is accepted by T iff there is an n such that T reaches the accepting state after applying the step relation n times. The step relation is decidable so this is a formula in Σ_1^0.

Given any set in Σ_1^0, it can be written as $\{x : \exists y P(x,y)\}$ where P is a decidable predicate. Because P is decidable, there is a total Turing machine that accepts $\{x, y : P(x,y)\}$. Construct a new Turing machine T' that initialises y to 0 and calls T. If T accepts, so does T'. If T rejects, T' increments y and tries again. It keeps trying like this until it accepts. T' accepts an input $\{x : \exists y P(x,y)\}$, which is the language we want. □

Remark. *The non-determinism of non-deterministic Turing machines gives rise to another form of existential quantification. This does not increase their rank in the arithmetical hierarchy because the new quantifier can be combined with the existing one into a single existential quantifier.*

We will now describe the structure of the arithmetical hierarchy. We will show that its classes are partially ordered and we will show how incomparable classes are related to each other.

Theorem 3. *For any n, $\Sigma_n^0 \cup \Pi_n^0 \subseteq \Delta_{n+1}^0$.*

Proof. Given any set $S \in \Sigma_n^0$, given by the construction $\exists y_1 \forall y_2 \dots \forall y_n P(x, y_1, y_2, \dots, y_n)$ or $\exists y_1 \forall y_2 \dots \exists y_n P(x, y_1, y_2, \dots, y_n)$ where P is decidable. Let $Q(x, y_1, y_2, \dots, y_n, z) \equiv P(x, y_1, y_2, \dots, y_n) \wedge z = z$. Q is clearly still decidable.

Equivalent definitions of S are given by $\forall z \exists y_1 \forall y_2 \dots \forall y_n Q(x, y_1, y_2, \dots, y_n, z)$ or $\forall z \exists y_1 \forall y_2 \dots \exists y_n Q(x, y_1, y_2, \dots, y_n, z)$ which are of the form Π_{n+1}^0 and by $\exists y_1 \forall y_2 \dots \forall y_n \exists z Q(x, y_1, y_2, \dots, y_n, z)$ or $\exists y_1 \forall y_2 \dots \exists y_n \forall z Q(x, y_1, y_2, \dots, y_n, z)$ which are of the form Σ_{n+1}^0. We can conclude that $\Sigma_n^0 \subseteq \Delta_{n+1}^0$.

The same method can be used to add extra quantifiers to Π_n^0 and prove that $\Pi_n^0 \subseteq \Delta_{n+1}^0$. Therefore, we can conclude that $\Sigma_n^0 \cup \Pi_n^0 \subseteq \Delta_{n+1}^0$. □

Theorem 4. *For any n, Π_n^0 consists of the complements of the sets in Σ_n^0 and vice versa.*

Proof. Given any set $S \in \Sigma_n^0$, the complement of S can be written in the form $\{x | \neg \exists y_1 \forall y_2 \dots P(x, y_1, y_2, \dots)\}$. We repeatedly apply De Morgan's laws to get a formula of the form $\{x | \forall y_1 \exists y_2 \dots \neg P(x, y_1, y_2, \dots)\}$. Decidable predicates are closed under complement, so $\neg P$ is decidable. We got a formula in Π_n^0. The same can be done to get from the complement of a set in Π_n^0 to a set in Σ_n^0. □

Theorem 5. *For any n, Δ_n^0 is closed under complementation.*

Proof. Let any set $S \in \Delta_n^0$ be given. Since $\Delta_n^0 = \Sigma_n^0 \cap \Pi_n^0$, S is both in Σ_n^0 and in Π_n^0. By Theorem 4, the complement of S is also both in Σ_n^0 and in Π_n^0. Therefore, the complement of S is in Δ_n^0. □

A diagram of the hierarchy along with the position of some classes and problems in it is shown in Figure 4.1.

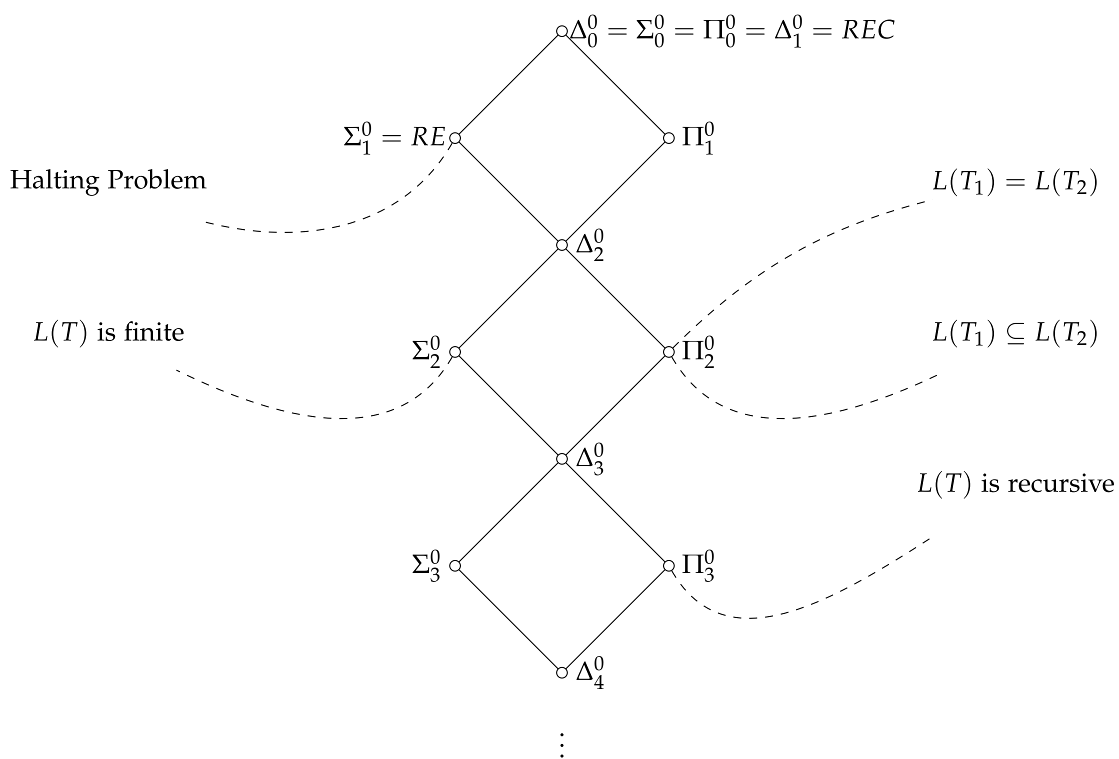

Figure 4.1: The arithmetical hierarchy with the position of some classes and decision problems indicated (See [Soa16]).

Chapter 5

Properties of Tape-quantifying Turing Machines

We will now derive a number of properties of the class of languages accepted by tape-quantifying Turing machines. We will show where the class fits into the arithmetical hierarchy. We also give constructions for the union and intersection of languages.

5.1 Position in the Arithmetical Hierarchy

We start by deriving the position of tape-quantifying Turing machines in the arithmetical hierarchy.

Theorem 6. *The class of languages accepted by tape-quantifying Turing machines is* Π^0_2.

Proof. Call the class of languages accepted by tape-quantifying Turing machines O. We will first show that $O \subseteq \Pi^0_2$ and then that $\Pi^0_2 \subseteq O$.

Given $L \in O$, there is a tape-quantifying Turing machine that accepts L. Call one such machine T.
A word w is in L iff for all c, the count instance with count c accepts iff for all c, there is an n such that the count instance with count c reaches the accepting state in n steps. The step relation is decidable so this is a formula in Π^0_2.

Conversely, Given a set S in Π^0_2, S can be written as $\{x : \forall y_1 \exists y_2 P(x, y_1, y_2)\}$ where P is decidable. Since P is decidable, there is a total Turing machine T that accepts $(x, y_1, y_2) : P(x, y_1, y_2)$. Without loss of generality we assume T always has an empty tape when it accepts or rejects. A tape-quantifying Turing machine T' that accepts S can now be constructed as in Figure 5.1.

Each count instance of this tape-quantifying Turing machine moves the input to the second tape to the left of the count in state q_1. There is also a number to the right of the count, but it is initialised to 0 in unary (the

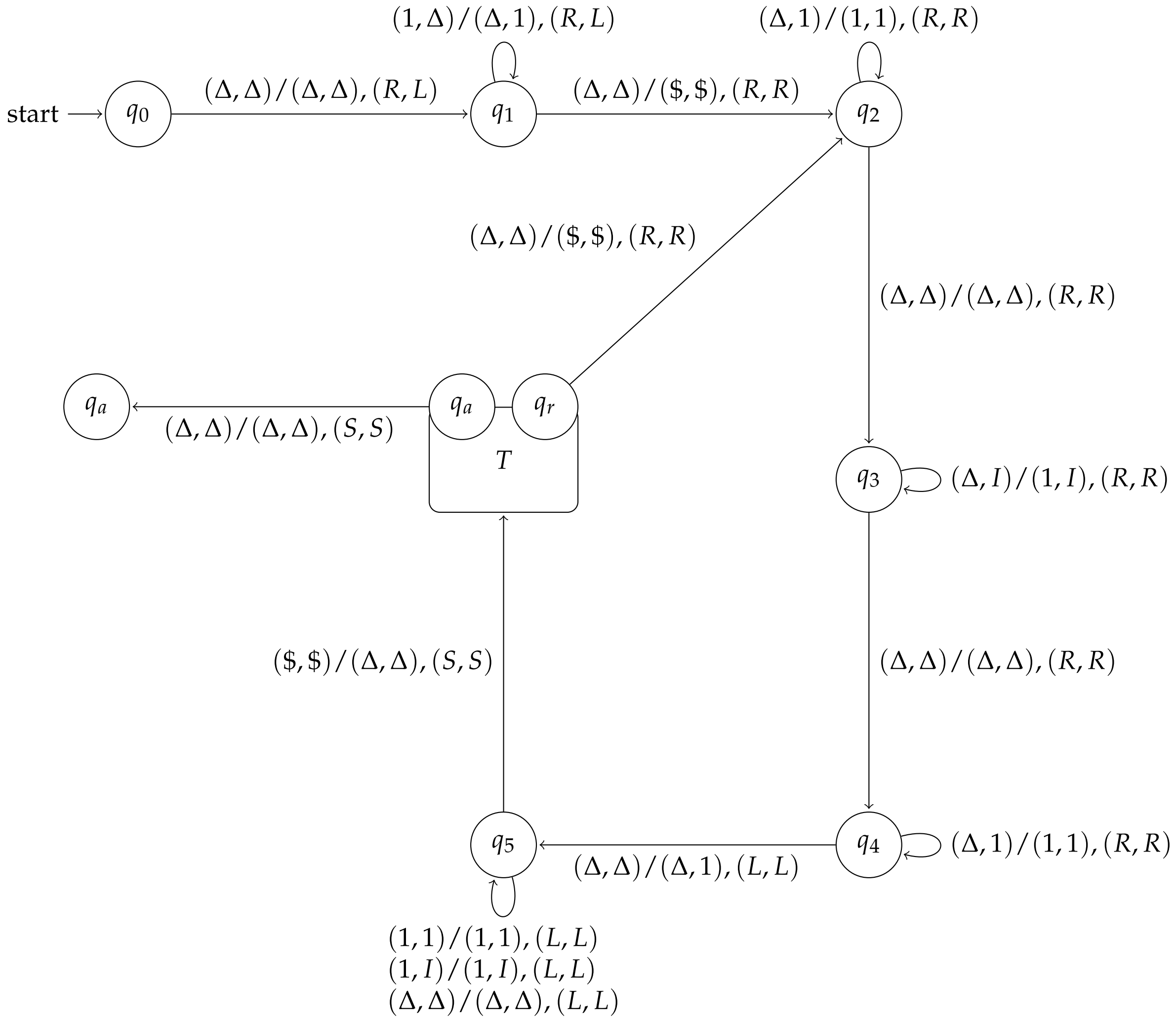

Figure 5.1: Construction of a tape-quantifying Turing machine for an arbitrary set in Π_2^0. The block T represents a total Turing machine deciding the predicate P with its accepting and rejecting states interpreted as normal states.

empty string), so nothing has to be done to put it there. It then copies the second tape to the first tape using states q_2, q_3 and q_4. It then increments the number to the right of the count and moves back to the left using q_5. Next, it simulates T on the first tape. If T accepts, so does the constructed machine. Otherwise, it goes back to q_2 to try again, but this time, the number to the right of the count has been incremented. The count is used to quantify over y_1 and the third number to quantify over y_2. Each count instance accepts iff there is an y_2 which makes T accept. The tape-quantifying Turing machine accepts input x iff $\forall y_1 \exists y_2 P(x, y_1, y_2)$, which is what we want. □

5.2 Closure Properties

The class Π_2^0 is known to be closed under finite union and intersection. Nevertheless, it is good to have constructions for this in the model itself. The class is not closed under complement. The complement of the languages in Π_2^0 form Σ_2^0, which is a different class.

Intersection Given two tape-quantifying Turing machines T_1 and T_2, construct a new tape-quantifying Turing machine as in Figure 5.2.

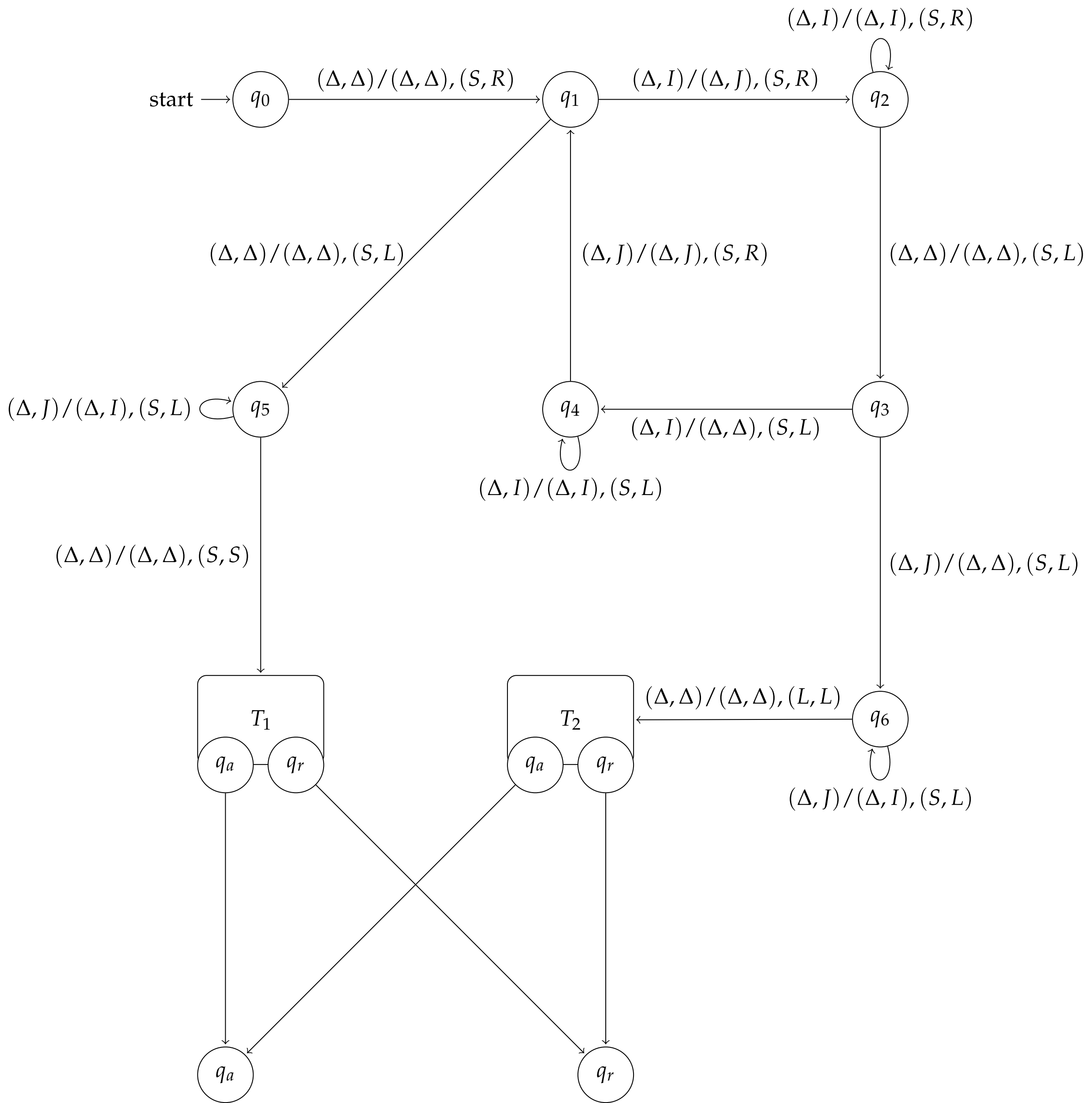

Figure 5.2: Construction of a tape-quantifying Turing machine that accepts the intersection of the languages of two other tape-quantifying Turing machines. The blocks T_1 and T_2 represent tape-quantifying Turing machines accepting the two languages of which we make the intersection. Their accepting and rejecting states are interpreted as normal states.

Each count instance of the this construction divides its count by 2 using states q_1 to q_4. If there is no remainder, it calls moves the tape head back to the initial position using q_5 and calls T_1 with the result of the division. Otherwise, it moves the tape head back using q_6 and calls T_2. This ensures that all count instances of both machines are evaluated. By the definition, the resulting machine accepts iff all of those calls reach the accepting state. Therefore, it accepts the intersection of the languages of T_1 and T_2.

Union The union is a lot more difficult. Given two tape-quantifying Turing machines T_1 and T_2, each count instance has to simulate both machines with the same count in parallel by alternately simulating a step of one and the other (dovetailing). If one of the two accepts, that machine will be simulated with all smaller counts as well. If they all accept, that count instance of the constructed machine accepts as well. In the meantime the other machine keeps being simulated as well. If that machine also accepts, it will also be checked with smaller counts in the same way.

The constructed tape-quantifying Turing machine itself accepts an input iff for all n, all count instances of one of the machines up to n accept, which is equivalent to saying that all count instances of one of the machines accept. Therefore, the resulting tape-quantifying Turing machine accepts iff at least one of the two accepts and thus accepts the union of the two languages.

Chapter 6

Discussion and Conclusions

In the previous chapters, we defined and analysed a new variation on Turing machines called tape-quantifying Turing machines. We demonstrated that they are more powerful than normal Turing machines. The class of languages they can accept turned out to be Π_2^0, while normal Turing machines can accept Σ_1^0.

This new type of machines may offer a more intuitive way to understand the class Π_2^0 from a computer science perspective, as it is a more algorithmic approach than typically used when describing the arithmetical hierarchy. This may also allow tools developed for the analysis of Turing machines to be used for the analysis of this class.

It would be useful if the definition of tape-quantifying Turing machines could be generalised to allow analysis of other classes in the arithmetical hierarchy. An intuitive approach would be to quantify over initialisations with other strings on the second tape. This can only change the class of languages it can accept if there is no computable bijection between the natural numbers and the set of initialisations.

One interesting possibility appears to be initialising the second tape to an infinite string over an alphabet of two or more symbols. There are uncountably many of those, so there is no bijection, let alone a computable bijection, with the natural numbers. It seems like this would transcend the arithmetical hierarchy into a higher hierarchy called the analytical hierarchy. We do not know if there are elegant options to generalise to get classes in between like the higher classes of the arithmetical hierarchy. Further investigation may shed more light on this.

Bibliography

[HR67] Jr. Hartley Rogers. *Theory of Recursive Functions and Effective computatability*. McGraw-Hill Book Company, 1967.

[Rob15] Borut Robič. *The Foundations of Computability Theory*. Springer, 2015.

[Soa16] Robert I. Soare. *Turing Computability. Theory and Applications*. Springer, 2016.