A Constructive Approach Towards Formalizing Relativization Using Combinatory Logic

MSc Thesis *(Afstudeerscriptie)*

written by

Marlou M. Gijzen
(born May 9th, 1994 in Haarlem, The Netherlands)

under the supervision of **Dr Benno van den Berg** and **Dr Leen Torenvliet**, and submitted to the Board of Examiners in partial fulfillment of the requirements for the degree of

MSc in Logic

at the *Universiteit van Amsterdam.*

Date of the public defense:	**Members of the Thesis Committee:**
September 12, 2018	Dr Benno van den Berg
	Dr Ronald de Haan
	Frederik Lauridsen MSc
	Dr Floris Roelofsen (Chair)
	Dr Leen Torenvliet
	Prof Dr Rineke Verbrugge

Institute for Logic, Language and Computation

Abstract

We present a formal system, ACT, that is used to express the time-complexity of computing functions. Every proof that can be made within the system relativizes. The system uses combinatory logic instead of Turing machines. We do show that it is invariant with respect to Turing machines.

We obtained that ACT is conservative over HA. We also present an extension to the system that allows for the usage of diagonalization. We then show that the P vs NP problem is independent from this system.

Contents

1 Introduction

In this work, we present a system that can be used to express the time-complexity of computations. Furthermore, every proof in the system *relativizes*.

The notion of relativization plays a mayor role in computational complexity. We say that a proof relativizes when it also holds relative to any oracle. Oracle machines are used to define Turing reductions. They have also been used to obtain a series of results that tell us which proof-techniques we can and cannot use to solve certain problems. We know for example that we need non-relativizing proof-techniques to solve the **P** vs **NP** problem.

However, it has never been proven that a proof-technique does not relativize, even though that has been said about, for example, *arithmetization*.

With this system we formalize the notion of relativization. The axiomatic system is called Absolute complexity theory, ACT, and it is an extension of Heyting arithmetic, HA. It uses combinatory logic as a model for computation. The length of the reduction serves as a benchmark for measuring the time-complexity of solving specific problems.

We expect that this system could be used to gain insight in non-relativizing proof-techniques. This information is necessary in order to obtain certainty about the usefulness of proof-methods for tackling specific problems. We also hope to be able to obtain independence of the **P** vs **NP** problem from general intuïtionistic logics with future work.

In Chapter 2 we outline some preliminary notions. In the following chapter the system ACT is presented.

Chapter 4 treats the conservativity of ACT over HA. In the next chapter we give several terms that compute functions such as addition and multiplication, and we prove the complexity of those computations.

We do not use Turing machines as a model for computation, in contrast to standard computational complexity. However, in Chapter 6 we do show that the system is invariant with respect to Turing machines: both models can simulate each other with only a polynomially-bounded overhead in time.

We are unable to define an encoding of terms within ACT. However, this encoding is needed to use certain proof-techniques. In Chapter 7 we extend the system with axioms that enable us to use diagonalization, which is regarded as a relativizing proof-technique. We show some known results from standard computational complexity and we obtain that the P vs NP problem is independent from the system.

2 Preliminaries

The formal system that will be presented in Chapter 3 can be used to describe the time-complexity of computations. It is based upon Heyting arithmetic. When we mention a *reduction* we mean a rewriting-step from one term into another. A term in *normal form* cannot be reduced to any other term.

In this chapter we will outline some notions from complexity theory that will be used for the formal system, as well as a description of Heyting arithmetic.

2.1 Basic notions from complexity theory

In computational complexity theory problems are categorized according to their difficulty. We try to measure the difficulty of problems by considering the amount of some resource that a Turing machine needs to solve a problem.

Definition 2.1.1 ([3, Sec. 1.2])**.** A *Turing machine* is described by a tuple (Γ, Q, δ):

- Γ is a finite set of tape symbols, the alphabet.
- Q is a finite set of states.
- δ is a transition function: $Q \times \Gamma \to Q \times \Gamma \times \{L, R, S\}$.

It has a one-way infinite read/write tape, a head that it uses for reading and writing on the cells of the tape and a register that holds the state of the machine. A single *computation step* is described by one application of the transition function: it reads a symbol in a specific state, and then writes down a symbol on that location, gets into some state and the head moves to the left, right or stays on that location.

A (regular) Turing machine works in a *deterministic* manner. We can also consider machines that work *nondeterministically*:

Definition 2.1.2 ([3, Sec. 2.1.2])**.** A *nondeterministic Turing machine* is a Turing machine with two transition functions. At each step, the machine chooses which of its transition functions it will apply. When dealing with 0/1-valued functions, we say that a nondeterministic machine accepts an input when one of its possible computational paths accepts (or outputs 1).

A Turing machine usually works with binary strings, and it thus computes functions $f : \{0,1\}^* \to \{0,1\}^*$. We will consider the amount of time needed for computations. In the case of Turing machines, this is measured as the number of steps that a machine takes before it halts. Since the exact number of steps is dependent on the size of the

input, we will refer to functions $T : \mathbb{N} \to \mathbb{N}$ to describe the computation times. $T(n)$ gives the number of steps that a machine needs in the worst case for a computation on an input of size n.

2.1.1 Some complexity classes

In standard computational complexity we usually consider the complexity of 0/1-valued functions or decision problems - i.e., $f : \{0,1\}^* \to \{0,1\}$.

The exact number of steps that are needed for a computation are not important, we are only interested in the biggest growing term of the function $T(n)$. We thus use the *Big-Oh notation*:

Definition 2.1.3. For f, $g : \mathbb{N} \to \mathbb{N}$, $f = O(g)$ when there exists a constant c and some $n_0 \in \mathbb{N}$ such that $f(n) \leq c \cdot g(n)$ for all $n \geq n_0$.

In this paper we will refer to three different complexity classes:

Definition 2.1.4 ([7])**.** The class **P** is the class of decision problems for which there exists a Turing machine that computes the problem and runs in time $O(p(n))$ for some polynomial p.

Definition 2.1.5 ([7])**.** The class **EXP** is the class of decision problems solvable by a Turing machine in time $O(2^{p(n)})$ for a polynomial p.

Definition 2.1.6 ([7])**.** The class **NP** consists of problems solvable in polynomial-time by a nondeterministic Turing machine. We say that the machine accepts when at least one computation path accepts, and it rejects when all computation paths do so.

Equivalently, a decision problem f is in **NP** when there are polynomials p and q such that there exists a Turing machine M that runs in time $O(p(n))$ and for every $x \in \{0,1\}^*$,

$$f(x) = 1 \Leftrightarrow \exists u \in \{0,1\}^{q(n)} \text{ s.t. } M \text{ accepts on input } (x,u)$$

2.1.2 Oracle machines

In addition to the classical notion of a Turing machine, we will also talk about Turing machines that have black-box access to some function:

Definition 2.1.7 ([10, p. 43])**.** An *oracle Turing machine* is a Turing machine with access to some oracle O. The machine has an extra one-way read/write tape, the oracle tape, and a corresponding head. The oracle O represents some function f. The machine also has two additional states: s_q, the query state, and s_a, the answer state. When the machine writes an input x on the oracle tape, and when it gets into the state s_q, in one step the contents of the oracle tape are replaced by the output $f(x)$, and the machine will go into the state s_a.

We will refer to computations performed by such machines with oracle access to an oracle O as *computations relative to* O. We can also consider complexity classes relative to oracles. The class $\mathbf{P}^O$ denotes the class of problems solvable in polynomial time by an oracle Turing machine with access to O.

In standard computational complexity the oracles that are used are generally also decision problems (or sets). We can use any function as an oracle, but if a machine runs within a specific time-bound, then this puts a limit on the number of bits that the machine can read from the output of the oracle.

2.1.3 Relativization and diagonalization

We call proof-techniques in complexity theory *relativizing* when the result also holds relative to any oracle. For example, the *Time Hierarchy Theorem* [8] is shown using *diagonalization*, which is regarded as a relativizing proof-technique. The theorem has as a result that $\mathbf{P} \neq \mathbf{EXP}$, so this means that we also have that $\mathbf{P}^O \neq \mathbf{EXP}^O$ for any oracle O.

Cantor was the first to use diagonalization in proofs and it can similarly be used in complexity theory.

For diagonalization in standard computational complexity, we need an enumeration of Turing machines and a way to (efficiently) simulate the machines.

For the enumeration, we use some fixed way of describing the Turing machines as binary strings. How this can be done is described in [3, Sec. 1.4]. We make use of two conventions. Firstly, that we can describe every Turing machine with a binary string and every string represents some Turing machine. Secondly, that there is an infinite number of binary strings that represents one Turing machine.

Generally, we use a *universal Turing machine* to simulate Turing machines based on their description:

Theorem 2.1.1 ([3, Th. 1.9])**.** *There exists a Turing machine U such that for every two inputs x, $\alpha \in \{0,1\}^*$, $U(x,\alpha) = M_\alpha(x)$, where M_α denotes the machine represented by the string α. If M_α halts within T steps on input x, then U halts within $CT \log T$ steps on inputs x and α (with C a number independent of x).*

A typical diagonalization proof then goes as follows. For an input $x \in \{0,1\}^n$, we run $U(x,x)$ for $g(n)$ steps, where g is some function. We can do this by adding a counter for the number of steps to the machine. If the machine halted by then, output the opposite. We then have a function that no machine that runs within time $O(\sqrt{g(n)})$ can compute.

Since in most proofs we could just as well diagonalize against oracle Turing machines, diagonalization is believed to be a relativizing proof-technique.

The notion of relativization gives us insight in the proof techniques that we need to solve certain problems. We know that we need non-relativizing proof techniques to solve the $\mathbf{P}$ vs $\mathbf{NP}$ problem, since it has *conflicting relativizations*:

Theorem 2.1.2 (Baker, Gill, Solovay, 1975, [5])**.** *There exist oracles A, B such that $\mathbf{P}^A = \mathbf{NP}^A$ and $\mathbf{P}^B \neq \mathbf{NP}^B$.*

2.2 Heyting arithmetic

Heyting arithmetic, HA, is a formal system that describes intuïtionistic first order arithmetic. It uses first-order intuïtionistic predicate logic with equality.

It has a constant 0 (zero), a unary function constant S (the successor), and function symbols for all primitive recursive functions.

We have the axioms from the first-order predicate logic, axioms for all the primitive recursive functions, and the following:

$$\neg S0 = 0$$
$$Sx = Sy \Rightarrow x = y$$
$$A(0) \wedge \forall x[A(x) \rightarrow A(Sx)] \Rightarrow \forall x A(x)$$

In HA every element is a natural number. We can denote the number n by $S^n 0$, successive application of S. We have that $\bot$ can be defined as $0 = S0$ [12, Sec. 3.3].

3 Absolute Complexity Theory

We will introduce a formal system called ACT: Absolute complexity theory. It is based upon the system TAPP, defined by Troelstra in [12, p. 475]. ACT offers a way to describe the complexity of solving partial functions from $\mathbb{N}$ to $\mathbb{N}$, using combinatory logic.

We will write

$$x \to_n y$$

when a program x can be rewritten into y with n time-steps. In the case of computation relative to an oracle f we write $x \to_n^f y$.

3.1 The formal system ACT

The system uses first order intuïtionistic predicate logic with equality ('=').

There is a unary predicate N, corresponding to being a natural number:

$$\forall x[N(x) \Leftrightarrow x \in \mathbb{N}]$$

We have a constant 0, the successor function S, and function symbols for all primitive recursive functions.

We also adopt the following axioms:

H.1 $0 \in \mathbb{N}$

H.2 $x \in \mathbb{N} \Rightarrow Sx \in \mathbb{N}$

H.3 $\forall x[\neg Sx = 0]$

H.4 $\forall x,\, y \in \mathbb{N}[Sx = Sy \Rightarrow x = y]$

H.5 $\varphi(0) \wedge \forall x \in \mathbb{N}[\varphi(x) \Rightarrow \varphi(Sx)] \Rightarrow \forall x \in \mathbb{N}\varphi(x)$

We also add the axioms for the primitive recursive function symbols, only we add the predicate N to the axioms such that the variables range over the natural numbers.

With respect to equality we have the following axioms:

E.1 $\forall x[x = x]$

E.2 $\forall x, y[x = y \Rightarrow y = x]$

E.3 $\forall x, y, z[x = y \wedge y = z \Rightarrow x = z]$

E.4 $\forall x, y[x = y \wedge \varphi(x) \Rightarrow \varphi(y)]$

Besides this, we have a total binary operation denoted by a dot '$\cdot$', which stands for application of terms. We assume left associativity and often leave out the '$\cdot$': when we write xyz, we mean $(x \cdot y) \cdot z$.

There is also a ternary predicate, the computation, described with '$\to$'.

We have a several combinators: $\mathbf{k}$, $\mathbf{s}$, $\mathbf{p}$, $\mathbf{p}_0$, $\mathbf{p}_1$, **succ**, **pred** and $\mathbf{d}$. The $\mathbf{k}$ and $\mathbf{s}$ are well-known from combinatory logic. The $\mathbf{p}$ is used for paring and $\mathbf{p}_0$ and $\mathbf{p}_1$ for projections. We have **succ** and **pred** as successor and predecessor combinators. Finally, the $\mathbf{d}$-combinator represents an if-then-else-construct. We thus have the following basic axioms for the combinators:

B.1 $\forall x, y[\mathbf{k}xy \to_1 x]$

B.2 $\forall x, y, z[\mathbf{s}xyz \to_1 xz(yz)]$

B.3 $\forall x, y[\mathbf{p}_0(\mathbf{p}xy) \to_1 x]$

B.4 $\forall x, y[\mathbf{p}_1(\mathbf{p}xy) \to_1 y]$

B.5 $\forall x, y[\mathbf{p}(\mathbf{p}_0 x)(\mathbf{p}_1 x) \to_1 x]$

B.6 $\forall x[\mathbf{succ}x \to_1 Sx]$

B.7 $\forall x[\mathbf{pred}Sx \to_1 x]$

B.8 $\mathbf{pred}0 \to_1 0$

B.9 $\forall x, y, z[\mathbf{d}(Sx)yz \to_1 y]$

B.10 $\forall y, z[\mathbf{d}0yz \to_1 z]$

We have the following axioms that we use to rewrite terms:

A.1 $\forall x, y, z[x \to_z y \Rightarrow z \in \mathbb{N}]$

A.2 $\forall x, y, n[x \to_n y \Rightarrow \forall m \geq n[x \to_m y]]$

A.3 $\forall x, y, z, n, m[x \to_n y \wedge y \to_m z \Rightarrow x \to_{n+m} z]$

A.4 $\forall x, y, z, n[x \to_n y \Rightarrow x \cdot z \to_n y \cdot z]$

A.5 $\forall x, y, z, n[x \to_n y \Rightarrow z \cdot x \to_n z \cdot y]$

A.6 $\forall x, n, y_1, y_2[x \to_n y_1 \wedge x \to_n y_2 \Rightarrow \exists z, m[y_1 \to_m z \wedge y_2 \to_m z]]$

A.7 $\forall x[N(x) \Rightarrow \forall y, n[x \to_n y \Rightarrow y = x]]$

Axiom A.1 ensure that the ternary predicate $\to$ has a natural number to denote the number of steps for rewriting. Axiom A.2 says that if we can deduce y from x within n steps, then it is also possible for any number of steps bigger than n. Axiom A.3 enables us to combine different reductions. The axioms A.4 and A.5 are needed for the ability to rewrite certain parts of a term at a time. Axiom A.6 expresses that the order of the reductions has no influence on the final result (as in the Church-Rosser Theorem). Axiom A.7 states that natural numbers are in normal form.

We also use the following abbreviations:

$x \to y := \exists n[x \to_n y]$

$x \doteq y := \forall n \in \mathbb{N} \exists z, m \in \mathbb{N}[x \cdot n \to_m z \wedge y \cdot n \to_m z]$

In Chapter 6, we will see that our system is Turing complete.

3.2 Relativized worlds

We will define another axiomatic system, extending ACT. For any total computable function $f : \mathbb{N} \to \mathbb{N}$, that is definable in ACT, we have a system ACT^f, where f can be used as an oracle for the computations.

Since f is computable, we thus have by assumption that there exists an term that computes the function and we will fix a canonically chosen term t_f.

Instead of using $\to$ as a symbol for the ternary predicate, we will now use $\to^f$ to denote computations relative to f.

The axioms of ACT^f are the same as those of ACT, except we use the symbol $\to^f$ everywhere. We also add one combinator and a corresponding axiom. Let $f(x)$ denote the output of f on input x. Then:

B.11 $\forall x \in \mathbb{N}[\mathbf{o}_f x \to_1^f S^{f(x)}0]$

We thus have that $\mathsf{ACT} \vdash \forall x \in \mathbb{N} \exists m \in \mathbb{N}[t_f x \to_m f(x)]$.

We have that every query to the oracle can be replaced by actually computing f, and the other way around. Thus, for any term t_f that computes f, and for any x in the domain of f, there exists an m such that $t_f \cdot x \to_m^f f(x) \Leftrightarrow \mathbf{o}_f \cdot x \to_1^f f(x)$.

Since all the other axioms of ACT^f are the same axioms as in ACT, we have the following:

Corollary 3.2.1. *For all φ, $\mathsf{ACT} \vdash \varphi \Rightarrow \mathsf{ACT}^f \vdash \varphi$*

Every statement in ACT has a relative variant in ACT^f for any total computable f. Put differently, anything that can be proven within ACT relativizes.

4 Conservativity over HA

We would like to show that the systems ACT and ACT^f for any f are conservative over HA.

Definition 4.0.1. A theory T' is a *conservative extension* of a theory T iff

$$T' \vdash \varphi \Rightarrow T \vdash \varphi$$

for all formulas φ in the language of T.

If we achieve this, we can conclude that ACT is consistent, given the consistency of HA. We also know that ACT says as much about the natural numbers as HA does.

However, there is one problem. In HA all free variables are natural numbers and we only quantify over natural numbers. This is not the case in ACT. Since we are only interested in formulas that are expressible in the language of HA, we will instead show the following:

$$\mathsf{ACT} \vdash \varphi^{\mathbb{N}} \Rightarrow \mathsf{HA} \vdash \varphi$$

for all formulas φ in the language of HA. The mapping $\cdot^{\mathbb{N}}$ takes formulas to similar formulas, but the free variables are natural numbers and quantifiers range over natural numbers.

This means that we will not achieve actual conservativity over HA. However, we will be able to conclude the things that we wanted to know about ACT (consistency and expressibility with respect to natural numbers).

We define $\cdot^{\mathbb{N}}$ formally as follows:

Definition 4.0.2.

$$\begin{aligned}
\phi(x_1, \ldots, x_n)^{\mathbb{N}} &:= N(x_1) \wedge \ldots \wedge N(x_n) \wedge \phi(x_1, \ldots, x_n) \text{ for atomic formulas } \phi \\
(\phi * \psi)^{\mathbb{N}} &:= \phi^{\mathbb{N}} * \psi^{\mathbb{N}} \text{ for } * \in \{\vee, \wedge, \rightarrow\} \\
(\forall x \phi)^{\mathbb{N}} &:= \forall x \in \mathbb{N} \phi^{\mathbb{N}} \\
(\exists x \phi)^{\mathbb{N}} &:= \exists x \in \mathbb{N} \phi^{\mathbb{N}}
\end{aligned}$$

In the rest of this chapter we will focus on a mapping $[\![\cdot]\!]$ from formulas of the language of ACT to formulas of the language of HA, which serves as a way to interpret ACT in HA. We will then show

$$\mathsf{ACT} \vdash \varphi \Rightarrow \mathsf{HA} \vdash [\![\varphi]\!].$$

After proving that

$$\mathsf{HA} \vdash (\llbracket \varphi^{\mathbb{N}} \rrbracket \Leftrightarrow \varphi)$$

for formulas φ in the language of HA, we can conclude that

$$\mathsf{ACT} \vdash \varphi^{\mathbb{N}} \Rightarrow \mathsf{HA} \vdash \varphi$$

for all formulas φ in the language of HA. This is also how Troelstra suggests showing that TAPP is conservative over HA [12, p. 489]. The case for ACT^f is similar.

For the mapping $\llbracket \cdot \rrbracket$, we will focus us on $RE(\omega)$, recursively enumerable subsets of $\mathbb{N}$. In HA we have function symbols for the primitive recursive functions, so we can describe all elements of $RE(\omega)$ (the range of a primitive recursive function is recursively enumerable). Since *Kleene's T-predicate* is also primitive recursive, we can describe all elements of $RE(\omega)$ using only a single primitive recursive function symbol. This allows us to quantify over the elements of $RE(\omega)$.

Kleene's T-predicate relies on a Gödel numbering that assigns functions to natural numbers. We then have that $T(n_x, m, z)$ is true when z encodes a valid and halting computation of the computable function with index n_x on input m. A set X in $RE(\omega)$ can be described as $\{m \mid \exists z(T(n_x, m, z))\}$ or $\{m \mid \varphi(n_x, m)\}$.

4.1 Coding sets and pairs

First, we have to fix a coding p for pairs. We will choose the well-known function $p(x, y) = \frac{1}{2}(x + y)(x + y + 1) + y$, but we will write (x, y) as an abbreviation. We will also use a coding for finite sets:

$\{k_0, \ldots, k_p\} := \sum_{i=0}^{p} 2^{k_i}$

$\{\emptyset\} := 0$

Note that 2^n codes $\{n\}$.

4.2 Defining the mapping

As in [12, p. 484], we can define a total application operation $\cdot$ by:

$X \cdot Y := \{z \mid \exists y \subset Y((y, z) \in X)\}$. When we say $y \subset Y$ then we mean that the set that is coded by y is a subset of Y. We let $\llbracket X \cdot Y \rrbracket := \llbracket X \rrbracket \cdot \llbracket Y \rrbracket$.

The application of two recursively enumerable sets is again recursively enumerable. This can be seen as follows: for an input z, we enumerate the elements of both Y and X step-by-step. With every new element of Y in the enumeration, for all pairs (y, z) in the enumeration of X, we check whether y encodes a subset of the already listed elements of Y. If yes, we accept z.

Variables are mapped to themselves. For the constants that are also in the language of HA we do the following:

$$\begin{aligned} \llbracket 0 \rrbracket &:= \{0\} \\ \llbracket S(n) \rrbracket &:= \{n + 1\} \\ \llbracket F(x_1, \ldots, x_n) \rrbracket &:= \{F(x_1, \ldots, x_n)\} \end{aligned}$$

Also as defined in [12] we have the following definitions for the combinators:

$$\begin{aligned}
[\![\mathbf{k}]\!] &:= \{(x,(y,z)) \mid z \in x\} \\
[\![\mathbf{s}]\!] &:= \{(x,(y,(z,w))) \mid z' \subset z((z',(u,w)) \in x) \wedge \forall u' \in u \exists z' \subset z((z',u') \in y)\} \\
[\![\mathbf{succ}]\!] &:= \{(2^n, n+1) \mid n \in \mathbb{N}\} \\
[\![\mathbf{pred}]\!] &:= \{(2^n, n-1) \mid n \in \mathbb{N}\} \\
[\![\mathbf{p}]\!] &:= \{(2^n,(0,2n)) \mid n \in \mathbb{N}\} \cup \{(0,(2^m,2m+1)) \mid m \in \mathbb{N}\} \\
[\![\mathbf{p}_0]\!] &:= \{(2^{2n}, n) \mid n \in \mathbb{N}\} \\
[\![\mathbf{p}_1]\!] &:= \{(2^{2n+1}, n) \mid n \in \mathbb{N}\} \\
[\![\mathbf{d}]\!] &:= \{(2^n,(x,(y,z))) \mid n = 0 \wedge z \in y \vee \neg n = 0 \wedge z \in x\}
\end{aligned}$$

We also have the following predicates:

$$\begin{aligned}
[\![X = Y]\!] &:= [\![X]\!] = [\![Y]\!] \\
[\![X \to_n Y]\!] &:= [\![X]\!] = [\![Y]\!] \wedge [\![N(n)]\!] \\
[\![N(X)]\!] &:= \exists n[2^n = X]
\end{aligned}$$

For $\vee$, $\wedge$, $\Rightarrow$, $\forall x$ and $\exists y$ we let $[\![\cdot]\!]$ act as a homomorphism.

4.3 Conservativity of ACT

Lemma 4.3.1. *For all formulas φ, $\mathsf{ACT} \vdash \varphi \Rightarrow \mathsf{HA} \vdash [\![\varphi]\!]$*

Proof. We prove this by showing that all axioms of ACT (that are not already in HA) hold in HA under the mapping.

From the basic axioms, we will show that $([\![\mathbf{k}]\!] \cdot X) \cdot Y = X$, and leave the verification of the other basic axioms to the reader.

$$\begin{aligned}
([\![\mathbf{k}]\!] \cdot X) \cdot Y &= \{a \mid \exists x \subset X((x,a) \in [\![\mathbf{k}]\!])\} \cdot Y \\
&= \{b \mid \exists y \subset Y((y,b) \in \{a \mid \exists x \subset X((x,a) \in [\![\mathbf{k}]\!])\})\} \\
&= \{b \mid \exists y \subset Y \wedge \exists x \subset X((x,(y,b)) \in [\![\mathbf{k}]\!])\} \\
&= X
\end{aligned}$$

Now for the other axioms:

A.1 $\forall x,y,z[\![x \to_z y \Rightarrow N(z)]\!] = \forall X,Y,Z[X = Y \wedge \exists n(2^n = Z) \Rightarrow \exists n(2^n = Z)]$

Which always holds.

A.2 $\forall x,y,n[\![x \to_n y \Rightarrow \forall m \geq n[x \to_m y]]\!]$

$= \forall X,Y,n[X = Y \wedge \exists z(2^z = n) \Rightarrow [\![\forall m \leq n]\!][X = Y \wedge \exists z(2^z = m)]]$

This should hold, because of how '$n \leq m$' is defined in HA. In the following we will leave out the "$\exists m(2^m = X)$ ".

A.3 $\forall x, y, z, n, m[\![x \to_n y \wedge y \to_m z \Rightarrow x \to_{n+m} z]\!] = \forall X, Y, Z[X = Y \wedge Y = Z \Rightarrow X = Z]$

This is obviously true. The last two axioms also follow quite easily:

A.4 $\forall x, y, z, n[\![x \to_n y \Rightarrow x \cdot z \to_n y \cdot z]\!] = \forall X, Y, Z[X = Y \Rightarrow X \cdot Z = Y \cdot Z]$

Assume that X = Y. Then:

$$\begin{aligned} X \cdot Z &= \{a \mid \exists z \subset Z((z, a) \in X)\} \\ &= \{a \mid \exists z \subset Z((z, a) \in Y)\} \\ &= Y \cdot Z \end{aligned}$$

A.5 $\forall x, y, z, n[\![x \to_n y \Rightarrow z \cdot x \to_n z \cdot y]\!] = \forall X, Y, Z[X = Y \Rightarrow Z \cdot X = Z \cdot Y]$

Assume that X = Y. Then:

$$\begin{aligned} Z \cdot X &= \{a \mid \exists x \subset X((x, a) \in Z)\} \\ &= \{a \mid \exists y \subset Y((y, a) \in Z)\} \\ &= Z \cdot Y \end{aligned}$$

□

Theorem 4.3.2. $\mathsf{ACT} \vdash \varphi^{\mathbb{N}} \Rightarrow \mathsf{HA} \vdash \varphi$ *for all formulas* φ *in the language of* HA

Proof. We will show that for formulas φ in the language of HA: $\mathsf{HA} \vdash (\varphi \Leftrightarrow [\![\varphi^{\mathbb{N}}]\!])$. Then, with Lemma 4.3.1 the theorem follows.

We will show this by induction on the formulas. First, we consider atomic formulas.

We have that $\mathsf{HA} \vdash [\![a = b]\!]$ is equal to $\mathsf{HA} \vdash [\![a]\!] = [\![b]\!]$, where a and b could be of the form 0, $S(n)$ or $F(x_1, \ldots, x_n)$. Looking at the definition of $[\![\cdot]\!]$ for these (functional) constants, we see that $\mathsf{HA} \vdash [\![a = b]\!] \Leftrightarrow a = b$, since the left side is equality of singletons that are equal to a and b itself.

For most logical connectives, $[\![\cdot]\!]$ and $\cdot^{\mathbb{N}}$ act as a homomorphism. The only noticeable cases are the quantifiers, since then $\cdot^{\mathbb{N}}$ acts differently.

The axioms concerning the primitive recursive function symbols and the successor symbol were modified in ACT such that they quantified over natural numbers only. In HA every variable is a natural number, and thus it quantifies solely over the natural numbers. So we can conclude that for all φ, $\mathsf{HA} \vdash (\varphi \Leftrightarrow [\![\varphi^{\mathbb{N}}]\!])$. □

Corollary 4.3.1. $\mathsf{ACT}^f \vdash \varphi^{\mathbb{N}} \Rightarrow \mathsf{HA} \vdash \varphi$ *for all formulas* φ *in the language of* HA *and total recursive* f *that are definable in* ACT.

Proof. Let t_f be the chosen term that computes the function f. We let $[\![\mathbf{o}_f]\!] := [\![t_f]\!]$. In ACT we should have a proof of $\forall x \in \mathbb{N} \exists m \in \mathbb{N}[t_f \cdot x \to_m f(x)]$. Then with Lemma 4.3.1 and the proof of Theorem 4.3.2 we can can prove the corollary. □

5 Functions and Lists

In this chapter, we will define several functions and discuss the complexity of solving them. We will also outline how we can use lists in ACT. We will restrict our attention to lists of natural numbers and functions from $\mathbb{N}$ to $\mathbb{N}$. In order to make the definitions of functions easier, we will explain how we can introduce lambda abstractions such as $\lambda x.t(x)$.

5.1 The length of a term

Before we can start to work with lists and the complexity of solving lambda abstractions, we will have to define the length of a term. We will avoid terms that contain primitive recursive function symbols. As we will see in Chapter 6, for each primitive recursive function there exists a term that computes that function using only the combinators and the binary application symbol.

Definition 5.1.1 (The length of a term)**.**

$$\begin{aligned} \ell(0) &= 1 \\ \text{For all } x,\ \ell(Sx) &= 1 + \ell(x) \\ \text{For all combinators } \mathbf{c},\ \ell(\mathbf{c}) &= 1 \\ \text{For variables } x,\ \ell(x) &= 1 \\ \ell(t_1 \cdot t_2) &= 1 + \ell(t_1) + \ell(t_2) \end{aligned}$$

As a result, we have that for $n \in \mathbb{N}$, $\ell(n) = n + 1$.

5.2 Lambda abstractions

We can rewrite terms such that they behave similarly to reducing lambda abstractions:

Proposition 5.2.1. *For each expression $t(x)$ with a variable x, there exists a term $\lambda x.t(x)$ such that for each term b in the language of* ACT*, $\mathsf{ACT} \vdash \lambda x.t(x) \cdot b = t(b)$, where $t(b)$ is a term in the language of* ACT *with b in place of the variable x.*

For example, the term $\lambda x.\mathbf{d}(x)(1)(0)$ will return '1' when a b applied to it is a natural number bigger than zero, and it will return '0' when $b = 0$.

We have that $\ell(\mathbf{d}(x)(1)(0)) = 8$.

Proof of Proposition 5.2.1. We will built $\lambda x.t(x)$ with induction on the structure of $t(x)$, similarly to for example [12, Prop. 9.3.5]. If $t(x)$ is x itself, then $\lambda x.t(x) = \mathbf{skk}$. If $t(x)$ is a variable or constant c different from x, then $\lambda x.t(x) = \mathbf{k}c$. If $t = t_1(x) \cdot t_2(x)$, then $\lambda x.t(x) = \mathbf{s}(\lambda x.t_1(x))(\lambda x.t_2(x))$. □

5.2.1 Complexity of rewriting lambda abstractions

For all terms $\lambda x.t(x)$ and b, we can calculate the time it takes to rewrite $\lambda x.t(x) \cdot b$ into $t(b)$. This does not take into account the time to rewrite $t(b)$ into a term in normal form. We will eventually prove the following theorem:

Theorem 5.2.1. *For all $t(x_1, \ldots, x_n)$ there exists a $k \in \mathbb{N}$ such that for all $b_1, \ldots, b_n$ $\mathsf{ACT} \vdash \lambda x_1, \ldots x_n.t(x_1, \ldots, x_n) \cdot b_1 \cdot \ldots \cdot b_n \to_k t(b_1, \ldots, b_n)$.*

In order to make general statements about the complexity of rewriting terms with a lambda abstraction, we have to say something about the length of those abstractions. We will prove the following lemma:

Lemma 5.2.2. $\ell(\lambda x.t(x)) \leq 5\ell(t(x))$

Proof. We will show this by induction on the structure of $t(x)$. When $t(x)$ is x itself, then $\ell(\lambda x.t(x)) = 5$. When $t(x)$ is a variable or constant c different from x, then $\ell(\lambda x.t(x)) = 2 + \ell(c)$.

If $t = t_1(x) \cdot t_2(x)$, then $\ell(\lambda x.t(x)) \leq 3 + \ell(\lambda x.t_1(x)) + \ell(\lambda x.t_2(x)) \overset{\text{I.H.}}{\leq} 3 + 5\ell(t_1(x)) + 5\ell(t_2(x)) \leq 5 + 5\ell(t_1(x)) + 5\ell(t_2(x)) = 5(1 + \ell(t_1(x) + \ell(t_2(x))) = 5\ell(t(x))$. □

We can now say something about the complexity of rewriting lambda expressions:

Lemma 5.2.3. *For all b, $\lambda x.t(x) \cdot b \to_{2\ell(t(x))} t(b)$*

So to continue with our previous example, we will calculate the n such that

$$\lambda x.\mathbf{d}(x)(1)(0) \cdot b \to_n \mathbf{d}(b)(1)(0).$$

For all lambda abstractions, this n is independent of b.

Proof. We will show this by induction on the complexity of $t(x)$.

If $t(x)$ is is x itself, then $\lambda x.t(x) \cdot b = \mathbf{skk}b \to_2 b$. When $t(x)$ is a constant c, then $\lambda x.t(x) \cdot b = \mathbf{k}cb \to_1 c$.

Now for the induction step, assume that $t(x) = t_1(x) \cdot t_2(x)$. Then $\ell(t(x)) = 1 + \ell(t_1(x)) + \ell(t_2(x))$.

$$
\begin{aligned}
\lambda x.t(x) \cdot b &= \mathsf{s}(\lambda x.t_1(x))(\lambda x.t_2(x))b \\
&\to_1 \lambda x.t_1(x) \cdot b(\lambda x.t_2(x) \cdot b) \\
\text{(I.H.)} &\to_{2\ell(t_1(x))} t_1(b)(\lambda x.t_2(x) \cdot b) \\
\text{(I.H.)} &\to_{2\ell(t_2(x))} t_1(b)(t_2(b)) \\
&= t(a)
\end{aligned}
$$

And $1 + 2\ell(t_1(x)) + 2\ell(t_2(x)) \leq 2\ell(t(x))$. □

We can now turn to the proof of the main theorem:

Proof of Theorem 5.2.1. We will show that the theorem holds with

$$k = 2\ell(t(x_1, \ldots, x_n)) \times \sum_{i=0}^{n-1} 5^i.$$

We will use induction. Lemma 5.2.3 shows us that the statement is true for $n = 1$. Now assume that it holds for n and let $|x| = (x_1, \ldots, x_{n+1})$.

Then by the induction hypothesis, we need $2\ell(\lambda x_{n+1} t(x_1, \ldots, x_{n+1})) \times \sum_{i=0}^{n-1} 5^i$ steps to rewrite $\lambda x_1 \ldots . \lambda x_{n+1} \cdot t(|x|) \cdot b_1 \cdot \ldots \cdot b_{n+1}$ into $\lambda x_{n+1}.t(b_1, \ldots, b_n, x_{n+1}) \cdot b_{n+1}$.

Then, by Lemma 5.2.3, we need $2 \times \ell(t(|x|))$ steps to rewrite it into $t(b_1, \ldots, b_{n+1})$.

Then, with Lemma 5.2.2, we have that the total number of steps is $2\ell(\lambda x_{n+1}.t(|x|)) \times \sum_{i=0}^{n-1} 5^i + 2\ell(t(|x|)) = 2 \times 5 \times \ell(t(|x|)) \times \sum_{i=0}^{n-1} 5^i + 2\ell(t(|x|))$ $= 2\ell(t(|x|)) \times \sum_{i=0}^{n} 5^i$. □

5.3 A fixpoint combinator

In order to reproduce recursion, we will use a fixpoint combinator. There exists a fixpoint combinator F for the SK-calculus:

Definition 5.3.1 ([13, p. 6])**.** $F = \mathbf{ssk}(\mathbf{s}(\mathbf{k}(\mathbf{ss}(\mathbf{s}(\mathbf{ssk})))))\mathbf{k})$

We have the following:

Proposition 5.3.1. *There exists a k such that for all x, $\mathsf{ACT} \vdash Fx \to_k x(Fx)$:*

Proof. Trivial. □

5.4 Arithmetical operations

5.4.1 Addition

We can define a term that calculates the sum of two natural numbers:

Definition 5.4.1.

$$\mathbf{plus}' := \lambda x.\lambda y.\lambda z.\mathbf{d}(z)(\mathbf{succ}(x \cdot y \cdot (\mathbf{pred}z)))(y)$$
$$\mathbf{plus} := F \cdot \mathbf{plus}'$$

As a recursive algorithm it would look like this:

```
plus(n,m):
If m = 0,
  output n
else
  -> plus(n, m-1) + 1
```

Proposition 5.4.1. $\mathsf{ACT} \vdash \exists c \forall n, m \in \mathbb{N}[\mathbf{plus} \cdot n \cdot m \to_{c(m+1)} n + m]$.

Proof. We will show this with induction. Assume that $m = 0$. Then,

$$\begin{aligned}
&\mathbf{plus} \cdot n \cdot 0 = F \cdot \mathbf{plus}' \cdot n \cdot 0 \\
(\text{Prop. 5.3.1}) &\to_k \mathbf{plus}' \cdot \mathbf{plus} \cdot n \cdot 0 \\
(\text{Th. 5.2.1}) &\to_{c'} \mathbf{d}(0)(\mathbf{succ}(\mathbf{plus} \cdot n \cdot (\mathbf{pred}0)))(n) \\
&\to_1 n
\end{aligned}$$

If we take $c = k + c' + 3$ then $\mathbf{plus} \cdot n \cdot 0 \to_c n + m$
Now assume that the statement holds for m.

$$\begin{aligned}
\mathbf{plus} \cdot n \cdot m + 1 &= F \cdot \mathbf{plus}' \cdot n \cdot m + 1 \\
&\to_k \mathbf{plus}' \cdot \mathbf{plus} \cdot n \cdot m + 1 \\
&\to_{c'} \mathbf{d}(m+1)(\mathbf{succ}(\mathbf{plus} \cdot n \cdot (\mathbf{pred}m + 1)))(n) \\
&\to_1 \mathbf{succ}(\mathbf{plus} \cdot n \cdot (\mathbf{pred}m + 1)) \\
&\to_1 \mathbf{succ}(\mathbf{plus} \cdot n \cdot m) \\
(\text{I.H.}) &\to_{c(m+1)} \mathbf{succ}(n + m) \\
&\to_1 n + m + 1
\end{aligned}$$

We have that $k + c' + 1 + 1 + c(m+1) + 1 = c(m+2)$. □

5.4.2 Subtraction

The following function works on two natural numbers x and y, and calculates $x - y$, but it gives 0 when $x < y$.

Definition 5.4.2.

$$\mathbf{min}' := \lambda x.\lambda y.\lambda z.\mathbf{d}(z)(\mathbf{pred}(x \cdot y \cdot (\mathbf{pred}z)))(y)$$
$$\mathbf{min} := F \cdot \mathbf{min}'$$

As a recursive algorithm it would be written like this (but in ACT, $0 - 1$ or **pred**0 gives 0).

```
min(n,m):
If m = 0,
  output n
else
  -> min(n, m-1) - 1
```

Proposition 5.4.1. $\mathsf{ACT} \vdash \exists c \forall n, m \in \mathbb{N}[\mathbf{min} \cdot n \cdot m \to_{c(m+1)} n - m]$.

The proof of this proposition, and the rest of the propositions in this chapter, can be found in the Appendix.

5.4.3 Multiplication

We can perform multiplication of natural numbers with this term:

Definition 5.4.3.

$$\mathbf{times}' := \lambda x.\lambda y.\lambda z.\mathbf{d}(z)(\mathbf{d}(\mathbf{pred}z)(\mathbf{plus}(x \cdot y \cdot (\mathbf{pred}z))y)(y))(0)$$
$$\mathbf{times} := F \cdot \mathbf{times}'$$

```
times(n,m):
If m = 0
  output 0
else
  if m=1
    output n
  else
    -> times(n, m-1) + n
```

Proposition 5.4.2. $\mathsf{ACT} \vdash \exists c \forall n, m \in \mathbb{N}[\mathbf{times} \cdot n \cdot m \to_{c(m+1)(n+2)} n \times m]$.

5.4.4 Division

We will give a function that for two natural numbers x and y, calculates $\lceil \frac{x}{y} \rceil$:

Definition 5.4.4.

$$\mathbf{div}' := \lambda x.\lambda y.\lambda z.\mathbf{d}(\mathbf{pred}z)(\mathbf{d}(\mathbf{min}\cdot y\cdot z)(\mathbf{succ}\cdot(x\cdot(\mathbf{min}\cdot y\cdot z)\cdot z))(1))(y)$$
$$\mathbf{div} := F\cdot\mathbf{div}'$$

```
div(n,m):
If m > 1
  if n > m
    -> div(n-m, m) + 1
  else
    output 1
else
  output n
```

Proposition 5.4.3. $\mathsf{ACT}\vdash\exists c\forall n,m\in\mathbb{N}[\mathbf{div}\cdot n\cdot m\to_{c(m+2)\lceil\frac{n}{m}\rceil}\lceil\frac{n}{m}\rceil]$.

5.4.5 Exponentiation

The following terms allow us to compute the exponent of two numbers.

Definition 5.4.5.

$$\mathbf{exp}' := \lambda x.\lambda y.\lambda z.\mathbf{d}(z)(\mathbf{times}\cdot y\cdot(x\cdot y\cdot(\mathbf{pred}z)))(1)$$
$$\mathbf{exp} := F\cdot\mathbf{exp}'$$

```
exp(n,m):
If m=0
  -> times(n, exp(n, m-1))
else
  output 1
```

Proposition 5.4.4. $\mathsf{ACT}\vdash\exists c\forall n,m\in\mathbb{N}[\mathbf{exp}\cdot n\cdot m\to_{c(n+2)\sum_{i=0}^{m-1}(n^i+1)}n^m]$.

5.4.6 Equality

The following functions works on two natural numbers. It outputs 1 when they are equal, and 0 when the numbers are unequal.

Definition 5.4.6.

$$\mathbf{e}' := \lambda x.\lambda y.\lambda z.\mathbf{d}(y)(\mathbf{d}(z)(x\cdot(\mathbf{pred}n)\cdot(\mathbf{pred}m))(0))(\mathbf{d}(z)(0)(1))$$
$$\mathbf{e} := F\cdot\mathbf{e}'$$

```
e(n,m):
if n > 0
  if m > 0
    -> e(n-1, m-1)
  else
    output 0
else
  if m > 0
    output 0
  else
    output 1
```

Proposition 5.4.5. *Let* $\min(x, y)$ *denote the minimum of* x *and* y*. Then,*

$$\mathsf{ACT} \vdash \exists c \forall n, m \in \mathbb{N}[\mathbf{e}_{c(\min(n,m)+1)} z],$$

where $z = 1$ *when* n *and* m *are equal and* $z = 0$ *otherwise.*

5.5 Lists

5.5.1 Lists of natural numbers

Lists are constructed by repeated pairing. We need something to represent the empty list. Because our if-then-else combinator $\mathbf{d}$ works with natural numbers in the if-statement, it is the most convenient to somehow denote the empty list with natural numbers. We therefore chose to represent the empty list with $\mathbf{p}00$. In order to distinguish the 0's in the empty list from the other natural numbers, we will add 1 to every number in the list. A delimited list $[a_0, a_1, \ldots, a_{n-1}]$ is then represented by

$$\mathbf{p}(a_0 + 1)(\mathbf{p}(a_1 + 1) \ldots (\mathbf{p}(a_{n-1} + 1)\mathbf{p}00) \ldots).$$

We will denote the length of a list t by $lh(t)$. In Section 5.5.2 we will give functions for giving the length of a list and obtaining an element at a certain index. We have that $\ell(\mathbf{p}00) = 5$ and $lh(\mathbf{p}00) = 0$. We also have the following:

Lemma 5.5.1. *When* t *is a list of natural numbers, all smaller than some* N*, then* $\ell(t) \leq (N + 5) \times lh(t) + 5$*.*

Proof. Let t be a list of natural numbers of length m, so $t = \mathbf{p}n_0\mathbf{p}n_1\mathbf{p} \cdots \mathbf{p}n_{m-1}\mathbf{p}00$. Then $\ell(t)$ will be equal to 5 plus the number of $\mathbf{p}'s$ in t without the $\mathbf{p}00$. Thus, $\ell(t)$ is equal to $lh(t)$ plus the number of applications, which is $2lh(t)$ plus the sum of the natural numbers that appear in t. So $\ell(t) = 5 + 3lh(t) + \sum_{i=0}^{lh(t)-1}(\ell(t_i) + 1)$. Since all natural numbers in t are bounded by N, and for an $n \in \mathbb{N}$ $\ell(n) = n + 1$, we have that $\sum_{i=0}^{lh(t)-1}(\ell(t_i) + 1) \leq (N + 2) \times lh(t)$. We conclude that $\ell(t) \leq (N + 5) \times lh(t) + 5$. $\square$

5.5.2 Operations on lists

We will define functions that give the length of a list, and give an element at a certain index of the list. We will denote elements of a sequence s at index i with s_i, where the first element has index 0. The following function will give the length of a list:

Definition 5.5.1.

$\mathbf{len}' := \lambda x.\lambda y.\mathbf{d}(\mathbf{p}_0 y)(\mathbf{succ}(x \cdot (\mathbf{p}_1 y)))(0)$
$\mathbf{len} := F \cdot \mathbf{len}'$

```
len(s):
If s has a first element x
  -> len(tail(s)) + 1
else
  output 0
```

The function $tail(s)$ is taken from the programming language Haskell and gives the list s without the first element. The first element of a list s is given by $head(s)$.

We can give the complexity of calculating the length of a list of natural numbers, but ACT has no way of checking that some term s is actually such a list.

Proposition 5.5.1. *When s is a sequence of natural numbers of length n, then*

$$\mathsf{ACT} \vdash \exists c[\mathbf{len} \cdot s \to_{c \times n} n].$$

The following function gives an element at a certain index:

Definition 5.5.2.

$\mathbf{ind}' := \lambda x.\lambda y.\lambda z.\mathbf{d}(z)(x \cdot (\mathbf{p}_1 y)(\mathbf{pred} z))(\mathbf{p}_0 y)$
$\mathbf{ind} := F \cdot \mathbf{ind}'$

```
ind(s,n):
If n=0
  output head(s)
else
  -> ind(tail(s), n-1)
```

Proposition 5.5.2. *When s is a sequence of natural numbers of length n, then*

$$\mathsf{ACT} \vdash \exists c \forall m \leq n[\mathbf{ind} \cdot s \cdot m \to_{c \times m} s_m].$$

6 Invariance with respect to Turing Machines

We have to argue that the system ACT acts a reasonable model of computation. A definition of a reasonable model is stated in the *Invariance Thesis*, from Slot and van Emde Boas [11]:

Invariance Thesis. *Reasonable models of computation simulate each other with polynomially bounded overhead in time and constant factor overhead in space.*

It was already proven that lambda calculus is such a reasonable model with respect to time-bounded computation [1, 2]. In order to prove that this also holds for ACT, we will show two things. Firstly, that there is a Turing machine that can rewrite a term using the axioms of ACT with a number of steps that is at most polynomially many more than we would have used in the system itself. Secondly, that for each Turing machine there is a term that will simulate it using at most polynomially many more steps than the machine itself.

There is one issue: the rewriting of terms is not deterministic. We will fix a specific rewriting rule for showing these two results. Namely, we will rewrite the terms using the axiom for the leftmost combinator that can be rewritten.

6.1 A Turing machine that rewrites terms

We will describe a Turing machine that simulates the rewriting of terms in ACT. That is, we prove the following theorem:

Theorem 6.1.1. *There is a Turing machine that with the binary description of a term t as an input, outputs z in $poly(n\ell(t))$ steps when $t \to_n z$ using a leftmost reduction-technique and z is in normal form.*

Before we do this, we need to show how we represent terms as binary strings.

6.1.1 Encoding terms in binary

We can uniquely encode terms as binary strings, and give an upper bound to the length of the encoding. This work is inspired by the work of Tromp [13]. We will represent binary strings as a concatenation of the numbers **0** and **1** in bold, so $\mathbf{01}^4 \equiv [0, 1, 1, 1, 1]$.

The encoding $\langle\cdot\rangle$ is as follows:

Definition 6.1.1.

$$\begin{aligned}
\langle \mathbf{k} \rangle &\equiv \mathbf{01}^2 \\
\langle \mathbf{s} \rangle &\equiv \mathbf{01}^3 \\
\langle \mathbf{p} \rangle &\equiv \mathbf{01}^4 \\
\langle \mathbf{p}_0 \rangle &\equiv \mathbf{01}^5 \\
\langle \mathbf{p}_1 \rangle &\equiv \mathbf{01}^6 \\
\langle \mathbf{succ} \rangle &\equiv \mathbf{01}^7 \\
\langle \mathbf{pred} \rangle &\equiv \mathbf{01}^8 \\
\langle \mathbf{d} \rangle &\equiv \mathbf{01}^9 \\
\text{For } n \in \mathbb{N},\ \langle n \rangle &\equiv \mathbf{01}^{n+10} \\
\langle x \cdot y \rangle &\equiv \mathbf{01} \langle x \rangle \langle y \rangle
\end{aligned}$$

Each binary sequence encodes a term. For a sequence s, we will denote the corresponding term with $\overline{s}$. Any sequence of zeroes at the end of the encoding can be ignored. So, for all terms t and $n \in \mathbb{N}$, $\overline{\langle t \rangle} = \overline{\langle t \rangle \mathbf{0}^n} = t$.

We can give the following upper-bound on the length of the encoding:

Lemma 6.1.2. *We have that $\ell(\langle x \rangle) \leq 82 \cdot \ell(x)$, so the length of the encoding is linear in the size of the represented term.*

Proof. This can be shown with induction. We have that:

$$\begin{aligned}
\ell(\langle n \rangle) &= \ell(\mathbf{01}^{n+10}) \\
\text{(Lem. 5.5.1)} &\leq (2+5) \cdot (n+11) + 5 \\
&= 7(n+1) + 75 \\
&= 7(\ell(n)) + 75 \\
&\leq 82\ell(n)
\end{aligned}$$

Now assume that the statement holds for x and y. Then:

$$\begin{aligned}
\ell(\langle x \cdot y \rangle) &= \ell(\mathbf{01} \langle x \rangle \langle y \rangle) \\
&= \ell(\mathbf{01}) - 5 + \ell(\langle x \rangle) - 5 + \ell(\langle y \rangle) \\
\text{(I.H.)} &\leq 11 + 82\ell(x) - 5 + 82\ell(y) \\
&\leq 82(1 + \ell(x) + \ell(y)) \\
&= 82\ell(x \cdot y)
\end{aligned}$$

□

6.1.2 Simulating the reduction steps

Before we can introduce the algorithm that rewrites the terms, we will need to verify that our terms cannot grow too fast in length, similarly to how invariance for lambda calculus was proven in [1]. We also need an additional definition:

Definition 6.1.2. We say that x is a *subterm of* t when x is t itself or x appears in t by application with other terms.

In order to verify that the terms cannot grow exponentially using a polynomially number of steps, we will show the following:

Lemma 6.1.3. *When $t \to_k u$, by always applying the axiom for the leftmost combinator that can be rewritten, then at every reduction-step, nothing that is duplicated contains two identical subterms that appeared by a single-step duplication, so $\ell(u) \leq (k+1)\ell(t)$.*

The proof of this lemma can be found in the Appendix.

We also need to make sure that choosing the subterm that we will apply one of the axioms to does not take too much time:

Lemma 6.1.4. *Finding the combinator for which we will use the corresponding axiom for the reduction of a term t can be done in time polynomial in $\ell(t)$.*

Proof. A Turing program for finding the combinator that we will apply one of the basic axioms to can be described as follows. We assume that we are working with a Turing machine with extra work tapes, so that we can make use of counters.

The machine will read the term from left to right, searching for the combinator that is closest to the beginning. We know that we've found a combinator when we have at least two 1's after a 0.

By reading the number of 1's we can figure out which of the combinators it represents. Then, we will denote the location of the combinator and we will check whether what comes after the combinator is such that we can apply the corresponding axiom. If we cannot reduce this combinator then from there we go further to the right to search for another combinator, making use of the denoted location.

We will describe how we can check whether the combinator can be reduced with an example. Suppose we found the **s** combinator and we want to know whether the subterm is of the form $\mathbf{s} \cdot x \cdot y \cdot z$, or, whether there's $\mathbf{01010101}^3\langle x\rangle\langle y\rangle\langle z\rangle$ written on the tape.

We need to make sure whether there are three separate subterms written after the $\mathbf{01}^3$. We start with x. That can be a combinator, or some application of two other terms $t_1 \cdot t_2$.

We can think of terms as trees: see Figure 6.1. We go trough the substring $\langle x\rangle$ from left to right. We denote the location of the beginning of the subterm x, which we will use in the proof of Theorem 6.1.1. We will also use a counter, which is initially set at zero. The maximum number that the counter will show gives the depth of the tree that represents the subterm x. Each time we read an application (or **01**), we add one

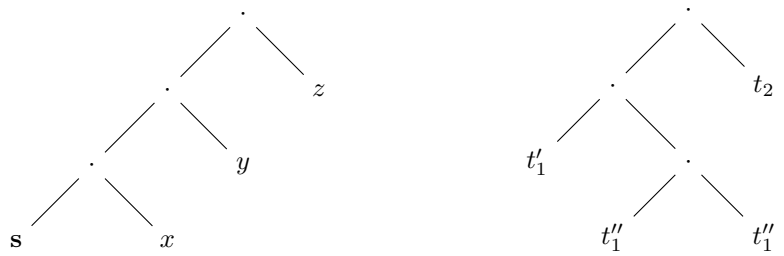

Figure 6.1: Terms as trees

to the counter. Each time we read a combinator, we subtract one from the counter. If we read a combinator while the counter is at zero, then we've reached the end of the subterm. This works, since with every application, we allow for one extra combinator in the term. We then continue to check the next subterms this way, and we also denote the location of the beginning of them.

This process of checking a subterm needs time linear in the length of the string and thus also in the length of the term (using Lemma 6.1.2).

If what comes next is not correct for applying the axiom (so no $\langle x\rangle\langle y\rangle\langle z\rangle$ for some x, y and z), we will go back to that combinator. This can also be done within linear time. From there we will go to the right again, looking for the next combinator, we denote the location of that one and we start again.

This process will repeat itself until we've found a combinator that we can rewrite (and if this doesn't happen then we are done and we output what's on the tape). We need to iterate this at most $\ell(t)$ times. In order to find a combinator to rewrite we would thus use a number of steps that is polynomial in $\ell(t)$. □

We can now give the proof of the main theorem:

Proof of Theorem 6.1.1. Assume that we have a term t, and $t \to_n z$, with z in normal form, where we always used the axiom for the leftmost combinator that could be rewritten.

By Lemma 6.1.4, we know that we can find the combinator for the reduction step of a term s in polynomial time. By Lemma 6.1.3, we know that every term that appears in the computation has length at most $(n+1)\ell(t)$. So each time, finding the combinator that we will apply one of the basic axioms to will need time polynomial in $(n+1)\ell(t)$.

Because we denoted the locations of the subterms that are going to get changed by the reduction, rewriting the term can also be done in polynomial time.

We then repeat this process n times. The entire computation can thus be done in time polynomial in $\ell(t)$ and n □

6.2 A term that simulates Turing machines

In this section we will describe how for each Turing machine, we can find a term that simulates that Turing machine.

We will adapt the Turing machine such that it is a machine with only one one-way infinite tape and that its computation halts when the head is located at the first bit of the output. For any Turing machine there exists another machine that is of this form and computes the same function with a polynomially bounded overhead in time [3, Ch. 1]. We also assume that the machine reads the entire input during the computation, which is often done in standard computational complexity.

We have that Γ is the alphabet with size l, that Q the set of k states, where q_1 is the starting state and q_k the halting state, and that $\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R, S\}$ is the transition function.

In order to describe a single configuration of the Turing machine during its computation, we will use the following tuple of four [14]:

$$C = (q, [T(n)], [T(n-1), T(n-2), \ldots, T(0)], [T(n+1), T(n+2), \ldots T(n+m)]).$$

The symbol q will be a number that represents the state of the Turing machine. $T(n)$ is the symbol under the head of the Turing machine. The other lists describe the symbols before and after $T(n)$.

In reality the Turing machine has an infinite tape, with an infinite number of "blank"-symbols going both sides, but we do not need to encode these in order to simulate the computation. We will denote the empty list by '[]', (see Section 5.5.1).

Assume that a Turing machine M will receive the input $x \in \{0,1\}^n$, and that it runs in time $f(n)$. We describe a term Δ that will simulate the computation of the machine on input $(0, x_0, [], [x_1, \ldots, x_{n-1}])$.

Definition 6.2.1.

$$\begin{array}{lll}
\Delta \cdot C := & \text{when } q_1 = C_0 \Rightarrow & \text{when } \gamma_1 = C_1 \quad \Rightarrow \Delta \cdot (q, \gamma, C_2', C_3') \\
 & & \text{when } \gamma_l = C_1 \\
 & \text{when } q_{k-1} = C_0 \Rightarrow & \\
 & \text{when } q_k = C_0 \Rightarrow & \mathbf{p} C_1 C_3
\end{array}$$

We cannot really perform recursion like this in ACT. However, using lambda abstractions and the F-combinator will unnecessarily complicate the description of this term. As we saw in Chapter 5, using this definition will have no influence on the complexity-analysis of the actual Δ's.

C_i is shorthand for $\mathbf{ind}Ci$ (see Section 5.5.2), and for checking equality we will use $\mathbf{e}nm$ (Section 5.4.6).

The 4-tuple (q, γ, C_2', C_3') will be set for each case, depending on δ:

1. When $\delta(q_i, \gamma_j) = (q_{i'}, \gamma_{j'}, L)$, then we put $(q_{i'}, \mathbf{p}_0 C_2, \mathbf{p}_1 C_2, \mathbf{p}\gamma_{j'} C_3)$.
2. When $\delta(q_i, \gamma_j) = (q_{i'}, \gamma_{j'}, R)$, then we put $(q_{i'}, \mathbf{p}_0 C_3, \mathbf{p}\gamma_{j'} C_2, \mathbf{p}_1 C_3)$.
3. When $\delta(q_i, \gamma_j) = (q_{i'}, \gamma_{j'}, S)$, then we put $(q_{i'}, \gamma_{j'}, C_2, C_3)$.

We will show that the term Δ reduces to a term in normal form with a number of steps that is polynomial in $f(n)$.

Theorem 6.2.1. *For each TM M, there is a term t such that for $x \in \{0,1\}^n$, $\mathsf{ACT} \vdash t \cdot x \to_m M(x)$, where m is polynomial in the computation time of M.*

Proof. Let M be the Turing machine described in the beginning of this section. Assume that M runs in time $f(n)$.

It is clear that there exists a term that on input $[x_0, \ldots, x_{n-1}]$, in linear time computes $(q_1, x_0, [], [x_1, \ldots, x_{n-1}])$.

We will then give this to Δ, which will compute $M(x)$. The rest of the proof will be a time-analysis of Δ.

Since there is a constant number of states, each q_i will be a natural number bounded by some constant. Since $\mathbf{e}nm$ runs in time linear in $\min(n, m)$, we will have that checking the equality for the states can be done in constant time. The same holds for checking equality for the symbols γ_i. Creating the new 4-tuple can also be done within a constant number of steps for each input.

We will need to perform $f(n)$ recursive applications of Δ, since M makes $f(n)$ steps. So the total time that Δ needs is $c \cdot f(n)$ for some c. $\square$

7 Complexity Results

In this chapter we will prove some results from standard computational complexity. As stated in Section 2.1.3, diagonalization is referred to as a relativizing proof-technique. However, we need more axioms before we can actually use diagonalization. Therefore, we will introduce an extension to ACT. We then define several complexity classes, and deduce some results concerning those.

7.1 Universal computation

In order to use diagonalization, we need the following:

1. An enumeration of all programs
2. A program that can simulate other programs

We will formalize these two statements in several axioms. We introduce two extra combinators: an encoding, **enc**, and its inverse, **dec**. We also have a universal combinator, **u**, that can simulate other terms.

U.1 $\forall x \exists n \in \mathbb{N}[\mathbf{enc} \cdot x \to_1 n \wedge \mathbf{dec} \cdot n \to_1 x]$

U.2 $\forall n \in \mathbb{N} \exists x[\mathbf{dec} \cdot n \to_1 x \wedge \mathbf{enc} \cdot x \to_1 n]$

U.3 $\forall n_1, n_2, n_3, n_4 \in \mathbb{N} \exists m \in \mathbb{N}[\mathbf{u} \cdot n_1 \cdot n_2 \cdot n_3 \cdot n_4 \to_{n_1^2} m \wedge (m = 1 \vee m = 0)]$

U.4 $\forall n_x, n_i, n_t, m \in \mathbb{N}[(\mathbf{dec} \cdot n_x) \cdot n_i \to_{n_t} m \Rightarrow \mathbf{u} \cdot n_t \cdot n_x \cdot n_i \cdot m \to 1]$

U.5 $\forall n_x, n_i, n_t, m \in \mathbb{N}[\neg((\mathbf{dec} \cdot n_x) \cdot n_i \to_{n_t} m) \Rightarrow \mathbf{u} \cdot n_t \cdot n_x \cdot n_i \cdot m \to 0]$

The universal combinator is in some way able to tell us whether $x \cdot n_i \to_{n_t} m$ or not, even though this is not decidable in ACT itself. We also have that the first natural number that **u** is applied to, gives us a bound on the total computation time of **u**. The combinator eventually always outputs 0 or 1.

We call the system ACT together with these axioms and the symbols for **enc**, **dec** and **u** the system $\mathsf{ACT}+\mathsf{U}$. Similarly, we have $\mathsf{ACT}^f+\mathsf{U}$, where of course in the axioms U.1 until U.5 we use $\to^f$. We also have the following:

Corollary 7.1.1. *For all total recursive f and for all formulas φ:*

$$\mathsf{ACT} + \mathsf{U} \vdash \varphi \Rightarrow \mathsf{ACT}^f + \mathsf{U} \vdash \varphi.$$

7.2 Enumerating the terms

What seems useful in proofs that use diagonalization, is to actually have an enumeration of terms that satisfies the following properties:

i Every number represents a term.

ii Every term occurs infinitely many often in the enumeration.

The first property is satisfied by axiom $U.2$. We will use the following enumeration to satisfy the second property:

1. $(1,1)$
2. $(2,1)$
3. $(2,2)$
4. $(3,1)$
5. $(3,2)$
6. $(3,3)$
7. $(4,1)$
8. $\cdots$

We ensured that each number appears infinitely-many often on the right side of the pairs. We also have that for any numbers m and c, there exists a pair (k,m) in the enumeration such that $c \leq k$. Later on, we will see that this is useful for diagonalization.

There is a program that on input n, gives the pair (k,m) from the enumeration in polynomial time. We will not prove this formally, but since we have that $1+2+\cdots+k = \frac{k(k+1)}{2}$, the following algorithm will work:

```
enum(n):
For i = 1, ...
  if n - i(i+1)/2 <= i
    if n - i(i+1)/2 = 0
      output (i, i)
    else
      output (i+1, n - i(i+1)/2)
```

We thus have the following term:

Definition 7.2.1.

$$\begin{aligned}
\mathbf{enum}'' := \lambda x.\lambda n.\lambda c.\\
&\mathbf{d}\big(\mathbf{min}(\mathbf{min}n(\mathbf{div}(\mathbf{times}i(\mathbf{succ}i))2))i\big)\\
&\big(\mathbf{d}(\mathbf{e}0(\mathbf{min}n(\mathbf{div}(\mathbf{times}i(\mathbf{succ}i))2)))\\
&(\mathbf{p}ii)\\
&(\mathbf{p}(\mathbf{succ}i)(\mathbf{min}n(\mathbf{div}(\mathbf{times}i(\mathbf{succ}i))2))))\big)\\
&\big(xn(\mathbf{succ}i)\big)\\
\mathbf{enum}' := F \cdot \mathbf{enum}''\\
\mathbf{enum} := \lambda n.\mathbf{enum}' \cdot n \cdot 1
\end{aligned}$$

We will assume that for any n, $\mathbf{enum} \cdot n$ will give the pair (k, m) in polynomial time.

7.3 Complexity classes

In the system ACT, complexity classes are represented by formulas. We will only consider the complexity of terms that compute functions from $\mathbb{N}$ to $\mathbb{N}$. A standard complexity class will look like this:

For a term t, $C_g(t)$ holds when $\exists t', t' \doteq t$ and $\forall n \in \mathbb{N}\ \exists m \in \mathbb{N}$ s.t. $t' \cdot n \to_{g(n)} m$.

The class of polynomial-time 0,1-valued functions can be described in the following way:

Definition 7.3.1. For a term t, $P(t)$ is true when there exists a term t', $t \doteq t'$, and constants c, c', such that for all $n \in \mathbb{N}$ there exists an m, $m = 0$ or $m = 1$, and $t' \cdot n \to_{n^c + c'} m$.

There is a difference between the formula P and the class $\mathbf{P}$ from standard computational complexity. For the complexity, we consider functions that are polynomials in the input, instead of polynomials in the length of the input.

We defined the length of a natural number n to be $n + 1$, so we can consider this as functions in the length of n. We could've chosen to consider functions as $\log(n)^c + c'$, to capture the length of the binary string that represents n. However, choosing the actual natural number as input for the complexity bound gives complexities of for example addition and multiplication that are in correspondence with the Turing complexities of those functions.

We also have exponential-time computation:

Definition 7.3.2. For a term t, $EXP(t)$ is true when there exists a term t', $t \doteq t'$, and constants c, c', such that for all $n \in \mathbb{N}$ there exists an m, $m = 0$ or $m = 1$, and $t' \cdot n \to_{2^{n^c} + c'} m$.

We can describe a formula that is true when s is a binary sequence:

$$\exists n, m[\mathbf{len}s \to_n m \wedge \forall i \leq m(\mathbf{ind}si = 0 \vee \mathbf{ind}si = 1)].$$

With that, we will define nondeterministic polynomial-time as follows:

Definition 7.3.3. For a term t, $NP(t)$ is true when for all n there exists an m such that $t \cdot n \to_m 0$ or $t \cdot n \to_m 1$. We also need that there exists a term t', constants c, c', and a polynomial p, such that for all $n \in \mathbb{N}$ and binary sequences s of length $p(n)$, there exists an m, $m = 0$ or $m = 1$, such that $t' \cdot (n, s) \to_{n^c+c'} m$. We have that

$$\forall n[t \cdot n \to 1 \Leftrightarrow \text{ There exists a binary sequence } s(\mathbf{len}s \leq p(n) \wedge t' \cdot (n, s) \to_{n^c+c'} 1)].$$

We only focused on 0/1-functions in this section. The classes P and EXP could of course also be defined for all functions. This is however not possible for NP.

7.4 Time Hierarchy Theorem

We will outline how to reproduce the Time Hierarchy Theorem of Hartmanis and Stearns with $\mathsf{ACT} + \mathsf{U}$.

Theorem 7.4.1 ([8])**.** $\mathsf{ACT} + \mathsf{U} \vdash P \neq EXP$

The original Time Hierarchy Theorem gives a sharper bound. The version that we prove here is easier to understand and gives the reader an impression of the proof-method that was used.

Proof. We will show the statement by actually showing the following

$$\mathsf{ACT} + \mathsf{U} \vdash \exists t[EXP(t) \wedge \neg P(t)].$$

We let $t = \lambda n.\mathbf{d}\big(\mathbf{u} \cdot (\mathbf{plus} \cdot (\mathbf{exp} \cdot (\mathbf{p}_1(\mathbf{enum} \cdot n)) \cdot (\mathbf{p}_0(\mathbf{enum} \cdot n))) \cdot (\mathbf{p}_1(\mathbf{enum} \cdot n))) \cdot (\mathbf{p}_1(\mathbf{enum} \cdot n)) \cdot (\mathbf{p}_1(\mathbf{enum} \cdot n)) \cdot 1\big)\big(0\big)\big(1\big)$.

Or:

```
t(n):
enum(n) = (k,m)
If u (m^k + k, m, m, 1) -> 1
  Output 0
else
  Output 1
```

So on input n, let $\mathbf{enum} \cdot n \to (k, m)$. The algorithm checks whether

$$(\mathbf{dec} \cdot m) \cdot m \to_{m^k+k} 1.$$

If yes, we output 0 and if no we output 1.

It is obvious that $\mathsf{ACT} + \mathsf{U} \vdash EXP(t)$. Now we show that $\mathsf{ACT} + \mathsf{U} \vdash \neg P(t)$. So assume that $P(t)$ holds. Then there exists a term t' and constants c and c', such that

$t \doteq t'$ and $t' \cdot n \to_{n^c+c'} 0/1$. By Axiom U.2 we have that there exists an m such that $\mathbf{dec} \cdot m \to_1 t'$. By the enumeration of the programs, we have that there exists an n such that $\mathbf{enum} \cdot n = (k, m)$ and $m^c + c' \leq m^k + k$. This means that $t' \cdot m \to 1 \Rightarrow t \cdot m \to 0$ and $t' \cdot m \to 0 \Rightarrow t \cdot m \to 1$. Contradiction. □

7.5 P vs NP is independent from $\mathsf{ACT} + \mathsf{U}$

In this section, we will show the following theorem:

Theorem 7.5.1. *$P = NP$ is independent from $\mathsf{ACT} + \mathsf{U}$.*

We will obtain this result by giving oracles f_A, f_B such that $\mathsf{ACT}^{f_A} + \mathsf{U} \vdash P = NP$ and $\mathsf{ACT}^{f_B} + \mathsf{U} \nvdash P = NP$. Then, together with Corollary 7.1.1, we have the theorem.

7.5.1 Relativized world where $P = NP$

Lemma 7.5.2. *[5] There exists a total recursive f_A such that $\mathsf{ACT}^{f_A} + \mathsf{U} \vdash P = NP$*

The proof of this lemma is based upon [3, Th. 3.7].

Proof. We assume that we have a suitable encoding and decoding for triples, that can be computed within polynomial time, such that for any $n \in \mathbb{N}$, there exists $x, y, z \in \mathbb{N}$ with $n = (x, y, z)$.

We let $t_{f_A} := \lambda(x, y, z).\mathbf{u} \cdot (\mathbf{exp} \cdot 2 \cdot x) \cdot y \cdot z \cdot 1$. The decoding of the triple is part of the computation of $t_{f_A} \cdot n$. t_{f_A} will thus simulate a computation ($\mathbf{dec}(y)$ on input z) for exponential time (2^x) and tell us whether the program accepted.

We can assume that $\mathsf{ACT}^{f_A} + \mathsf{U} \vdash \forall x[P(x) \Rightarrow NP(x)]$.We will now argue that $\mathsf{ACT}^{f_A} + \mathsf{U} \vdash \forall x[NP(x) \Rightarrow P(x)]$.

Assume that for some t we have that $NP(t)$ holds. Then for all n, $t \cdot n \to 0/1$. There also exists a term t', constants c, c' and a polynomial p, such that for all n, $t \cdot n \to 1$ iff there exists a binary sequence s such that $\mathbf{len} s \leq p(n) \wedge t' \cdot (n, s) \to_{n^c+c'} 1$.

We can now make a make a term x as follows. On input n, we enumerate all binary sequences of length at most $p(n)$. This can be done in exponential time. For each of such sequences s, we run t' on input (n, s). This is done in polynomial time. If for one sequence t' gives 1, x also outputs 1. If this doesn't happen for any sequence then x outputs 0.

In total, x runs in exponential time, say, $2^{q(n)}$ for some q. We also have that there exists a term m such that $\mathbf{enc} \cdot x \to_1 m$. With this, we can make a term y as follows.

On input n, encode a triple $(m, q(n), n)$ and give this to the oracle $\mathbf{o}_{f_A}$. Then output the same result. We have that y needs polynomial time for the computation, and $y \doteq t$. So $P(t)$ holds. □

7.5.2 Realizability with truth

We cannot actually use the known proof of Baker, Gill and Solovay in [5] to show that there exists a function f_B such that $\mathsf{ACT}^{f_B} + \mathsf{U} \vdash P \neq NP$. This is because our universal term does not allow us to have specific behaviour depending on what string is given to the oracle. Also, the creation of the oracle f_B in several stages poses a problem. However, with results from Chapter 6, on the invariance of ACT with respect to Turing machines, we can argue for the existence of a term that acts as a universal machine and has somewhat more flexibility with respect to its behaviour. We will show the following theorem, which still depends on results from [5]:

Theorem 7.5.3. *There exists an oracle f_B and a term t such that $\mathsf{ACT}^{f_B} + \mathsf{U} \vdash NP(t)$ but $\mathsf{ACT}^{f_B} + \mathsf{U} \nvdash P(t)$.*

This theorem is implies that $\mathsf{ACT}^{f_B} + \mathsf{U} \nvdash P = NP$. However, some of the reasoning in the proof can thus only be done in the meta-theory. In order to obtain a contradiction after assuming that $\mathsf{ACT}^{f_B} + \mathsf{U} \vdash P(t)$, we need to make sure that statements from $\mathsf{ACT}^{f_B} + \mathsf{U}$ about terms also extend to the meta-theory. We thus first need to show the following:

Lemma 7.5.4. *When $\mathsf{ACT}^{f} + \mathsf{U} \vdash \exists x \varphi(x)$, then there exists a closed term t such that $\mathsf{ACT}^{f} + \mathsf{U} \vdash \varphi(t)$.*

We will do this by using realizers. First, for any formula φ, we define another formula $x \ \underline{rt} \ \varphi$ as follows:

Definition 7.5.1.

$$
\begin{aligned}
x \ \underline{rt} \ \phi &:= \phi \text{ if } \phi \text{ is an atomic formula} \\
x \ \underline{rt} \ (\phi \wedge \psi) &:= \mathbf{p}_0 x \ \underline{rt} \ \phi \wedge \mathbf{p}_1 x \ \underline{rt} \ \psi \\
x \ \underline{rt} \ (\phi \vee \psi) &:= (\mathbf{p}_0 x = 0 \Rightarrow (\mathbf{p}_0 \mathbf{p}_1 x)0 \ \underline{rt} \ \phi) \wedge (\mathbf{p}_0 x \neq 0 \Rightarrow (\mathbf{p}_1 \mathbf{p}_1 x)0 \ \underline{rt} \ \psi) \\
x \ \underline{rt} \ (\phi \Rightarrow \psi) &:= (\phi \Rightarrow \psi) \wedge \forall y (y \ \underline{rt} \ \phi \Rightarrow x \cdot y \ \underline{rt} \ \psi) \\
x \ \underline{rt} \ \exists y \phi &:= \mathbf{p}_1 x \ \underline{rt} \ \phi(\mathbf{p}_0 x) \\
x \ \underline{rt} \ \forall y \phi &:= \forall y (x \cdot y \ \underline{rt} \ \phi)
\end{aligned}
$$

We can now turn to the proof of the lemma. Since the proof is quite similar to how this is shown other systems, such as for HA^* in [6], we will not fill in all the details.

Proof sketch. We will first show that $x \to_n y$ and $y \ \underline{rt} \ \varphi$ implies that $x \ \underline{rt} \ \varphi$.

This is done with induction on the complexity of φ. When it is an atomic formula then the statement obviously holds. When $y \ \underline{rt} \ \varphi$ and $\varphi := \phi \wedge \psi$, then by definition $\mathbf{p}_0 y \ \underline{rt} \ \phi$ and $\mathbf{p}_1 y \ \underline{rt} \ \psi$. We have that $x \to_n y$ implies that $\mathbf{p}_0 x \to_n \mathbf{p}_0 y$. So by the induction hypothesis, also $\mathbf{p}_0 x \ \underline{rt} \ \phi$. The case for $\mathbf{p}_1 y$ is similar.

We can show the other cases for φ in the same way. We conclude that

$$(x \to_n y \wedge y \ \underline{rt} \ \varphi) \Rightarrow x \ \underline{rt} \ \varphi.$$

We have that for any φ without free variables, if $\mathsf{ACT}^f + \mathsf{U} \vdash \varphi$ then there exists a closed term t in the language of $\mathsf{ACT}^f + \mathsf{U}$ such that $\mathsf{ACT}^f + \mathsf{U} \vdash t \ \underline{rt} \ \varphi$. This can be shown by giving terms that realize the axioms. For implications we show that realizers for the premises give us realizers for the conclusions.

The combinators are realized by themselves. The axiom for induction can be realized with the following term i:

$$i' := \lambda x \lambda y \lambda z \mathbf{d}(z)(\mathbf{p}_1 y \cdot (\mathbf{pred} \cdot z) \cdot (x \cdot y \cdot (\mathbf{pred} \cdot z)))(\mathbf{p}_0 y)$$
$$i := F \cdot t$$

Now assume that $\mathsf{ACT}^f + \mathsf{U} \vdash \exists x \varphi(x)$. Then there exists an s such that $\mathsf{ACT}^f + \mathsf{U} \vdash s \ \underline{rt} \ \exists x \varphi(x)$. So, by definition, $\mathbf{p}_1 s \ \underline{rt} \ \varphi(\mathbf{p}_0 s)$. If we let $t = \mathbf{p}_0 s$, then the lemma follows. □

7.5.3 Relativized world where $P \neq NP$

We will first describe how we can construct a universal term. Take the Turing machine that simulates computations of terms, Theorem 6.1.1. Then simulate this Turing machine with the term that simulates Turing machines, Theorem 6.2.1. We now have such a universal term u'.

What's more, this term simulates other terms with only a polynomial overhead in time. Recall the encoding of terms as binary strings, Definition 6.1.1. Note that we can decide whether a binary string encodes a valid term. We can decide to let u' immediately output 0 on strings that do not encode a term. Then we also have that the representation of terms as binary strings satisfies the following two criteria:

i Every binary string represents a term.

ii Every term is represented by infinitely many binary strings.

We can also make a term u'' that works as u' only it can also read $\mathbf{o}_f$-combinators for a specific f, with an obvious modification to the binary representation of terms. We can expand the universal term even more, by giving it a counter to keep track of the reduction steps that are done. This way we can give an additional natural number as input and let the universal term simulate for at most that many steps.

There is one problem: for the proof we need that the oracle f_B is actually a function that takes binary strings as input. As was argued in Section 7.3, there is a formula that can tell us when a term is a binary string. We need to alter the axiom for the $\mathbf{o}_f$-combinator such that it also allows for oracles that are functions in binary strings.

Thus, let $f(x)$ denote the output of f on input x. Then:

B.11 $(\forall x : dom_f(x))[\mathbf{o}_f x \to_1^f f(x)]$

We can now mimic the known proof of Baker, Gill and Solovay to show Theorem 7.5.3:

Proof sketch. We will define a term t_{f_B} that computes function f_B, and for all binary sequences x, decide whether $f_B(x) = 1$ ($t_{f_B} \cdot x \to 1$) or $f_B(x) = 0$ ($B \cdot x \to 0$). We will do this in several stages. During this process, we need to keep track of sequences which we already considered. We will do this by updating list, L, that contains binary strings. We 'remember' L using recursion.

The rest of the proof then proceeds as outlined in for example [3, p. 74].

The idea is as follows. We will look at the function g that on input n, gives 1 when there is a binary sequence x of length n with and $t_{f_B} \cdot x \to 1$. If there isn't such a sequence x then $g(n) = 0$.

There exists a term t that computes this function g with $\mathbf{ACT}^{f_B} \vdash NP(t)$. This is because we can create a term t' that gets as an input an n and s, computes $\mathbf{o}_{f_B} \cdot s$ and outputs the result.

We will now define t_{f_B} such that we cannot have that $P(t)$ holds. We do this with diagonalization, for several stages.

In stage i, with the universal term, we can simulate a term t_i represented by string i on some input n_i for $\frac{2^{n_i}}{10}$ steps. We choose n_i such that it is bigger than the strings that we've previously put in the list L. During the computation, t_i can query the oracle for several strings, but at most $\frac{2^{n_i}}{10}$ many of them, so there are always strings of length n_i that t_i cannot query. If i halted within that time, then depending on whether $t_i \cdot n_i \to 1$ or $t_i \cdot n_i \to 0$ we can decide to put a string that it hasn't queried in the list L or put no strings of length n_i in L. We make sure that t_i gives the wrong answer on whether there exists a string of length n_i in L or not.

The term t_{f_B} then runs as follows: on input $x \in \{0,1\}^n$ it goes through all stages starting from 1 until we simulate a term on an input $n_i \geq n$. It then checks whether $x \in L$ or not and outputs 1 or 0 accordingly to the answer. So $t_{f_B} \cdot x \to 1 \Leftrightarrow x \in L$. By our construction of L and t_{f_B}, we have that all terms that run in polynomial-time have at least one input on which it differs with g.

So, when we assume that $\mathsf{ACT}^{f_B} + \mathsf{U} \vdash P(t)$, we have in $\mathsf{ACT}^{f_B} + \mathsf{U}$ that there exists a t', $t' \doteq t$, and that there exists constants c, c', such that for all $n \in \mathbb{N}$ there exists an m, $m = 0$ or $m = 1$, and $t' \cdot n \to_{2^{n^c}+c'} m$. Then, by Lemma , we have that there exists a closed term t'' such that in $\mathsf{ACT}^{f_B} + \mathsf{U}$ the above holds. However, this is in contradiction with our construction of t_{f_B}.

We can conclude that there exists a term t such that $\mathsf{ACT}^{f_B} + \mathsf{U} \vdash NP(t)$ and $\mathsf{ACT}^{f_B} + \mathsf{U} \nvdash P(t)$. □

7.5.4 Independence result

Lemma 7.5.5. $\mathsf{ACT}^{f_B} + \mathsf{U} \nvdash P = NP$

Proof. We have that $\mathsf{ACT}^{B} + \mathsf{U} \vdash NP(t)$ for the term t of Theorem 7.5.3. The theorem also shows that there cannot exist a term s such that $\mathsf{ACT}^{B} + \mathsf{U} \vdash t \doteq s \wedge P(s)$.

Assume, towards a contradiction, that $\mathsf{ACT}^B + \mathsf{U} \vdash P = NP$. Then we also have that $\mathsf{ACT}^B + \mathsf{U} \vdash \exists x[t \doteq x \wedge P(x)]$. With Lemma 7.5.3, we have that there exists a term s such that $\mathsf{ACT}^B + \mathsf{U} \vdash t \doteq s \wedge P(s)$. Contradiction. □

Theorem 7.5.6. *P vs NP is independent of* $\mathsf{ACT} + \mathsf{U}$

Proof. Suppose that $\mathsf{ACT} + \mathsf{U} \vdash P = NP$ or $\mathsf{ACT} + \mathsf{U} \vdash P \neq NP$. Then, by Corollary 7.1.1, for all f, also $\mathsf{ACT}^f + \mathsf{U} \vdash P = NP$ or $\mathsf{ACT}^f + \mathsf{U} \vdash P \neq NP$. But this is in contradiction with lemmas 7.5.2 and 7.5.5. □

Corollary 7.5.1. $\mathsf{ACT} + \mathsf{U}$ *is consistent.*

8 Conclusion

We've presented a system, ACT, in which we can express the complexity of solving problems. Everything that can be proven in ACT also relativizes.

The system uses combinatory logic to perform computations and it is invariant with respect to Turing machines.

A small extension of the system enables us to reason about computations relative to oracles. For any total computable function f, there exists the system ACT^f, where all computations are relative to the oracle f. Any proof in ACT can also be done within ACT^f for any f, which ensures that all proofs from ACT relativize. Both the systems ACT and ACT^f are consistent.

In ACT, we are only able to consider the time-bounded complexity of functions. It would be interesting to extend the system such that it can also express space-bounded complexity. The length of a term seems to be a good measure for this, and also allows for an orthodox interpretation of the invariance thesis (a single simulation achieves the polynomial overhead in time and constant factor overhead in space bounds). However, ACT is not able to talk about the length of its own terms.

A way of defining nondeterministic and probabilistic computations is also something to consider in future work. With probabilistic computations we can try to say more about proof-methods such as *arithmetization*, which is regarded as non-relativizing.

An important thing to note is that ACT cannot tell us which proof-methods do *not* relativize. We can only be sure that statements that can be proven within ACT are relativizing. Still, it is expected that we can obtain information on non-relativizing techniques with further studies.

But even then, it is difficult to mimic all proofs of standard computational complexity, since a lot of proofs are dependent on the exact workings of a Turing machine. To mimic the proof of the construction of the oracle relative to which $P \neq NP$ we had to use binary sequences. The proof could also not be done within the system itself, but only in the meta-theory. It would be better when this was possible within ACT and when we could construct an oracle that worked for natural numbers, instead of an oracle for binary sequences.

There has been done some work similar to this. In [4] and [9] a system based on Cobham's characterization of the polynomial-time functions was used. It was argued that in this system every proof relativizes. By adding axioms to the system the authors showed that non-relativizing results could be proven. This gives insight in the non-relativizing aspects of proofs. Since the system was based on polynomial-time functions, it is only possible to give definitions based on those. In ACT we can use any function for the time-bounded complexity.

In this work, we only used computable oracles. This was more in line with the constructive approach that we wanted to take. Besides this, in standard computational complexity almost all oracles that are used are actually computable.

We also chose to only have total functions as an oracle, even though perhaps partial functions would've worked as well. We did this, because it fits nicer with the requirement that the oracle is computable.

There is still a lot of work that needs to be done before we can obtain a system that is more expressive. So far we can only talk about the time-bounded complexity and deterministic computations. However, ACT sets a basis for further studies on this matter.

9 Acknowledgements

In the process of writing this thesis, my two supervisors where the most important. I would like to thank Benno van den Berg for giving me the idea to create this formal system. I also want to thank him for answering the numerous questions that came up throughout the whole process, even though he had perhaps too many other things to do. I also have lot of gratitude towards Leen Torenvliet. No matter the topic, he was always able to give me related papers to expand my background knowledge. But most of all, he was very supportive and believed in me, even when I didn't do this myself. I hope that in the future there will be more opportunities to collaborate with the both of them.

Since this thesis also finalizes my time in the Master of Logic, I want to say some things about the program. I very much enjoyed all the opportunities I got for conducting research in my own interests. But most of all, I was happy to be able to spend so much time together with the other students. They were inspiring in many ways and I've never felt like I fitted in as much as I did among them.

10 Bibliography

[1] Beniamino Accattoli and Ugo Dal Lago. "Beta reduction is invariant, indeed". In: *Proceedings of the Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*. ACM. 2014, p. 8.

[2] Beniamino Accattoli and Ugo Dal Lago. "On the invariance of the unitary cost model for head reduction". In: *LIPIcs-Leibniz International Proceedings in Informatics*. Vol. 15. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik. 2012.

[3] Sanjeev Arora and Boaz Barak. *Computational complexity: a modern approach*. Cambridge University Press, 2009.

[4] Sanjeev Arora, Russell Impagliazzo, and Umesh Vazirani. *Relativizing versus nonrelativizing techniques: the role of local checkability*. 1992.

[5] Theodore Baker, John Gill, and Robert Solovay. "Relativizations of the P=?NP question". In: *SIAM Journal on computing* 4.4 (1975), pp. 431–442.

[6] Samuel R Buss. *Handbook of proof theory*. Vol. 137. Elsevier, 1998.

[7] *Complexity Zoo*. `https://complexityzoo.uwaterloo.ca/Complexity_Zoo`. Accessed: 07/2018.

[8] Juris Hartmanis and Richard E Stearns. "On the computational complexity of algorithms". In: *Transactions of the American Mathematical Society* 117 (1965), pp. 285–306.

[9] Russell Impagliazzo, Valentine Kabanets, and Antonina Kolokolova. "An axiomatic approach to algebrization". In: *Proceedings of the forty-first annual ACM symposium on Theory of computing*. ACM. 2009, pp. 695–704.

[10] Dieter van Melkebeek. *Randomness and Completeness in Computational Complexity*. Vol. 1950. Jan. 2000. ISBN: 3-540-41492-4.

[11] Cees Slot and P Boas. "On tape versus core an application of space efficient perfect hash functions to the invariance of space". In: *Proceedings of the sixteenth annual ACM symposium on Theory of computing*. ACM. 1984, pp. 391–400.

[12] Anne S Troelstra and Dirk van Dalen. *Constructivism in mathematics, an Introduction*. Vol. 1 and 2. North-Holland, 1988.

[13] John Tromp. "Kolmogorov Complexity in Combinatory Logic". In: (2002).

[14] Alan M Turing. "Computability and Lambda-Definability". In: *The Journal of Symbolic Logic* 2.4 (1937), pp. 153–163.

11 Appendix

11.1 Proofs of Chapter 5

Proposition 5.4.1. $\mathsf{ACT} \vdash \exists c \forall n, m \in \mathbb{N}[\mathbf{min} \cdot n \cdot m \to_{c(m+1)} n - m]$.

Proof. This is done by induction on m. If $m = 0$:

$$\begin{aligned}
&\mathbf{min} \cdot n \cdot 0 = F \cdot \mathbf{min}' \cdot n \cdot 0 \\
(\text{Prop. 5.3.1}) &\to_k \mathbf{min}' \cdot \mathbf{min} \cdot n \cdot 0 \\
(\text{Th. 5.2.1}) &\to_{c'} \mathbf{d}(0)(\mathbf{pred}(\mathbf{min} \cdot n \cdot (\mathbf{pred}0)))(n) \\
&\to_1 n
\end{aligned}$$

We take $c = k + c' + 3$. Now assume that the statement holds for $n' \leq m$.

$$\begin{aligned}
\mathbf{min} \cdot n \cdot m + 1 &= F \cdot \mathbf{min}' \cdot n \cdot m + 1 \\
&\to_k \mathbf{min}' \cdot \mathbf{min} \cdot n \cdot m + 1 \\
&\to_{c'} \mathbf{d}(m+1)(\mathbf{pred}(\mathbf{min} \cdot n \cdot (\mathbf{pred}m + 1)))(n) \\
&\to_1 \mathbf{pred}(\mathbf{min} \cdot n \cdot (\mathbf{pred}m + 1)) \\
&\to_1 \mathbf{pred}(\mathbf{min} \cdot n \cdot m) \\
(\text{I.H.}) &\to_{c(m+1)} \mathbf{pred}(n - m) \\
&\to_1 n - m - 1
\end{aligned}$$

We have that $k + c_1 + 3 + c(m+1) = c(m+2)$. □

Proposition 5.4.2. $\mathsf{ACT} \vdash \exists c \forall n, m \in \mathbb{N}[\mathbf{times} \cdot n \cdot m \to_{c(m+1)(n+2)} n \times m]$.

Proof. This is done by induction on m. If $m = 0$:

$$\begin{aligned}
&\mathbf{times} \cdot n \cdot 0 = F \cdot \mathbf{times}' \cdot n \cdot 0 \\
(\text{Prop. 5.3.1}) &\to_k \mathbf{times}' \cdot \mathbf{times} \cdot n \cdot 0 \\
(\text{Th. 5.2.1}) &\to_{c_1} \mathbf{d}(0)(\mathbf{d}(\mathbf{pred}0)(\mathbf{plus}(\mathbf{times} \cdot n \cdot (\mathbf{pred}0))n)(y))(0) \\
&\to_1 0
\end{aligned}$$

We take $c = \max(k + c_1 + 4, c_2)(n + 2)$, with c_2 coming from **plus**:

$$\begin{aligned}
\mathbf{times} \cdot n \cdot m + 1 &= F \cdot \mathbf{times}' \cdot n \cdot m + 1 \\
\text{(Prop. 5.3.1)} &\to_k \mathbf{times}' \cdot \mathbf{times} \cdot n \cdot m + 1 \\
\text{(Th. 5.2.1)} &\to_{c_1} \mathbf{d}(m+1)(\mathbf{d}(\mathbf{pred}m + 1)(\mathbf{plus}(\mathbf{times} \cdot n \cdot (\mathbf{pred}m + 1))n)(n))(0) \\
&\to_1 \mathbf{d}(\mathbf{pred}m + 1)(\mathbf{plus}(\mathbf{times} \cdot n \cdot (\mathbf{pred}m + 1))n)(n) \\
&\to_1 \mathbf{d}(m)(\mathbf{plus}(\mathbf{times} \cdot n \cdot (\mathbf{pred}m + 1))n)(n) \\
&\to_1 \mathbf{plus}(\mathbf{times} \cdot n \cdot (\mathbf{pred}m + 1))n \\
&\to_1 \mathbf{plus}(\mathbf{times} \cdot n \cdot m)n \\
\text{(I.H.)} &\to_{c\cdot(m+1)(n+1)} \mathbf{plus}(n \times m)n \\
\text{(Prop. 5.4.1)} &\to_{c_2(n+1)} n \times (m+1)
\end{aligned}$$

We have that $k + c_1 + c_2(n+1) + 4 + c(m+1)(n+2) \leq c(n+2) + c(m+1)(n+2) = c(m+2)(n+2)$. □

Proposition 5.4.3. $\mathsf{ACT} \vdash \exists c \forall n, m \in \mathbb{N}[\mathbf{div} \cdot n \cdot m \to_{c(m+2)\lceil \frac{n}{m} \rceil} \lceil \frac{n}{m} \rceil]$.

Proof. This is done by induction on m. If $m = 1$:

$$\begin{aligned}
\mathbf{div} \cdot n \cdot 1 &= F \cdot \mathbf{div}' \cdot n \cdot 0 \\
\text{(Prop. 5.3.1)} &\to_k \mathbf{div}' \cdot \mathbf{div} \cdot n \cdot 1 \\
\text{(Th. 5.2.1)} &\to_{c_1} \mathbf{d}(\mathbf{pred}1)(\mathbf{d}(\mathbf{min} \cdot n \cdot 1)(\mathbf{succ} \cdot (\mathbf{div} \cdot (\mathbf{min} \cdot n \cdot 1) \cdot 1))(1))(n) \\
&\to_1 \mathbf{d}(0)(\mathbf{d}(\mathbf{min} \cdot n \cdot 1)(\mathbf{succ} \cdot (\mathbf{div} \cdot (\mathbf{min} \cdot n \cdot 1) \cdot 1))(1))(n) \\
&\to_1 n
\end{aligned}$$

We take $c = \max(2c_2, k + c_1 + 5)$, with c_2 coming from **min**:

$$
\begin{aligned}
\mathbf{div}\cdot n\cdot m+1 &= F\cdot\mathbf{div}'\cdot n\cdot m+1\\
\text{(Prop. 5.3.1)}\ &\to_k \mathbf{div}'\cdot\mathbf{div}\cdot n\cdot m+1\\
\text{(Th. 5.2.1)}\ &\to_{c_1} \mathbf{d}(\mathbf{pred}m+1)(\mathbf{d}(\mathbf{min}\cdot n\cdot m+1)(\mathbf{succ}\cdot(\mathbf{div}\cdot(\mathbf{min}\cdot n\cdot m+1)\cdot m+1))(1))(n)\\
&\to_1 \mathbf{d}(m)(\mathbf{d}(\mathbf{min}\cdot n\cdot m+1)(\mathbf{succ}\cdot(\mathbf{div}\cdot(\mathbf{min}\cdot n\cdot m+1)\cdot m+1))(1))(n))\\
&\to_1 \mathbf{d}(\mathbf{min}\cdot n\cdot m+1)(\mathbf{succ}\cdot(\mathbf{div}\cdot(\mathbf{min}\cdot n\cdot m+1)\cdot m+1))(1)\\
\text{(Prop. 5.4.1)}\ &\to_{c_2(m+2)} \mathbf{d}(n-(m+1))(\mathbf{succ}\cdot(\mathbf{div}\cdot(\mathbf{min}\cdot n\cdot m+1)\cdot m+1))(1)\\
&\to_1 \mathbf{succ}\cdot(\mathbf{div}\cdot(\mathbf{min}\cdot n\cdot m+1)\cdot m+1)\\
\text{(Prop. 5.4.1)}\ &\to_{c_2(m+2)} \mathbf{succ}\cdot(\mathbf{div}\cdot(n-(m+1))\cdot m+1)\\
&\to_1 \mathbf{succ}\cdot(\mathbf{div}\cdot(\mathbf{min}\cdot n\cdot m+1)\cdot m+1)\\
\text{(I.H.)}\ &\to_{c\cdot(m+3)\lceil\frac{n-(m+1)}{m+1}\rceil} \mathbf{succ}\cdot(\lceil\frac{n-(m+1)}{m+1}\rceil)\\
&\to_1 \lceil\frac{n-(m+1)}{m+1}\rceil+1=\lceil\frac{n}{m+1}\rceil
\end{aligned}
$$

We have that $k+c_1+2c_2(m+2)+5+c\cdot(m+3)\lceil\frac{n-(m+1)}{m+1}\rceil \le c(m+3)+c\cdot(m+3)\lceil\frac{n-(m+1)}{m+1}\rceil = c(m+3)(\lceil\frac{n-(m+1)}{m+1}\rceil+1) = c(m+3)\lceil\frac{n}{m+1}\rceil$. □

Proposition 5.4.5. *Let* $\min(x,y)$ *denote the minimum of* x *and* y*. Then,*

$$\mathsf{ACT}\vdash \exists c\forall n,m\in\mathbb{N}[\mathbf{e}_{c(\min(n,m)+1)}z],$$

where $z=1$ *when* n *and* m *are equal and* $z=0$ *otherwise.*

Proof. This is done by induction on $n+m$. If $n+m=0$:

$$
\begin{aligned}
\mathbf{e}\cdot 0\cdot 0 &= F\cdot\mathbf{e}'\cdot 0\cdot 0\\
\text{(Prop. 5.3.1)}\ &\to_k \mathbf{e}'\cdot\mathbf{e}\cdot 0\cdot 0\\
\text{(Th. 5.2.1)}\ &\to_{c'} \mathbf{d}(0)(\mathbf{d}(0)(\mathbf{e}\cdot(\mathbf{pred}0)\cdot(\mathbf{pred}0))(0))(\mathbf{d}(0)(0)(1))\\
&\to_1 \mathbf{d}(0)(0)(1)\\
&\to_1 1
\end{aligned}
$$

If we take $c=k+c'+4$ then $\mathsf{e}\cdot 0\cdot 0\to_{c\cdot(1)} n+m$

Now assume that the statement holds for $n+m$. For $n+m+1$ we can have $n+1$ and m or n and $m+1$. Both cases work similarly so we'll only show one of them:

$$
\begin{aligned}
&\mathbf{e}\cdot n+1\cdot m = F\cdot\mathbf{e}'\cdot n+1\cdot m\\
(\text{Prop. 5.3.1}) &\to_k \mathbf{e}'\cdot\mathbf{e}\cdot n+1\cdot m\\
(\text{Th. 5.2.1}) &\to_{c'} \mathbf{d}(n+1)(\mathbf{d}(m)(\mathbf{e}\cdot(\mathbf{pred}n+1)\cdot(\mathbf{pred}m))(0))(\mathbf{d}(m)(0)(1))\\
&\to_1 \mathbf{d}(m)(\mathbf{e}\cdot(\mathbf{pred}n+1)\cdot(\mathbf{pred}m))(0)\\
&\to_1 \mathbf{e}\cdot(\mathbf{pred}n+1)\cdot(\mathbf{pred}m)\\
(\text{I.H.}) &\to_c (\min(n,m-1)+1)z
\end{aligned}
$$

We have that $k+c'+4+c(\min(n,m-1)+1) = k+c'+4+c\min(n+1,m) = c(\min(n+1,m)+1)$. □

Proposition 5.4.4. $\mathsf{ACT}\vdash\exists c\forall n,m\in\mathbb{N}[\mathbf{exp}\cdot n\cdot m\to_{c(n+2)\sum_{i=0}^{m-1}(n^i+1)} n^m]$.

Proof. We use induction. If $m=0$:

$$
\begin{aligned}
&\mathbf{exp}\cdot n\cdot 0 = F\cdot\mathbf{exp}'\cdot n\cdot 0\\
(\text{Prop. 5.3.1}) &\to_k \mathbf{exp}'\cdot\mathbf{exp}\cdot n\cdot 0\\
(\text{Th. 5.2.1}) &\to_{c_1} \mathbf{d}(0)(\mathbf{times}\cdot n\cdot(\mathbf{exp}\cdot n\cdot(\mathbf{pred}0)))(1)\\
&\to_1 1
\end{aligned}
$$

Now for $m+1$:

$$
\begin{aligned}
&\mathbf{exp}\cdot n\cdot m+1 = F\cdot\mathbf{exp}'\cdot n\cdot m+1\\
(\text{Prop. 5.3.1}) &\to_k \mathbf{exp}'\cdot\mathbf{exp}\cdot n\cdot m+1\\
(\text{Th. 5.2.1}) &\to_{c_1} \mathbf{d}(m+1)(\mathbf{times}\cdot n\cdot(\mathbf{exp}\cdot n\cdot(\mathbf{pred}m+1)))(1)\\
&\to_1 \mathbf{times}\cdot n\cdot(\mathbf{exp}\cdot n\cdot(\mathbf{pred}m+1))\\
&\to_1 \mathbf{times}\cdot n\cdot(\mathbf{exp}\cdot n\cdot m)\\
(\text{I.H.}) &\to_{c(n+2)\sum_{i=0}^{m-1}(n^i+1)} \mathbf{times}\cdot n\cdot n^m\\
(\text{Prop. 5.4.2}) &\to_{c_2(n+2)(n^m+1)} n^{m+1}
\end{aligned}
$$

We have that there exists a c such that $k+c_1+2+c_2(n+2)(n^m+1)+c(n+2)\sum_{i=0}^{m-1}(n^i+1) = c(n+2)\sum_{i=0}^{m}(n^i+1)$. □

Proposition 5.5.1. *When s is a sequence of natural numbers of length n, then*

$$\mathsf{ACT}\vdash\exists c[\mathbf{len}\cdot s\to_{c\times n} n].$$

Proof. We will show this with induction on n. Assume that $n = 0$ (and thus $s = \mathbf{p}00$). Then,

$$\begin{aligned}
&\mathbf{len} \cdot s = F \cdot \mathbf{len}' \cdot s \\
(\text{Prop. 5.3.1}) &\to_k \mathbf{len}' \cdot \mathbf{len} \cdot s \\
(\text{Th. 5.2.1}) &\to_{c'} \mathbf{d}(\mathbf{p}_0 s)(\mathbf{succ}(\mathbf{len} \cdot (\mathbf{p}_1 s)))(0) \\
&\to_1 \mathbf{d}(s_0)(\mathbf{succ}(\mathbf{len} \cdot (\mathbf{p}_1 s)))(0) \\
&\to_1 0
\end{aligned}$$

Now assume that it works for lists of at most length n. Then for an s with $lh(s) = n + 1$:

$$\begin{aligned}
&\mathbf{len} \cdot s = F \cdot \mathbf{len}' \cdot s \\
(\text{Prop. 5.3.1}) &\to_k \mathbf{len}' \cdot \mathbf{len} \cdot s \\
(\text{Th. 5.2.1}) &\to_{c'} \mathbf{d}(\mathbf{p}_0 s)(\mathbf{succ}(\mathbf{len} \cdot (\mathbf{p}_1 s)))(0) \\
&\to_1 \mathbf{d}(s_0)(\mathbf{succ}(\mathbf{len} \cdot (\mathbf{p}_1 s)))(0) \\
&\to_1 \mathbf{succ}(\mathbf{len} \cdot (\mathbf{p}_1 s)) \\
&\to_1 \mathbf{succ}(\mathbf{len} \cdot ([s_1, \ldots, s_{n-1}])) \\
(\text{I.H.}) &\to_{cn} \mathbf{succ}(n) \\
&\to_1 n + 1
\end{aligned}$$

If we set $c = 4 + k + c'$ then the proposition follows. □

Proposition 5.5.2. *When s is a sequence of natural numbers of length n, then*

$$\mathsf{ACT} \vdash \exists c \forall m \leq n[\mathbf{ind} \cdot s \cdot m \to_{c \times m} s_m].$$

Proof. We will show this with induction on n. Assume that $n = 0$. Then,

$$\begin{aligned}
&\mathbf{ind} \cdot s \cdot 0 = F \cdot \mathbf{ind}' \cdot s \cdot 0 \\
(\text{Prop. 5.3.1}) &\to_k \mathbf{ind}' \cdot \mathbf{index} \cdot s \cdot 0 \\
(\text{Th. 5.2.1}) &\to_{c'} \mathbf{d}(0)(\mathbf{ind} \cdot (\mathbf{p}_1 s) \cdot \mathbf{pred}0)(\mathbf{p}_0 s) \\
&\to_1 \mathbf{p}_0 s \\
&\to_1 s_0
\end{aligned}$$

Now assume that it works for indexes up to n.

$$
\begin{aligned}
\mathbf{ind} \cdot s \cdot 0 &= F \cdot \mathbf{ind}' \cdot s \cdot n + 1 \\
(\text{Prop. 5.3.1}) &\to_k \mathbf{ind}' \cdot \mathbf{ind} \cdot s \cdot n + 1 \\
(\text{Th. 5.2.1}) &\to_{c'} \mathbf{d}(n+1)(\mathbf{ind} \cdot (\mathbf{p}_1 s) \cdot (\mathbf{pred} n + 1))(\mathbf{p}_0 s) \\
&\to_1 \mathbf{ind} \cdot (\mathbf{p}_1 s) \cdot (\mathbf{pred} n + 1) \\
&\to_1 \mathbf{ind} \cdot ([s_1, \ldots, s_n]) \cdot (\mathbf{pred} n + 1) \\
&\to_1 \mathbf{ind} \cdot ([s_1, \ldots, s_n]) \cdot (n) \\
(\text{I.H.}) &\to_{cn} s_{n+1} \\
&\to_1 s_0
\end{aligned}
$$

If we set $c = 3 + k + c'$ then the proposition follows. □

11.2 Proof of Lemma 6.1.3

Here we will present the proof of the following lemma:

Lemma. *When $t \to_k u$, by always applying the axiom for the leftmost combinator that can be rewritten, then at every reduction-step, nothing that is duplicated contains two identical subterms that appeared by a single-step duplication, so $\ell(u) \leq (k+1)\ell(t)$.*

To make it easier for the reader, we repeat the basic axioms here:

B.1 $\mathbf{k}xy \to_1 x$

B.2 $\mathbf{s}xyz \to_1 xz(yz)$

B.3 $\mathbf{p}_0(\mathbf{p}xy) \to_1 x$

B.4 $\mathbf{p}_1(\mathbf{p}xy) \to_1 y$

B.5 $\mathbf{p}(\mathbf{p}_0x)(\mathbf{p}_1x) \to_1 x$

B.6 $\mathbf{succ}x \to_1 Sx$

B.7 $\mathbf{pred}Sx \to_1 x$

B.8 $\mathbf{pred}0 \to_1 0$

B.9 $\mathbf{d}(Sx)yz \to_1 y$

B.10 $\mathbf{d}0yz \to_1 z$

In the following, we will use the abbreviation of $x \in B$ when x is a term that is as a term on the left side of the arrow of one of the basic axioms (a redex). So $\mathbf{k}xy \in B$ for all x, y, etc.

We apply the axiom for the leftmost combinator that can be rewritten (we use a leftmost-outermost rewriting strategy). This can be formalized with the following axioms. We adapt axioms A.1 until A.3. For axiom A.4 and A.5 we will use the following rules:

- When $x \cdot z \notin B$ and $x \to_1 y$ then $x \cdot z \to_1 y \cdot z$
- When z is in normal form, $z \cdot x \notin B$ and $x \to_1 y$ then $z \cdot x \to_1 z \cdot y$.

These rules cannot be formalized within ACT, but they are stricter than A.4 and A.5, so rewriting by using these axioms can be done within ACT.

We will now prove the lemma, by going over all possible terms for which the situation of the lemma can occur.

Proof. The first observation that we make, is that the only combinator that will duplicate subterms is the **s**-combinator. So we will only need to show the following: assume that we have $t \to_1 t'$, where t had the subterm a and t' the subterm b, with $a \to_1 b$ by applying the axiom for an **s** in a (so a subterm of a is doubled). Then for further reduction steps, we will never encounter terms t'' and t''' such that $t'' \to_1 t'''$ by reducing $\mathbf{s}xyb \to_1 xb(yb)$, for some x and y. In other words: once we've copied subterms we will not copy them again with a single step.

We will show this by going over all possible situations for which we can have that $a \to_1 b$.

We observe that when a is the leftmost subterm of t, then b is the leftmost subterm of t' and then we will never end up with $\mathbf{s}xyb$ somewhere in the derivation, so we're already done.

We have the following two other possibilities: either $t = x \cdot a$ for some x, or $t = x \cdot y$ for some x and y.

$t = x \cdot a$ First assume that $t = x \cdot a$. Because we used the leftmost rewriting strategy for $t \to_1 t'$, x has to be in normal form and $x \cdot a \notin B$. Since x is in normal form, the only way to change the expression $x \cdot b$ is when $x \cdot b \in B$. We only have that $x \cdot a \notin B$ and $x \cdot b \in B$ with the axioms of pairing or the successor and predecessor axioms.

$t = x \cdot y$ Now assume that $t = x \cdot y$. Here we also have to consider several cases. First, assume that a is a subterm of y. We have three possibilities. $t = x \cdot (z \cdot a)$ for some z, $t = x \cdot (a \cdot z)$ for some z or $t = x \cdot (z_1 \cdot z_2)$ for some z_1 and z_2, where a is either a subterm of z_1 or z_2.

$t = x \cdot (z \cdot a)$ We consider the first possibility: $t = x \cdot (z \cdot a)$. Then we need that x and z are in normal form, $x \cdot (z \cdot a) \notin B$ and $z \cdot a \notin B$. Could it be that $x \cdot (z \cdot b) \in B$? This is only possible when we have $\mathbf{p}(\mathbf{p}_0 b)(\mathbf{p}_1 a) \to_1 \mathbf{p}(\mathbf{p}_0 b)(\mathbf{p}_1 b) \to b$. What if $x \cdot (z \cdot b) \notin B$ and $z \cdot b \in B$? This is only the case when we have that z is either the successor or predecessor combinator, because in all the other cases we would have that $z \cdot a \in B$ also. The other option is that $x \cdot (z \cdot b)$ is in normal form and then we're done.

$t = x \cdot (a \cdot z)$ Now the second possibility: $t = x \cdot (a \cdot z)$. Because we used the leftmost-outermost rewriting strategy for $t \to_1 t'$, we have that $x \cdot (a \cdot z) \notin B$. We then have that $x \cdot (b \cdot z) \in B$ only when we apply one of the pairing axioms. If $x \cdot (b \cdot z) \notin B$, we can have that $b \cdot z \in B$, but then we will not double b with an **s**-combinator. We can rewrite z but if $x \cdot (b \cdot z) \notin B$ then $x \cdot (b \cdot z') \notin B$ for any z'. Again, if $b \cdot z' \in B$ for some z' then we will also not duplicate b.

$t = x \cdot (z_1 \cdot z_2)$ For the third possibility, $t = x \cdot (z_1 \cdot z_2)$ with a a subterm of $z_1 \cdot z_2$, we need that x is in normal form and $x \cdot (z_1 \cdot z_2) \notin B$. Rewriting $(z_1 \cdot z_2)$ cannot change what is in x (so x will remain in normal form). If $x \cdot y \notin B$ (or $x \cdot (z_1 \cdot z_2) \notin B$), then only $x \cdot y' \in B$ for one of the pairing axioms or the successor/predecessor cases. The only option for b to double is then by rewriting $z_1 \cdot z_2$. If a is a subterm of z_1, then this is the same as considering $x' \cdot y'$ with a a subterm of x', since x will remain in normal form. We will consider this case later.

Assuma that a is a subterm of z_2. Since x will stay in normal form, no matter what $z_1 \cdot z_2$ will reduce to, if we want to duplicate b then it will have to be done by rewriting $z_1 \cdot z_2$. So this is the same as considering $x' \cdot y'$ with a a subterm of y', as we are considering now.

We can go trough all possibilities for what z_2 could look like. Since x is in normal form, it will remain the same. As mentioned before, rewriting $z_1 \cdot z_2$ into some term z' will not enable us to rewrite $x \cdot z'$ unless x is the successor or predecessor combinator. We can then only consider rewriting $z_1 \cdot z_2$ and view it as $x' \cdot y'$ with a a subterm of y'. This is similar to the case of $t = x \cdot y$. We can then again go over the possibilities for what z_2

looks like. But the term t has a finite length and we can continue to apply this reasoning. So eventually we will have to end up with a situation that was previously discussed.

Now the only thing left to do is to consider $t = x \cdot y$ with a a subterm of x. We can assume that $x \cdot y \notin B$. Here we have two possibilities: $t = (z \cdot a) \cdot y$ for some z, or $t = (z_1 \cdot z_2) \cdot y$ for some z_1 and z_2.

$t = (z \cdot a) \cdot y$ For the first possibility, we need that z is in normal form and $z \cdot a \notin B$. Then $z \cdot b \in B$ only happens for pairing, successor and predecessor. If $(z \cdot b) \cdot y \in B$ (but $(z \cdot a) \cdot y \notin B$), then this only happens when we had $\mathbf{p}(\mathbf{p}_0 a)(\mathbf{p}_1 b) \to_1 \mathbf{p}(\mathbf{p}_0 b)(\mathbf{p}_1 b) \to_1 b$.

$t = (z_1 \cdot z_2) \cdot y$ The second possiblity, $t = (z_1 \cdot z_2) \cdot y$, has two sub-possibilities: a can either be a subterm of z_1 or of z_2.

a a subterm of z_1 We have two options for when a is a subterm of z_1 (when we assume that a is not the leftmost subterm of t): $((w \cdot a) \cdot z_2) \cdot y$ and $((w \cdot v) \cdot z_2) \cdot y$.

$((w \cdot a) \cdot z_2) \cdot y$ We have that w is in normal form, and that all the subterms of $((w \cdot a) \cdot z_2 \cdot y$ are not in B. We have that $w \cdot b \in B$ only for pairing, predecessor or successor. We have considered $(w \cdot b) \cdot z_2 \in B$ before (this happens only for $\mathbf{p}(\mathbf{p}_0 a)(\mathbf{p}_1 b) \to_1 \mathbf{p}(\mathbf{p}_0 b)(\mathbf{p}_1 b) \to_1 b$). The case when $((w \cdot b) \cdot z_2) \cdot y \in B$ but not with a only happens with the $\mathbf{d}$-combinator.

$((w \cdot v) \cdot z_2) \cdot y$ Now consider $((w \cdot v) \cdot z_2) \cdot y$, where a is either a subterm of w or v. When $((w \cdot v) \cdot z_2) \cdot y \notin B$ with the subterm a, then also not when we reduce a to b. If we consider rewriting $(w \cdot v) \cdot z_2$, then this is the same as considering $(z_1' \cdot z_2') \cdot y'$, as we are doing now.

a a subterm of z_2 Finally, we will look at the possibilities for when a is a subterm of z_2. We have that z_1 then needs to be in normal form.

$(z_1 \cdot (a \cdot w)) \cdot y$ The only option for when $(z_1 \cdot (b \cdot w)) \cdot y \in B$ is with pairing and $b = \mathbf{p}_0$. The case for $z_1 \cdot (b \cdot w) \in B$ is the same as $x' \cdot (a \cdot z')$. The possibility of $b \cdot w \in B$ is also considered there.

$(z_1 \cdot (w \cdot a)) \cdot y$ Again, the only possiblity for when $(z_1 \cdot (w \cdot a)) \cdot y \in B$ is for pairing $(\mathbf{p}(\mathbf{p}_0 a)(\mathbf{p}_1 b) \to_1 \mathbf{p}(\mathbf{p}_0 b)(\mathbf{p}_1 b) \to_1 b)$. The case for $z_1 \cdot (w \cdot b) \in B$ is as $x' \cdot (z' \cdot a)$ as we considered before.

$(z_1 \cdot (w \cdot v)) \cdot y$ Also $(z_1 \cdot (w' \cdot v')) \cdot y \in B$ only for the pairing axioms. For considering $z_1 \cdot (w' \cdot v')$ we can look at the previously covered $x' \cdot (z_1' \cdot z_2')$ with a a subterm of $z_1' \cdot z_2'$.

□