Leibniz Universität Hannover

Fakultät für Elektrotechnik und Informatik

Institut für Theoretische Informatik

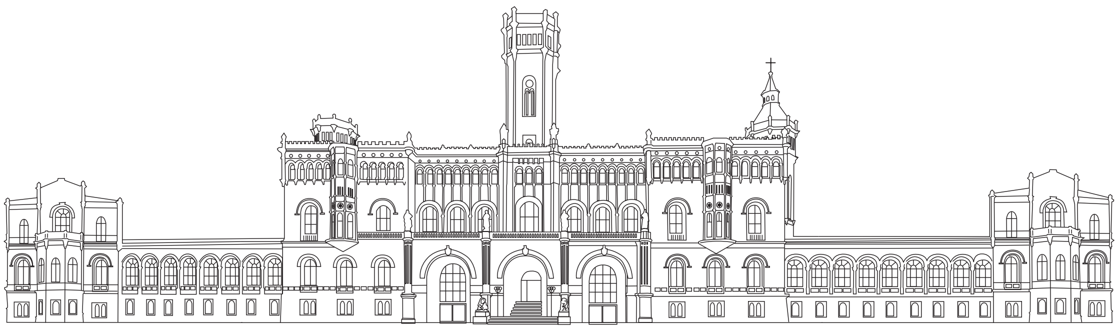
Decidability for real-valued computation

Tobias Brockmeyer

Matriculation number 10011858

Master Thesis

October 24, 2022



Prüfer:

Prof. Dr. Heribert Vollmer

PD Dr. Arne Meier

Betreuer:

Timon Barlag

Sabrina Alexandra Gaube

Erklärung der Selbstständigkeit

Hiermit versichere ich, die vorliegende Masterarbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet zu haben. Die Arbeit hat in gleicher oder ähnlicher Form noch keinem anderen Prüfungsamt vorgelegen.

Lehrte, 24. Oktober 2022

Contents

1 Introduction

In classical complexity theory, Turing machines have been the preferred mathematical model for physical computers since they were invented by Alan Turing in 1936 [Tur37]. They are unaffected by minor changes to their definition (e.g. multi-tape Turing machines can be simulated by single-tape Turing machines with only quadratic slow-down) and are therefore well-suited for the study of decidability and complexity questions. Since physical computers are finite systems based on digital circuits, it is reasonable to base mathematical models such as Turing machines and boolean circuits on a binary alphabet.

A binary alphabet is, of course, sufficient to represent arbitrary rational numbers. While the representation's length is, in theory, logarithmic with respect to the precision and absolute value of the number, actual systems use a standard encoding with a fixed length. For instance, integers are stored using either a 32-bit or 64-bit representation, depending on the specific system. Therefore, it is also reasonable to treat numbers as atomic (therefore constant) parts of mathematical models of computation. From a numerical point of view, this facilitates the analysis of algorithms because arithmetic operations such as addition and multiplication become a base operation of the machine model. This is supported by the fact that the complexity of arithmetic operations is almost never the bottleneck of an algorithm's performance.

In 1989, Blum, Shub and Smale invented another model of computation which deals with this problem [BSS89]. Instead of only allowing finite alphabets, their machines (called BSS machines) use the elements of algebraic rings as their alphabet. Turing machines can now be seen as a special case of BSS machines where the underlying ring is finite.

However, the more interesting cases are BSS machines working over either $\mathbb{Z}$, $\mathbb{Q}$, $\mathbb{R}$ or $\mathbb{C}$. The main concern of this thesis is to study the computational power of BSS machines (especially for machines working over $\mathbb{R}$) in comparison to Turing machines. The results are then transferred to algebraic circuits, a model for parallel computation based on BSS machines and boolean circuits.

In Chapter 2, we will formally define BSS machines over arbitrary rings as well as algebraic circuits along with an introduction to some algebraic foundations.

The 3rd chapter deals with encodings of polynomials and BSS machines to ultimately prove that (just as it is the case for Turing machines) there exists a universal BSS machine (i.e. a BSS machine that can simulate any given BSS machine). Another surprising result will be that BSS machines over $\mathbb{R}$ are able to decide any problem that has a discrete input set (i.e. all problems that can be stated for Turing machines including the halting problem).

We then proceed by studying space restrictions for BSS machines over $\mathbb{R}$ in Chapter 4. The main result of this chapter (as proven by Michaux in 1989) is that any BSS machine over $\mathbb{R}$ can be simulated by an equivalent machine using only linear space.

Next, the consequences of restricting a machine's access to irrational machine constants are investigated in Chapter 5. We use Meer and Ziegler's proof that more machine constants lead to strictly increasing computational power to show that non-uniform algebraic circuit families can be used to decide BSS-undecidable problems.

Finally, Chapter 6 deals with machines and circuits that are able to perform exact computations on real numbers, but are only allowed to process discrete inputs. We will study certain restrictions on the instruction set of BSS machines and the impact on their computational power on discrete inputs. The results of this chapter depend on whether the BSS machine is working over an ordered or unordered ring (in the latter case, the machine's decisions depend only on equality checks whereas in the former case, the machine may make decisions based on questions of the type "$x \geq 0$?"). We show that Turing machines are able to simulate BSS machines in a uniform way precisely if

"$x \geq 0$?" checks are not allowed. Moreover, for BSS machines that are not allowed to perform multiplications, there is even an efficient way to realize this simulation. This result can be transferred to algebraic circuits, ultimately proving that boolean circuits with unlimited fan-in can simulate algebraic circuits whose instruction sets include only additions and equality checks with constant blow-ups in size and depth.

2 Preliminaries

The main interest of this thesis lies in the computational power of machine models over algebraic rings. In the present section, some algebraic foundations concerning rings and transcendental field extensions are introduced. Additionally, we define the relevant machine models for this thesis and illustrate these definitions with basic examples.

2.1 Algebra

This section is split into three parts: Some important terminology concerning rings will be introduced in Section 2.1.1, followed by the definition of field extensions and related notions in Section 2.1.2 (c.f. [Bos20]). Lastly, in Section 2.1.3, we define (semi-)algebraic sets and discuss some important properties thereof.

2.1.1 Rings and polynomials

A *ring* is a set R with two operations $*\colon R \to R$ and $+\colon R \to R$ where

(R1) $(R, +)$ is an abelian group,

(R2) $*$ is associative and

(R3) the distributive laws $a * (b + c) = a * b + a * c$ and $(a + b) * c = a * c + b * c$ hold.

Throughout this thesis, we assume that the rings we work with are nontrivial commutative rings with unit (i.e. $*$ is commutative and there exists an element $1 \in R$ such that $1 * r = r$ for each $r \in R$). We will also write ab or $a \cdot b$ instead of $a * b$.

Definition 2.1. Let R be a ring. A set $J \subseteq R$ with the properties

(J1) $(J, +)$ is a group

(J2) $ar \in J$ for each $a \in J, r \in R$

is called an *ideal* of R. $\triangleleft$

One can easily check that the ring axioms also hold for ideals, showing that an ideal is a ring as well (although not necessarily a ring with unit, even if R is a ring with unit). For instance, $2\mathbb{Z} := \{2z \mid z \in \mathbb{Z}\}$ is an ideal without unit in $\mathbb{Z}$.

If R is a ring, then by $R[x]$ we denote the *polynomial ring* over R, i.e. the set of polynomials over R whose elements are of the form

$$f = a_0 + a_1x + a_2x^2 + a_3x^3 + \cdots + a_dx^d$$

where $a_i \in R, a_d \neq 0$. The *degree* of a polynomial f is written as $\deg f$ and defined by $\deg f = \max\{d \mid a_d \neq 0\}$.

The preceding definition of a polynomial ring can easily be extended to multiple variables by letting $R[x_1, \ldots, x_m] := R[x_1, \ldots, x_{m-1}][x_m]$. For $f \in R[x_1, \ldots, x_m]$, we say that the dimension of f is m and we write $\dim f = m$.

In the next section, we will define BSS machines over R. These machines are allowed to compute any polynomial function (or rational function in case R is a field) in a single step. If R is a field, a *rational function* over R is a function $f\colon R^m \to R$ that can be written as

$$f(x_1, \ldots, x_m) = \frac{g(x_1, \ldots, x_m)}{h(x_1, \ldots, x_m)},$$

where $g, h \in R[x_1, \ldots, x_m]$ and $h \neq 0$.

The composition of polynomial functions can easily be seen to be a polynomial function as well. The same holds true for rational functions. This will be crucial for the following discussion of BSS machines.

2.1.2 Field extensions

The following foundations of field extensions are mostly relevant for Chapter 5 where we will investigate the computational power of BSS machines with respect to the permitted number of real machine constants.

Definition 2.2. Let K be a field. A field $E \supseteq K$ is called a *field extension* of K. This is also written as E/K. $\lhd$

Definition 2.3. Let E/K be a field extension and $A \subseteq E$. Then,

$$K[A] := \bigcap\{R \mid R \text{ is a ring with } K \cup A \subseteq R\} \text{ and}$$

$$K(A) := \bigcap\{F \mid F \text{ is a field with } K \cup A \subseteq F\}. \qquad \lhd$$

Thus, $K[A]$ is the smallest ring containing both the elements from K and A. For instance, the polynomial ring $K[x]$ is the smallest ring that contains both K and x and we have $\mathbb{R}[i] = \mathbb{R}(i) = \mathbb{C}$. Since $K(A)$ is the smallest field containing both K and A and each field is also a ring, we always have $K[A] \subseteq K(A)$.

Definition 2.4. Let E/K be a field extension and $a \in E$. We say that a is *algebraic* over K if there exists a polynomial $f \in K[x] \setminus \{0\}$ such that $f(a) = 0$. Otherwise, a is called *transcendental* over K. E is called an *algebraic field extension* of K if each $a \in E$ is algebraic over K. Otherwise, E is called a *transcendental field extension* of K. $\lhd$

Definition 2.5. Let E/K be a field extension. A set $S \subseteq E$ is called *algebraically independent* over K if s is transcendental over $K(S \setminus \{s\})$ for each $s \in S$. $\lhd$

Definition 2.6. Let E/K be a field extension. An algebraically independent set $S \subseteq E$ is called a *transcendence basis* of E if $E/K(S)$ is algebraic. $|S|$ is called the *transcendence degree* of the field extension E/K. In case $K = \mathbb{Q}$, we write $\operatorname{trdeg} E$ for the transcendence degree of E/K. $\lhd$

If no field K is specified, we assume $K = \mathbb{Q}$. For instance, $\sqrt{2}$ is algebraic over $\mathbb{Q}$ since $\sqrt{2}$ is a root of $f(x) = x^2 - 2$. This implies that the transcendence degree of $\mathbb{Q}(\sqrt{2})$ is

0 because $\mathbb{Q}(\sqrt{2})/\mathbb{Q}$ is an algebraic field extension and therefore, $\varnothing$ is a transcendence basis of $\mathbb{Q}(\sqrt{2})$. Of course, the transcendence degree is only well-defined if for each field extension E/K, we have that any two transcendence bases S_1, S_2 of E have the same cardinality. The proof can be found in [Bos20, Section 7.1].

2.1.3 Algebraic sets

A natural requirement for machines over an algebraic ring is the possibility to take different paths of computation depending on how the input looks like. In this section, by looking at algebraic and semi-algebraic sets, we examine some candidates for sets that describe the inputs which take a specific path of computation.

Definition 2.7 [MP07, Definition 1.1]**.** Let $F \subseteq R[x_1, \dots, x_m]$. The set

$$\mathcal{V}(F) := \{x \in R^m \mid f(x) = 0 \text{ for all } f \in F\}$$

is called the *affine algebraic set* or *vanishing set* defined by F. A set $A \subseteq R^m$ is called algebraic if $A = \mathcal{V}(F)$ for some $F \subseteq R[x_1, \dots, x_m]$. $\lhd$

It is well known that finite unions as well as arbitrary intersections of algebraic sets are again algebraic. Furthermore, for each algebraic set A, there exist some $f_1, \dots, f_k \in R[x_1, \dots, x_m]$ such that $A = \mathcal{V}(f_1, \dots, f_k)$, i.e. F can be chosen to be finite [MP07, Remark 1.3].

In the setting of computation over a ring, we will want to allow checks like "$x \geq 0$?" if R is ordered. This yields a slightly modified version of algebraic sets, called semi-algebraic sets.

Definition 2.8 [Blu+98, page 50]**.** Let R be an ordered ring and $f, g \in R[x_1, \dots, x_m]$. We call $f(x) = g(x)$ a *polynomial equation* over R. In addition, the following are *polynomial inequalities* over R.

$$f(x) \neq g(x), f(x) < g(x), f(x) \leq g(x), f(x) > g(x), f(x) \geq g(x)$$

A set $A \subseteq R^m$ is called *basic semi-algebraic* over R if A is the set of elements in R^m that satisfy a finite system of polynomial equations or inequalities over R. A *semi-algebraic* set over R is a finite union of basic semi-algebraic sets over R. $\triangleleft$

We will also use the term *system of polynomial constraints* for a system of polynomial equations and inequalities. We can oberve that if $A \in R^m$ is basic semi-algebraic over R, then there exists a system S of polynomial constraints over R that defines A and contains only the following types of constraints.

$$f(x) = 0, f(x) \neq 0, f(x) < 0, f(x) \leq 0 \text{ where } f \in R[x_1, \ldots, x_m]$$

First note that the constraint $f(x) = g(x)$ is equivalent to the constraint $f(x) - g(x) = 0$ (this argument of course also holds for inequalities). The inequalities $f(x) > 0$ and $f(x) \geq 0$ can be replaced by $-f(x) < 0$ and $-f(x) \leq 0$.

We will write $\mathcal{V}(S)$ for the basic semi-algebraic set defined by the system of polynomial constraints S if there is no confusion with vanishing sets as in Definition 2.7.

Lemma 2.9. *Let $A \subseteq \mathbb{R}$ be a finite set which is semi-algebraic over some field extension $K = \mathbb{Q}(B) \subseteq \mathbb{R}$. Then, the field extension $K(A)/K$ is algebraic.*

Proof. Let $A = \{a_1, \ldots, a_k\} \subseteq \mathbb{R}$ where $a_1 < \cdots < a_k$. For each a_j, there is a system of polynomial constraints S over K such that $a_j \in \mathcal{V}(S)$. We can add constraints to S to force $a_{j-1} < s < a_{j+1}$ for each $s \in \mathcal{V}(S)$. Hence, for each $a_j \in A$, we have that $\{a_j\}$ is basic semi-algebraic over K.

Furthermore, the set $\{a_j\}$ is a closed subset of $\mathbb{R}$. We will now prove that each system S' of polynomial constraints over K with $\mathcal{V}(S') = \{a_j\}$ must contain at least one equality constraint $f(x) = 0$ for some $f \in K[x] \setminus \{0\}$ (or we can add an equality constraint without changing $\mathcal{V}(S')$).

The first observation is that it is not possible for S' to only consist of "$<$" constraints. This is because each set defined only by "$<$" constraints is open and each open subset of $\mathbb{R}$ is either empty or infinite. Suppose S' only contains "$<$" and "$\neq$" constraints. Each constraint $f(x) \neq 0$ only removes a finite number of elements (because f has only a finite number of roots) unless f is constant in which case the constraint produces either the empty set or can be dropped. Thus, it is insufficient to add finitely many "$\neq$" constraints in order to obtain the set $\{a_j\}$.

Therefore, S' must contain at least one equation (in which case $f(a_j) = 0$ for some f and thus, a_j is algebraic over K and we are done) or S' contains at least one constraint $f(x) \leq 0$. In the latter case, let $f_1 \leq 0, \dots, f_n \leq 0$ be all constraints in S' of the form $f(x) \leq 0$. If $f_i(a_j) \neq 0$ for each $1 \leq i \leq n$, then the constraints $f_1 \leq 0, \dots, f_n \leq 0$ can be replaced by $f_1 < 0, \dots, f_n < 0$ without changing the defined semi-algebraic set. However, this is impossible because this would define an infinite (or empty) set. Hence, $f_i(a_j) = 0$ for some i and therefore, a_j is algebraic over K. Thus, each $a_j \in A$ is algebraic over K and therefore, $K(A)/K$ is an algebraic field extension. □

We will see in Chapter 5 that there is a close relation between semi-algebraic sets and the halting sets of BSS machines.

For the remainder of this section, we will restrict ourselves to basic semi-algebraic sets that are definable by a system of linear inequalities with integral coefficients. We will call these sets *polytopes* as they are a generalization of polyhedras to higher dimensions in a geometric sense.

Definition 2.10. For $z \in \mathbb{Z}$, let

$$\text{size}(z) := |\text{bin}(z)|$$

denote the length of the binary representation for z. For rational numbers $y \in \mathbb{Q}$, set

$$\text{size}(y) = \min\left\{\text{size}(p) + \text{size}(q) \mid \frac{p}{q} = y \text{ and } p, q \in \mathbb{Z}\right\}.$$

The *size* of a vector $v = (v_1, \dots, v_k)$ with rational entries is defined as $\sum_{i=1}^{k} \text{size}(v_i)$. ◁

For simplicity, we will assume $\text{size}(z) = \log(|z|)$ for each $z \in \mathbb{Z}$.

Definition 2.11. A set $P \subseteq \mathbb{R}^m$ is called a *polytope* if

$$P = \{x \in \mathbb{R}^m \mid Ax \leq b \text{ and } A'x < b'\}$$

for some $A \in \mathbb{Z}^{N \times m}, A' \in \mathbb{Z}^{N' \times m}, b \in \mathbb{Z}^N, b' \in \mathbb{Z}^{N'}, N, N' \in \mathbb{N}$. ◁

Our goal is to prove that each nonempty polytope $P \subseteq \mathbb{R}^m$ defined by $Ax \leq b$ and $A'x < b'$ contains a rational point whose binary representation has a size polynomial in m and the size of the entries of A, A', b, b'. In other words, the solution's length does not depend on the number of inequalities that define P. Thus, for a fixed m, we can always find some $x \in P$ with a "small" binary representation as long as the coefficients in the defining inequalities are sufficiently small. We will need the following result.

Lemma 2.12 [Sch86, Corollary 7.1i]**.** *If a vector $b \in \mathbb{R}^m$ is a positive linear combination of vectors $a_1, \ldots, a_n \in \mathbb{R}^m$ (i.e. if there exist coefficients $x_i \geq 0$ such that $\sum_{i=1}^n x_i a_i = b$), then there exists a linearly independent subset $\{a_{i_1}, \ldots, a_{i_k}\} \subseteq \{a_1, \ldots, a_n\}$ such that b is a positive linear combination of $a_{i_1}, \ldots, a_{i_k}$.*

Proof. See [Sch86, Theorem 7.1]. □

Lemma 2.13 [Koi94, Theorem 5]**.** *Let $P \subseteq \mathbb{R}^m$ be a nonempty polytope defined by a system $Ax \leq b$ of N linear inequalities where the entries of A and b are integers of size L. Then, there is a rational point $x \in P$ such that* size(x) *is polynomial with respect to m and L.*

Proof. We will prove this lemma in multiple steps. In the first step, we will transform the system defining P into a system

$$A'x' = b', x'_i \geq 0, \qquad (*)$$

which has a solution if and only if P has a solution. For each variable x_i, we introduce two variables y_i and y'_i and replace each occurence of x_i by $y_i - y'_i$. This allows us to assign positive values to y_i and y'_i. We also add a slack variable to each equation, transforming $a_{j1}x_1 + \cdots + a_{jm}x_m \leq b_j$ into

$$a_{j1}(y_1 - y'_1) + \cdots + a_{jm}(y_m - y'_m) + s_j = b_j.$$

The matrix A' in $(*)$ is a matrix of size $N \times N + 2m$. Since the slack variables $s_1, \ldots, s_N$ that correspond to the N rightmost columns of A' form an identity matrix (because each

s_j appears in exactly one equation), A' has rank N. The second step is to transform the system from $(*)$ into a system

$$A''x'' = b'', x''_i \geq 0 \tag{$**$}$$

where A'' is an $N \times N$ matrix. Let $v_1, \ldots, v_{N+2m}$ be the column vectors of A'. We know that b' can be written as a positive linear combination

$$v_1 x'_1 + \cdots + v_{N+2m} x'_{N+2m}.$$

Thus, we can apply Lemma 2.12 to see that there is even a positive linear combination

$$v_{i_1} x'_{i_1} + \cdots + v_{i_N} x'_{i_N} = b'.$$

Hence, we set $x'_i = 0$ for each x'_i with $i \neq i_j$ for $j = 1, \ldots, N$ and immediately obtain a system as in $(**)$. Obviously, A'' has rank N as well. Therefore, we know from linear algebra (Cramer's rule, c.f. [Lan87, Theorem 4.1]) that $(**)$ has a unique solution

$$x'' = (x''_1, \ldots, x''_N)^T \text{ where } x''_i = \frac{\det(A''_i)}{\det(A'')}.$$

Here, A''_j denotes the matrix obtained from A'' by replacing the jth column by b''. Since A' contains N column vectors that belong to an identity matrix, we know that all but at most $2m$ columns of A'' contain presisely one 1 and further only 0's. By permutations of rows and columns, each of the matrices A'', A''_i can be brought into the form

$$B = \begin{pmatrix} C & 0 \\ D & I \end{pmatrix},$$

where I is an identity matrix and C is a square matrix of size $2m+1$. Hence, $\det(B)$ is the sum of at most $(2m+1)!$ terms d_j where each d_j is the product of at most $2m+1$ numbers of absolute value at most 2^L. Since all entries of B are integers, $\det(B)$ must be an integer as well. Therefore, we have

$$\begin{aligned} \text{size}(\det(B)) &\leq \text{size}\big((2m+1)! \cdot 2^{L(2m+1)}\big) \\ &= \log\big((2m+1)! \cdot 2^{L(2m+1)}\big) \\ &\leq \log\big((2m+1)^{2m+1}\big) + \log(2^{L(2m+1)}\big) \\ &= (2m+1)\log(2m+1) + L(2m+1). \end{aligned}$$

□

Finally, we are able to prove the following result which generalizes the preceding lemma to polytopes where the defining inequalities may also be strict. This lemma will play a key role for the simulation of BSS machines by Turing machines.

Lemma 2.14 [Koi94, Theorem 6]**.** *Let $P \subseteq \mathbb{R}^m$ by a nonempty polytope defined by a system φ of N inequations of the form*

$$Ax \leq b, A'x < b',$$

where the entries of A, A', b, b' are integers of size L. Then, there is a rational point $x \in P$ such that $\operatorname{size}(x)$ *is polynomial with respect to m and L.*

Proof. Let P' be the polytope of $\mathbb{R}^{m+1}$ defined by the system φ' obtained from φ by the following transformations:

- add a new variable K and the constraint $K \geq 1$,
- replace the subsystem $Ax \leq b$ by $Ay \leq Kb$,
- replace the subsystem $A'x < b'$ by $A'y \leq Kb' - 1$

For instance, the inequality $4x_1 + 2x_2 < 10$ will be replaced by $4y_1 + 2y_2 - 10K \leq -1$. Intuitively, each solution (x_1, x_2) where $|4x_1 + 2x_2 - 10| < 1$ can be transformed into a solution for $4y_1 + 2y_2 - 10K \leq -1$ by choosing K sufficiently large and multiplying x_1, x_2 by K. Hence, the new system has a solution $y = (y_1, \ldots, y_m, K) \in \mathbb{R}^{m+1}$ if and only if φ has a solution $(x_1, \ldots, x_m) \in \mathbb{R}^m$. Because $P \neq \varnothing$, we know that such a solution exists and by Lemma 2.13, there exists a point $y = (y_1, \ldots, y_m, K) \in P'$ with size polynomial in L and m. Let

$$x = \left(\frac{y_1}{K}, \ldots, \frac{y_m}{K}\right).$$

Clearly, $x \in P$ and since $\operatorname{size}(y)$ is sufficiently small, $\operatorname{size}(x)$ is polynomial in L and m. □

2.2 Computation over a ring

As pointed out in the introduction, a Turing machine can be seen as a machine that works over the field $\mathbb{Z}_2$. The following definition aims to generalize this to an arbitrary field (or, even more general, an arbitrary ring). In the following, let R be a ring.

In the classical case, a Turing machine reads either a 0 or a 1 at each step. It is then allowed to either keep the value (therefore applying the identity function) or replace it by another value. Any of these functions can be seen as a polynomial function over $\mathbb{Z}_2$. For instance, flipping a bit can be interpreted as evaluating the function $f(x) = x + 1$.

In the following, we generalize this notion to an arbitrary ring by allowing our machines to store any element of a fixed ring R in each of their cells.

2.2.1 Infinite-dimensional spaces

Classically, when considering Turing machines, any finite sequence of symbols is allowed as an input, leading to the set Σ^* where Σ is an alphabet. Analogously, any finite sequence of elements from R is allowed as an input for BSS machines over R. We denote the set of these sequences by R^∞ where

$$R^\infty = \bigcup_{n \in \mathbb{N}} R^n.$$

Let $x \in R^\infty$. By $|x|$, we denote the *length* of x and if $x \in R^n$, we set $|x| = n$.

The working tape of classical Turing machines is unbounded in two directions. This property is simulated by the *bi-infinite direct sum space over* R which we shall denote by R_∞. Elements of R_∞ have the form

$$x = (\dots, x_{-2}, x_{-1}, x_0 . x_1, x_2, \dots),$$

where $x_i \in R$ for all $i \in \mathbb{Z}$ and $x_k = 0$ for all but finitely many $k \in \mathbb{Z}$. Intuitively, the marker "." between x_0 and x_1 can be thought of as the head position of a Turing machine with x on its working tape.

The construction of R^∞ and R_∞ can also be found in the book of Blum et al. [Blu+98, Section 3.1]. For the remainder of this section, let $x = (\dots, x_{-2}, x_{-1}, x_0.x_1, x_2, \dots)$.

Definition 2.15 [Blu+98, Section 3.1]**.** Let $h\colon R^m \to R$ be a polynomial function of degree d over R. Then h defines a polynomial function $\hat{h}\colon R_\infty \to R$ of degree d and dimension m on R_∞ by letting $\hat{h}(x) = h(x_1, \dots, x_m)$ for each $x \in R_\infty$.

If $g_i\colon R^m \to R$ $(i = 1, \dots, m)$ are polynomial functions of maximum degree d over R, we define the polynomial function of degree d and dimension m on R_∞ defined by the g_i by letting

$$g(x) = (\dots, x_{-2}, x_{-1}, x_0.g_1(x), g_2(x), \dots, g_m(x), x_{m+1}, x_{m+2}, \dots). \qquad \lhd$$

Therefore, for instance, if we apply the polynomial functions

$$g_1(x_1, x_2) = x_1 + 3 \text{ and } g_2(x_1, x_2) = x_2 - 2x_1$$

to the input $x = (\dots, 0, 0.1, 4, 1, 5, 9, \dots)$, we get

$$g(x) = (\dots, 0, 0.\mathbf{4}, \mathbf{2}, 1, 5, 9, \dots).$$

With the above definition, rational functions on R_∞ can be definied analogously by replacing “polynomial function” by “rational function” everywhere in the definition.

A BSS machine over R will have a finite set of instructions (more precisely, polynomial or rational functions on its state space R_∞). Having only access to functions as in the previous definition, the machine can only access a constant number of registers (this is due to the finite degree of any function in the instruction set of the machine). To circumvent this issue, we need two additional *shift functions* which are defined as follows.

Definition 2.16 [Blu+98]**.** The functions *shift left* $\sigma_\ell\colon R_\infty \to R_\infty$ and *shift right* $\sigma_r\colon R_\infty \to R_\infty$ are defined by letting, for all $i \in \mathbb{Z}$,

$$\sigma_\ell(x)_i = x_{i+1} \text{ and } \sigma_r(x)_i = x_{i-1}. \qquad \lhd$$

Applying a shift function resembles the movement of a Turing machine's head where σ_ℓ is a movement to the right and σ_r is a movement to the left. With these functions, our machines will now be able to access the whole space R_∞.

2.2.2 BSS machines

Definition 2.17 [CM99; Blu+98]**.** A BSS machine over R consists of an *input space* $\mathcal{I} = R^\infty$, an *output space* $\mathcal{O} = R^\infty$ and a *state space* $\mathcal{S} = R_\infty$, together with a connected directed graph whose nodes are labeled $1, \dots, N$.

The node labeled 1 is the *input node.* Associated with this node there is a next node $\beta(1)$ and the input map $g_I \colon \mathcal{I} \to \mathcal{S}$.

The node labeled N is the *output node.* It has no next nodes, once it is reached the computation halts, and the output map $g_O \colon \mathcal{S} \to \mathcal{O}$ places the result of the computation in the output space.

Each of of the remaining nodes has one of the following types:

1. *Computation nodes.* Associated with a node m of this type there are a next node $\beta(m)$ and a function $g_m \colon \mathcal{S} \to \mathcal{S}$. The g_m is a polynomial (or rational if R is a field) function on R_∞ in the sense of Definition 2.15.

2. *Branch nodes.* There are two nodes associated with a node m of this type: $\beta^+(m)$ and $\beta^-(m)$. The next node is $\beta^+(m)$ if $x_1 \geq 0$ and $\beta^-(m)$ if $x_1 < 0$.

3. *Shift nodes.* Associated with a node m of this type is a next node $\beta(m)$ and a function $g_m \colon \mathcal{S} \to \mathcal{S}$ where either $g_m = \sigma_\ell$ or $g_m = \sigma_r$.

The input map g_I transforms an input $(x_1, \dots, x_n) \in R^\infty$ into the state $(\dots, 0, \hat{n}.x_1, \dots, x_n, 0, \dots) \in R_\infty$ where $\hat{n}$ denotes a sequence of n 1s.

The output map g_O transforms a state $(\dots, 0, \hat{n}.x_1, x_2, x_3, \dots) \in \mathcal{S}$ into the output $(x_1, \dots, x_n) \in \mathcal{O}$.

Let $\mathcal{N} = \{1, \dots, N\}$ be the set of nodes of the machine M. Then, each *configuration* of M can be described by some $z \in \mathcal{N} \times \mathcal{S}$. The starting configuration of M is thus $z^0 = (1, g_I(x))$. Let $\mathcal{B}$ be the set of branch nodes of M. The machine works by generating the computation $z^0, z^1, z^2, \dots$ with $z^k = (\eta^k, x^k)$ for each $k \in \mathbb{Z}$ and

$$z^{i+1} = H(\eta^i, x^i),$$

where $x^k = (\dots, x^k_{-1}, x^k_0.x^k_1, \dots)$ and

$$H(\eta^k, x^k) = \begin{cases} (\beta^+(\eta^k), x^k), & \text{if } \eta^k \in \mathcal{B} \text{ and } x^k_1 \geq 0 \\ (\beta^-(\eta^k), x^k), & \text{if } \eta^k \in \mathcal{B} \text{ and } x^k_1 < 0 \\ (\beta(\eta^k), g_{\eta^k}(x^k)), & \text{otherwise.} \end{cases}$$

If for some k, $\eta^k = N$, then the machine halts and outputs $g_O(x^k)$. $\lhd$

Note that in the case of $\mathbb{R}$, the sequence $\hat{n}$ in the input and output function can be replaced by n (therefore only using one register instead of n registers). This is not possible for rings R with $\operatorname{char} R \neq 0$ (e.g. $R = \mathbb{Z}_2$) as they do not have enough elements to represent each $n \in \mathbb{N}$.

In the original definition by Blum, Smale and Shub in 1989 [BSS89], an additional type of node was used instead of shift nodes which allowed their machines to access the content of an arbitrary register by its address. The definition using shift nodes instead of these random access nodes is equivalent [Blu+98, Section 3.6].

The entries x_i of a machine's state $x \in \mathcal{S}$ are refered to as the registers of that machine throughout this thesis. Furthermore, each BSS machine M has access to only a finite number of machine constants over R (i.e. constants that are hard-wired in the description of M). These constants are exactly the coefficients in the polynomial (or rational) functions that are associated with the computation nodes of M.

If M is a BSS machine over R, the function $\Phi_M(x)\colon R^\infty \to R^\infty$ that maps each $x \in R^\infty$ to the output of M on input x is called the *input-output-map* of M. Moreover, let Ω_M be the *halting set* of M, i.e. the set of all $x \in R^\infty$ such that M halts on input x.

Definition 2.18 [Blu+98, Section 4.2]**.** Let M be a BSS machine over R and $x \in R^\infty$. We define $cost(x)$ to be the number of nodes traversed during the computation of M on input x. M works in time $f(n)$ if $cost(x) \leq f(n)$ for each input x with $|x| = n$. We say that M works in polynomial time if M works in time $p(n)$ for some polynomial p. $\lhd$

We end this section by presenting a simple example for a BSS machine over $\mathbb{R}$ that strictly follows our definition. This example should clarify how the shift operations can be used to simulate the behavior of a Turing machine.

Given an input $(s_1, s_2, \dots, s_n) \in \mathbb{R}^n$, our machine M will compute $s_1 + s_2 + \cdots + s_n$. Given this input, M will produce the initial state

$$(\dots, 0, n.s_1, \dots, s_n, 0, \dots).$$

Note that we use n instead of $\hat{n}$ for the input length since we are working over $\mathbb{R}$. As our machine can only access registers with positive coordinates (i.e. $x_1, x_2, \dots$), we first need to apply a right shift to get access to n. After the right shift, we proceed by adding x_2 and x_3 and storing the result in x_3 followed by a left shift. This way, the register x_2 always contains a sum of of the form $s_1 + s_2 + \cdots + s_k$ where $k \in \mathbb{N}$ is the number of iterations. Thus, we get the following sequence of states:

$$\begin{gathered}
(\dots, 0.n, s_1, s_2, s_3, s_4, \dots) \\
(\dots, 0, n.s_1, s_1 + s_2, s_3, s_4, \dots) \\
(\dots, 0, n, s_1.s_1 + s_2, s_1 + s_2 + s_3, s_4, \dots) \\
\vdots \\
(\dots, s_1 + \cdots + s_{n-2}.s_1 + \cdots + s_{n-1}, s_1 + \cdots + s_n, 0, \dots)
\end{gathered}$$

This is close to the actual algorithm. However, the machine needs an additional piece of information to know when to stop. This information is provided by n. We can prevent the n from being shifted to the left by swapping it with the value at the second coordinate each time before we apply the left shift. We thus get the following sequence of states:

$$(\dots, 0.n, s_1, s_2, s_3, \dots)$$
$$(\dots, 0.n, s_1, s_1 + s_2, s_3, \dots)$$
$$(\dots, 0.s_1, n, s_1 + s_2, s_3, \dots)$$
$$(\dots, 0, s_1.n, s_1 + s_2, s_3, \dots)$$

The value n can now be used to count the number of remaining additions. Initially, we need to decrement n because we only need $n-1$ additions for the sequence $(s_1, \dots, s_n)$. The machine then decrements the counter again at the start of each iteration.

Combining all these ideas finally yields a machine as shown in Figure 2.1. The nodes at the right side have the only purpose to output x_2. For this, the machine needs to produce the state

$$(\dots, 1.x_2, \dots).$$

This can be done by writing the value 1 to x_1 and then performing a shift left operation.

2.2.3 Algebraic circuits

In classical complexity theory, several models of parallel computation such as parallel RAMs and boolean circuits have been studied. As in the case of Turing machines, boolean circuits can be seen as operating over the field $\mathbb{Z}_2$. A more general definition of circuits that allows inputs from an arbitrary ring instead of the fixed field $\mathbb{Z}_2$ is presented in this section.

Definition 2.19 [Blu+98, page 349]**.** An *algebraic circuit* over R is an acyclic directed graph with n *input nodes* labeled $1, \dots, n$ and m *output nodes* labeled $1, \dots, m$.

Input nodes have indegree 0. Output nodes have indegree 1 and outdegree 0. The remaining nodes have one of the following types:

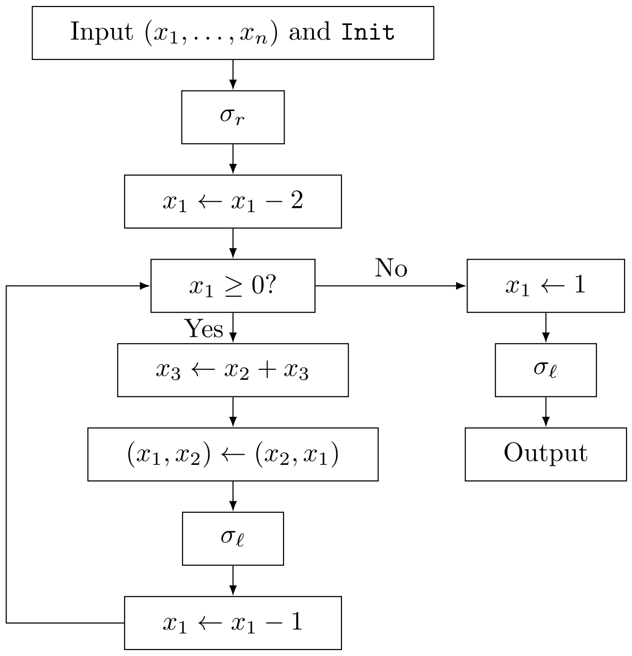


Figure 2.1: The BSS machine M for implementing the addition of array elements

1. *Arithmetic nodes.* Associated with a node of this type is one of the binary operations $+, -, *$ (and division $\div$ in case R is a field) of R. Arithmetic nodes have indegree 2.

2. *Constant nodes.* Nodes of this type are labeled with an element of R and have indegree 0.

3. *Sign nodes.* These nodes have indegree 1.

For an algebraic circuit C, the *size* of C is the number of nodes in C. The *depth* of C is the length of the longest path from some input node to some output node. We will also refer to the nodes of C as *gates*.

Let C be an algebraic circuit with n input gates and m output gates. We inductively associate a function $f_g \colon R^n \to R$ with each gate g of C. Let $x = (x_1, \dots, x_n)$.

- If g is the ith input gate of C, then $f_g(x) = x_i$.

- If g is a constant gate labeled $a \in R$, then $f_g(x) = a$.

- If g is an arithmetic node labeled $\circ \in \{+, -, *, \div\}$ with predecessors h_1 and h_2, then $f_g(x) = f_{h_1}(x) \circ f_{h_2}(x)$.
- If g is a sign node with predecessor h, then $f_g(x) = \begin{cases} 1, & f_h(x) \geq 0 \\ 0, & \text{otherwise} \end{cases}$
- If g is an output node with predecessor h, then $f_g(x) = f_h(x)$.

The circuit C computes the function $f_C \colon R^n \to R^m$ defined by

$$f_C(x) = (f_{g_1}(x), \dots, f_{g_m}(x)),$$

where $g_1, \dots, g_m$ are the output nodes of C. $\lhd$

By convention, we let $x \div y := 0$ for $y = 0$ to make sure that $f_C(x)$ is always defined. The preceding definition only allows inputs of a specific, fixed length. We have already seen that BSS machines over a ring R are allowed to process any inputs from R^∞. Therefore, if we want to investigate the parallel complexity of problems over R^∞, we need to expand our definition of algebraic circuits in order to allow them to process the same inputs as other machines.

Definition 2.20. A *circuit family over R* is a sequence $C = (C_n)_{n \in \mathbb{N}}$ of algebraic circuits where C_n has n input nodes for each $n \in \mathbb{N}$. Let f^n be the function computed by C_n. We say that C computes the function $f_C \colon R^\infty \to R^\infty$ where

$$f_C(x) = f^{|x|}(x)$$

for each $x \in R^\infty$. $\lhd$

The above definition of a circuit family is *non-uniform* in a sense that there is no finite description of a circuit family as it may consist of possibly infinitely many unrelated circuits. A solution to solve this problem in the boolean context is to study only *uniform* circuit families, i.e. circuit families whose structure can be computed by a Turing machine. The Turing machine then gives a finite description of the circuit family.

Classically, the complexity results for uniform circuit families are quite different from non-uniform circuit families. While the uniform versions of NC and AC are contained in P, this is not the case if we allow non-uniform circuit families as they can even solve the unary halting problem which is undecidable by Turing machines (see Section 5.3 for more details).

Naturally, we can apply the same distinction to families of algebraic circuits. Later in this thesis, we will show that non-uniform circuit families over $\mathbb{R}$ are stronger than machines over $\mathbb{R}$ (just as in the boolean case), although the argument is more complicated than in the boolean case.

Definition 2.21 [Blu+98, page 353]**.** A family $C = (C_n)_{n\in\mathbb{N}}$ of algebraic circuits over R is said to be *uniform* if there exists a BSS machine M over R that, on input (n, i), outputs the description of the ith gate of C_n. It is said to be $\mathrm{P}_\mathbb{R}$-uniform if M works in polynomial time. $\lhd$

There are many ways to describe a node in an algebraic circuit. One way is to associate a node labeled n with a tuple (n, t, g_1, g_2) where n is the node's label, t is the node's type (for arithmetic nodes, t also encodes the associated operation) and g_1, g_2 are the node's predecessors. For a node without any predecessors, we set $g_1 = g_2 = 0$. For nodes with exactly one predecessor, set $g_2 = 0$.

Definition 2.22 [Blu+98, page 353]**.** NC^i_R is the class of sets $A \subseteq R^\infty$ such that there is a family of circuits that computes the characteristic function of A and has size polynomial in n and depth $\mathcal{O}(\log^k n)$. The union of the NC^i_R is denoted by NC_R. $\lhd$

We may define a uniform version of the above classes by additionally requiring the circuit family to be $\mathrm{P}_\mathbb{R}$-uniform. We denote the uniform version of NC^i_R by $U_{\mathrm{P}_\mathbb{R}}$-$\mathrm{NC}^i_R$. By NC^i and AC^i (without the subscript R), we refer to the parallel complexity classes in the boolean context.

For instance, any sequential algorithm for the problem of matrix multiplication for $n \times n$ matrices can be proven to require at least n^2 steps, but the problem requires only logarithmic time (i.e. depth) in the setting of algebraic circuits. It can easily be verified that the circuit shown in Figure 2.2 can be generalized to the multiplication of $n \times n$

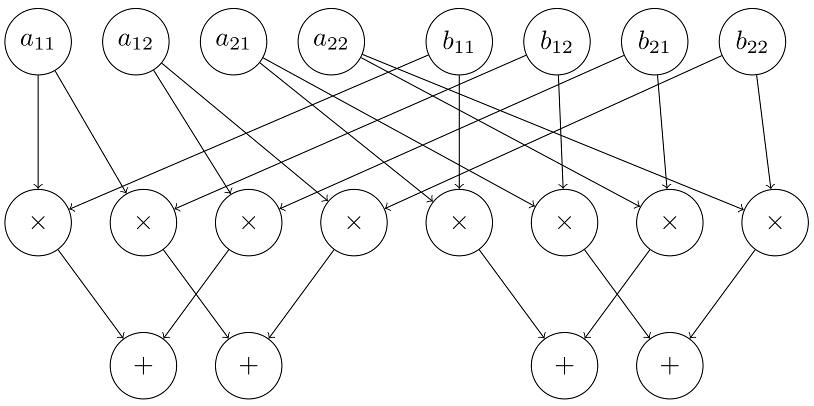


Figure 2.2: An algebraic circuit for the multiplication of two 2×2 matrices.

matrices with logarithmic depth by arranging the addition gates in the bottom layer in a binary tree, thus proving that matrix multiplication is in $U_{\mathrm{P}_{\mathbb{R}}}$-$\mathrm{NC}^1_R$ [Vol99, Theorem 5.3].

3 Universal machines

An important observation about Turing machines is that any Turing machine can be represented using only a single positive integer (the underlying encoding was presented by Gödel in 1931 [Göd31]). This allows the construction of a universal Turing machine, i.e. a Turing machine that, on input (M, x), outputs $\Phi_M(x)$. Moreover, this allows the following formulation of the (discrete) halting problem which is known to be undecidable.

Problem:	$\mathbb{H}$
Input:	A Turing machine M and an input x
Question:	Does M halt on input x?

This leads to the question whether it makes sense to define a similar halting problem for machines over R. Indeed, we will be able to construct a universal BSS machine and therefore, it makes sense to define a real halting problem $\mathbb{H}_\mathbb{R}$.

Problem:	$\mathbb{H}_\mathbb{R}$
Input:	A BSS machine M over $\mathbb{R}$ and an input x
Question:	Does M halt on input x?

The aim of this section is to construct a universal machine over $\mathbb{R}$. As a first step, we need to find an encoding $\pi(M)$ of a BSS machine M by a finite sequence of real values. Over $\mathbb{N}$, it is possible to find a recursive bijection $\mathbb{N}^\infty \to \mathbb{N}$. Hence, we can encode finite sequences of integers by a single integer and, by recursion, obtain a way to encode multiple sequences of integers. This is, however, not possible for $\mathbb{R}$.

In general, a sequence $(x_1, \ldots, x_n) \in \mathbb{R}^n$ cannot easily be distinguished from a shorter sequence that ends with one or more 0s. Therefore, the input function in the definition of machines over $\mathbb{R}$ adds the prefix n to inputs of length n. This idea can be generalized to sequences of real numbers [Blu+98, page 80]. We can encode k sequences of real numbers by

$$(k, n_1, \ldots, n_k, a_1, \ldots, a_k),$$

where $k, n_i \in \mathbb{N}$ and $a_i \in \mathbb{R}^{n_i}$ for $i = 1, \ldots, k$.

Lemma 3.1. *Each BSS machine over $\mathbb{R}$ has a recursive representation $\pi(M) \in \mathbb{R}^\infty$.*

The proof also works for any other ring with characteristic 0. For other rings, the proof can be adjusted by using the sequence $\hat{n}$ (the sequence of n 1s) instead of n at the respective places.

Proof. [Blu+98, page 81] To describe a machine M over $\mathbb{R}$, it suffices to represent the nodes of the underlying graph. Let $\mathcal{N}$ be the set of nodes in the graph of M. Each $\eta \in \mathcal{N}$ can be represented by a tuple

$$\eta = (n, \beta_\eta, t_\eta, g_\eta),$$

where $n \in \{1, \ldots, N\}$ is the label of η, $\beta_\eta = (\beta_\eta^+, \beta_\eta^-)$ is the pair of next nodes of η, t_η is the type (input, output, branch, computation or shift) of η and g_η is the function associated with η. By setting unused values to 0 (e.g. the function g_η is not required if η is a branch node), we can represent each type of nodes with the same set of values. In particular, we assume that the next node is always stored in β_η^+ if η has only one next node.

All these values except g_η can trivially be encoded by a single positive integer. Let m be the dimension of M, i.e. the maximum dimension of the functions associated with the nodes of M. To encode g_η, we can store the sequence of coefficients of g_η using a lexicographic order on the monomials of dimension m. The encoding $\pi(M)$ of the machine is the sequence of encodings of its nodes. □

An easy way to get a lexicographic ordering on the monomials of dimension m is to order the monomials by their total degree, i.e.

$$1, \underbrace{x_1, \dots, x_m}_{\text{total degree 1}}, \underbrace{x_1^2, x_1x_2, \dots, x_1x_m, x_2x_1, \dots, x_m^2}_{\text{total degree 2}}, x_1^3, \dots$$

For instance, the polynomial function $2x_1^2 + \pi x_2 + \sqrt{2}x_4 + 4$ of dimension 4 can be encoded by the sequence

$$(4, 0, \pi, 0, \sqrt{2}, 2).$$

To simulate a given BSS machine M over R, the universal machine has to evaluate the polynomials that appear in the computation of M. We will do so by using a *universal polynomial evaluator* (UPE) as subroutine, i.e. a machine that, given a polynomial $f \in R[x_1, \dots, x_m]$ and $x \in R^m$, outputs $f(x)$. This can easily be generalized to rational functions if R is a field. One procedure to implement a UPE is to compute each monomial's value separately and sum up the monomials' values. This can be done in polynomial time and with a constant amount of registers apart from the representation of f.

We will now investigate the special case where $R \subseteq \mathbb{Q}$, i.e. the machine has to evaluate a polynomial $f \in \mathbb{Q}[x_1, \dots, x_m]$ (again, the following observations also hold for rational functions). The important difference in this case is that each such f can be represented using only a single positive integer r. Thus, f can be stored in a single register by a machine M over $\mathbb{R}$. However, M is unable to access the digits of r directly due to its restriction to rational functions. The following result shows that M can access the digits of r in polynomial time and with only a constant amount of auxiliary registers.

Lemma 3.2. *Let $r \in \mathbb{R}, r \geq 0$ and let $\dots r_2r_1r_0.r_{-1}r_{-2}\dots$ be the b-adic representation of r, i.e. $r_i \in \{0, 1, \dots, b-1\}$ and*

$$r = \sum_{k \in \mathbb{Z}} r_k \cdot b^k.$$

For each $z \in \mathbb{Z}$, there exists a machine M_b over $\mathbb{R}$ without shift nodes that, on input (r, z), outputs r_z.

Proof. Since $r \in \mathbb{R}$, there exists a maximum $k \in \mathbb{Z}$ such that $r_k \neq 0$. Hence, $\frac{r}{b^k} < 1$ for some $k \in \mathbb{Z}$. M_b works by first dividing r by b until it obtains some $r' < 1$. The count of divisions performed this way is stored in a separate register and will be denoted by n. If $n \leq z$, output 0.

In the next step, the machine repeats the following steps $n - z - 1$ times (we assume that r' is stored in the register x_1 at this point):

1. Multiply x_1 by b.

2. While $x_1 \geq 1$, set $x_1' \leftarrow x_1 - 1$.

After this sequence of operations, the register x_1 holds the value

$$\ldots 0.r_z r_{z-1} r_{z-2} \ldots$$

in b-adic representation. Now, to obtain r_z, the machine only needs to multiply x_1 by b and then subtract 1 from x_1 until $x_1 < 1$. By counting the number of subtractions, M_b can compute r_z. □

This allows machines over $\mathbb{R}$ to store the configuration of a Turing machine in a single register. By storing the head position in a separate register, a machine over $\mathbb{R}$ can thus simulate any Turing machine in constant space (use one register to store the tape and simulate the machine as in the discrete case).

Corollary 3.3. *Each set $A \subseteq \mathbb{N}$ is decidable over $\mathbb{R}$.*

Proof. [MZ06] Let

$$a_n = \chi_A(n) = \begin{cases} 1, & n \in A \\ 0, & n \notin A \end{cases}$$

for $n \in \mathbb{N}$. The sequence $a = (a_n)_{n \in \mathbb{N}}$ can be encoded into a real number $R_a \in \mathbb{R}$ by letting

$$R_a = \sum_{k \in \mathbb{N}} a_k \cdot 2^{-k}.$$

On input z, the machine performs the instruction $x_1 \leftarrow R_a$ and proceeds as in the proof of Lemma 3.2 to access a_z. It accepts if it finds the value 1 and rejects otherwise. □

By letting $A = \mathbb{H}$ in the predecing result, it follows that the discrete halting problem is decidable by a BSS machine over $\mathbb{R}$. In fact, a single real constant is enough for BSS machines over $\mathbb{R}$ to decide the halting problem. We will show in Chapter 5 that the power of machines over $\mathbb{R}$ strictly increases with the number of permitted constants, i.e. for each $n \in \mathbb{N}$, there is a set $A \subseteq \mathbb{R}$ such that A is undecidable by a machine over $\mathbb{R}$ with n constants, but there is a machine with $n + 1$ constants that decides A.

Going back to UPEs, we get another useful result from Lemma 3.2 that will be important in Chapter 4.

Corollary 3.4. *Let $f \in \mathbb{Q}[x_1, \ldots, x_m]$ and $x \in \mathbb{R}^m$. There exists a machine M over $\mathbb{R}$ such that*

1. *M outputs $f(x)$ on input (f, x),*
2. *M uses only $m + k$ registers where $k \in \mathbb{N}$ and*
3. *M does not modify the input registers.*

Proof. Immediate from Lemma 3.2 and the discussion on page 27. □

A universal machine over $\mathbb{R}$ will have to simulate the computation of any machine M over $\mathbb{R}$. We can use the UPE subroutine (cf. page 27) to simulate the effect of a computation node of M and are thus ready to describe a universal machine.

Theorem 3.5. *There exists a machine U over $\mathbb{R}$ such that, when input a pair $(\pi(M), x)$, U outputs $\Phi_M(x)$.*

Note that a machine over $\mathbb{R}$ can swap the values of some registers x_i, x_j with $1 \leq i < j$ by evaluating the polynomial map

$$(x_1, \ldots, x_{i-1}, x_i, x_{i+1}, \ldots, x_j) \leftarrow (x_1, \ldots, x_{i-1}, x_j, x_{i+1}, \ldots, x_i)$$

of dimension j. Operations of this kind are referred to as swap operations during the following proof.

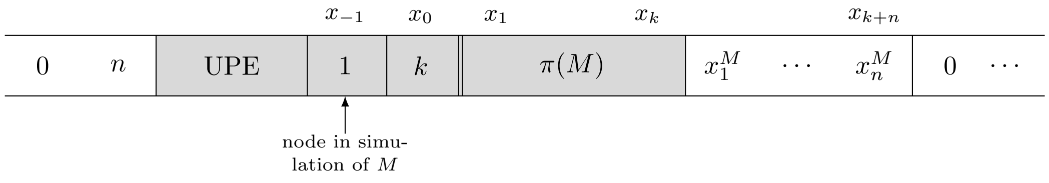


Figure 3.1: Initial state of U during the simulation of M. Control information is stored in the gray registers. The white registers contain the state of M during the simulation.

Proof. We give a description of U. On input $(\pi(M), x)$, U will enter the initial state

$$(\ldots, 0, n, k.m_1, \ldots, m_k, x_1, \ldots, x_n, 0, \ldots),$$

where $|x| = n$ and $|\pi(M)| = k$ (this can be done by extracting n and k from the input, see the discussion on page 26). U will use most of its registers to store the current state of the simulation. It needs the following additional registers (see Figure 3):

- k registers to store $\pi(M)$
- one register to store k
- one register to store the current node of M in the simulation
- a constant number of registers for the UPE subroutine

The state shown in Figure 3 can easily be reached from the initial state by a sequence of shift and swap operations.

U can only "see" the registers $x_1, \ldots, x_m$ for some fixed m (due to the finite dimension of U). However, the number of control registers for U is not constant (due to $\pi(M)$). This can be solved by adding further control registers. These registers contain the coordinate p of the leftmost register that stores $\pi(M)$ and space to copy instructions from $\pi(M)$. This enables U to systematically access $\pi(M)$ using shift operations and reduces the number of remaining control registers to a constant, enabling U to access these registers without shift or swap operations. U accesses $\pi(M)$ and the state of M by performing a sequence of shift operations and tracking the offset using p.

A step in the simulation of M works as follows:

- look up the next operation in $\pi(M)$ based on the current node η stored at x_{-1}
- perform an operation based on the current type of node $(*)$
- determine the next node in the computation of M and update the state of U accordingly

The specific operations performed by U in $(*)$ depend on the type of node that U has to simulate:

- If η is a shift node, U enters a shift node and then swaps a constant number of registers to make sure the control registers of U remain in the same position as before.
- If η is a branch node, U checks whether the register storing x_1 is ≥ 0 and updates the node register accordingly. x_1 can be accessed by a sequence of shift and swap operations.
- If η is a computation node, copy the starting coordinate of the polynomial's representation in $\pi(M)$ to one of the registers of the UPE subroutine. The subroutine can use shift and swap operations to access the relevant registers of the simulation and evaluates the components one by one. U can use temporary auxiliary registers to store the results that were generated by the UPE before writing them back to the simulation registers.

If M reaches an output node, U computes the respective output state, then enters the output node of U. Clearly, U computes $\Phi_M(x)$. □

In classical recursion theory, the existence of a universal Turing machine implies that the discrete halting problem is undecidable. The same argument can now be used to show that the real halting problem is undecidable over $\mathbb{R}$.

Corollary 3.6. $\mathbb{H}_{\mathbb{R}}$ *is undecidable over* $\mathbb{R}$.

Proof. Assume that $\mathbb{H}_\mathbb{R}$ is decidable over $\mathbb{R}$. Let M be a machine over $\mathbb{R}$ that, on input x, decides if $(x, x) \in \mathbb{H}_\mathbb{R}$. Construct a machine M' that simulates M, halts if the result is 0 and goes into an infinite loop if the result is 1. Clearly, this is a contradiction:

- If $(\pi(M'), \pi(M')) \in \mathbb{H}_\mathbb{R}$, then M' goes into an infinite loop on input $\pi(M')$. Thus, $(\pi(M'), \pi(M')) \notin \mathbb{H}_\mathbb{R}$. Contradiction.
- If $(\pi(M'), \pi(M')) \notin \mathbb{H}_\mathbb{R}$, then M' halts on input $\pi(M')$. Thus, $(\pi(M'), \pi(M')) \in \mathbb{H}_\mathbb{R}$. Contradiction.

Hence, $\mathbb{H}_\mathbb{R}$ must be undecidable. $\square$

The next chapters will give a deeper understanding of the causes of decidability and undecidability over $\mathbb{R}$ by investigating certain restrictions of BSS machines over $\mathbb{R}$.

4 Real computation with restricted space

It is a natural question to ask how the computational power of BSS machines changes if they have access to only a limited amount of space. More precisely, we are interested in the number of registers a machine over $\mathbb{R}$ needs to compute a function $f(x)$ with respect to the input length $|x|$.

It was proven by Michaux in 1989 [Mic89] that the restriction of BSS machines to polynomial space does not actually restrict their computational power over $\mathbb{R}$. In fact, the following proof by Michaux shows that linear space is enough to compute any computable function over $\mathbb{R}$.

On page 17 it was already pointed out that each machine over $\mathbb{R}$ can only have a finite number of machine constants in its description (the coefficients that appear in the functions in the machine's computation nodes). This is a key premise for the argument underlying the proof.

Theorem 4.1. *Let $f\colon \mathbb{R}^m \to \mathbb{R}$ be a function that can be computed by a BSS machine over $\mathbb{R}$. Then, there exists a BSS machine M' over $\mathbb{R}$ without shift nodes such that $\Phi_M(x) = \Phi_{M'}(x)$ for all $x \in \mathbb{R}^m$.*

A machine over $\mathbb{R}$ without shift nodes is often referred to as a finite-dimensional machine over $\mathbb{R}$. This is due to the fact that it is unable to access more than a constant amount of registers. The original definition of a finite-dimensional machine over $\mathbb{R}$ can be found in the book of Blum et al. [Blu+98, page 40].

Proof. Consider the entries of a state $x \in \mathcal{S}$ during the calculation of M. Initially, they hold the values

$$(\dots, 0, n.s_1, \dots, s_n, 0, \dots)$$

where $s = (s_1, \dots, s_n)$ is the input to M. Clearly, the value at each coordinate can be expressed as a rational function on s.

At each step during the computation of M, the machine applies a rational function to some of its registers. Being compositions of rational functions, the values in the registers of M can be expressed by rational functions on s at each step of M. Let $c = (c_1, \dots, c_k)$ be the machine constants of M. We can transform the rational functions on s that describe the state space of M at a certain step of computation into rational functions on s and c by interpreting the c_i $(i = 1, \dots, k)$ as arguments of these functions. Hence, the values in the registers of M at a certain state $x \in \mathcal{S}$ can be interpreted as rational functions on s and c and can thus be written as

$$x_i = \frac{g(s_1, \dots, s_n, c_1, \dots, c_k)}{h(s_1, \dots, s_n, c_1, \dots, c_k)} \tag{$*$}$$

where x_i denotes the value at the ith coordinate in the state x of M. Let $\mathcal{N}$ be the set of nodes of M. A configuration $z = (\eta, x)$ of M with $\eta \in \mathcal{N}, x \in \mathcal{S}$ can therefore be represented by two integers. The first integer encodes η while the second one encodes the sequence

$$p = (p_l, \dots, p_r)$$

where each p_i is a rational function as in $(*)$ that has to be evaluated to obtain x_i. The coefficients of each p_i are in $\mathbb{Z}$ and thus, each p_i can be encoded as a single integer by a computable function. With the same argument, the whole list p can be represented by a single integer. We will write $\langle p \rangle$ for the integer that encodes p.

The idea for the machine M' is to store the rational function that needs to be evaluated in order to get one of the x_i instead of the actual values. M' has the following types of registers:

- n registers for storing the input (their values will remain constant during the computation of M')
- k registers for storing the machine constants

- one register for storing the state of M
- one register to implement the shift operations of M

The registers for the machine constants can easily be initialized by executing the instruction

$$(x_l, \dots, x_r) \leftarrow (c_1, \dots, c_k)$$

directly after g_I was executed where x_l is the leftmost such register and x_r the rightmost. The machine then produces the initial state

$$(\dots, 0.s, c, \langle p^0 \rangle, \sigma^0, \dots).$$

Here, p^0 denotes the sequence of rational functions that represent the initial state of M. M' works by updating the functions in p whenever M computes a function at one of its computation nodes.

The state of M' after T steps of computation looks like

$$(\dots, s, c, \langle p^T \rangle, \sigma^T, \dots).$$

σ^T is necessary to implement the shift operations of M. It starts with $\sigma^0 = 0$, is increased by 1 whenever M performs a shift right operation and decreased by 1 whenever M performs a shift left operation. By Lemma 3.2, M' can compute the values $\langle p_i \rangle$ for $i \in \mathbb{N}$ using only a constant amount of auxiliary registers.

For the branch operations, M' needs to evaluate the function stored in p_1^T for some $T \in \mathbb{N}$ during the computation. This can be realized by calling a UPE as constructed in Corollary 3.4 as a subroutine. This adds only a constant amount of additional registers to M'. Rational functions are handled analogously (the numerator and denominator can be stored separately). To check whether a rational function is 0, it suffices to check whether the numerator is 0.

When M' reaches an output node, it can use the UPE to restore the output value and compute the output state. Clearly, M' computes the same function as M. □

Note that the only possible coefficients in the functions associated with the computation nodes of M' are 1 and 0. For instance, a possible step in the calculation of M' could be as follows. Let $(s_1, \dots, s_n) \in \mathbb{R}^n$ be the input of M' and let η be a computation node of M with $g_\eta = \sqrt{2}x_3^2 + 4x_4 + 5$. If η is the only computation node of M, then $\sqrt{2}, 4$ and 5 are the only constants of M and M' will produce the initial state

$$(\dots, 0.s_1, \dots, s_n, 5, 4, \sqrt{2}, \langle p \rangle, \sigma, 0, \dots)$$

with $\sigma = 0, p = (p_0, \dots, p_n) \in \mathbb{Q}[\alpha_1, \dots, \alpha_{n+k}]^{n+1}, p_0 = n$ and $p_i = \alpha_i$ for $i \in \{1, \dots, n\}$. Here, $\sigma = 0$ indicates that the leftmost entry of the sequence stored in p refers to the register x_0 of M and α_i is a pointer to the value in the ith register of M'. If M performs a shift right operation, M' will increment σ to indicate that the leftmost entry of p now refers to x_1. Thus, after another shift right operation, σ holds the value 2 and the state of M would look like

$$(\dots, 0.0, n, s_1, \dots, s_n, 0, \dots).$$

Let $|p|$ denote the number of polynomials encoded by p. If M now enters the computation node η, then M' computes a new pointer expression to write to $p_{1-\sigma}$. Here, M' writes the result to p_{-1} because p_0 now represents the value at x_2. The polynomial expression

$$\sqrt{2}x_3^2 + 4x_4 + 5$$

will be transformed into a representation that can be stored into p. This is done by first replacing the coefficients by the respective address references, yielding

$$\alpha_{n+3}x_3^2 + \alpha_{n+2}x_4 + \alpha_{n+1}.$$

Afterwards, each occurance of x_i is replaced by the expressions in the respective register (this is done by looking at p). For instance, each occurance of x_3 will be replaced by the expression stored in $p_{3-\sigma}$. If the machine tries to access p_i for $i < 0$ or $i \geq |p|$, then the value can be treated as 0. This transforms the expression into

$$\alpha_{n+3}\alpha_1^2 + \alpha_{n+2}\alpha_2 + \alpha_{n+1}.$$

M' finishes the computation step by replacing p_{-1} by this expression. Since p_{-1} is not yet stored in p, the machine prepends this value to the list. For later computations, this value will be treated as p_0 (i.e. $p = (p_0, \dots, p_k)$ at each step during the computation of

M' for some $k \in \mathbb{N}$). This requires another change to σ since p_0 no longer stores the value x_2. The final state of M' after this sequence of operations is thus

$$(\ldots, 0.s_1, \ldots, s_n, 5, 4, \sqrt{2}, \langle p \rangle, 1, 0, \ldots),$$

where $p = (p_0, \ldots, p_{n+1})$ and

$$p_i = \begin{cases} \alpha_{n+3}x_3^2 + \alpha_{n+2}x_4 + \alpha_{n+1}, & \text{if } i = 0 \\ n, & \text{if } i = 1 \\ \alpha_{i-1}, & \text{otherwise.} \end{cases}$$

Let k denote the number of machine constants of M'. M' may output the value x_1 by evaluating the polynomial $f \in \mathbb{Q}[\alpha_1, \ldots, \alpha_{n+k}]$ stored at $p_{1-\sigma}$ on $(s_1, \ldots, s_n, 5, 4, \sqrt{2}) \in \mathbb{R}^{n+k}$ using the UPE.

Corollary 4.2. *Each function $f\colon \mathbb{R}^\infty \to \mathbb{R}^\infty$ that is computable by a BSS machine over $\mathbb{R}$ can be computed by a machine over $\mathbb{R}$ in linear space.*

Proof. By analyzing the proof of Theorem 4.1, it can be seen that we can build a machine M'' over $\mathbb{R}$ for the whole input space $\mathbb{R}^\infty$ just as in the case of a fixed input length with the only modification that we need to use a number of shift operations depending on $|x|$ each time we evaluate the value of a register (i.e. if the machine reaches a branch node or the output node). This, however, does not require any additional space and therefore M'' only needs the following space:

$$(\ldots, x_1, \ldots, x_n, c_1, \ldots, c_k, \mathcal{S}_M, \sigma_M, \ldots)$$

$\mathcal{S}_M$ denotes the sequence of polynomials representing the state of M. Hence, each BSS-computable function $f\colon \mathbb{R}^\infty \to \mathbb{R}^\infty$ can be computed by a machine over $\mathbb{R}$ in linear space. $\square$

5 Restrictions on machine constants

We have seen in Chapter 3 that a single machine constant is sufficient to give machines over $\mathbb{R}$ the power to decide any subset of $\mathbb{N}$ (or, more generally, every problem whose inputs can be represented using a discrete input set such as $\{0,1\}^*$). Without the use of real machine constants, however, discrete problems that are classically undecidable are undecidable even for BSS machines over $\mathbb{R}$. This is due to the fact that Turing machines are able to simulate BSS machines on discrete inputs if only rational machine constants are permitted. Hence, machines with one real constant are strictly stronger than machines without a real constant. Meer and Ziegler [MZ06] proved that this phenomenon appears in general if we allow additional constants, i.e. for each i, there is a problem that can be decided by a machine with i real constants, but not by a machine with $i-1$ real constants.

To prepare the proof, we will first discuss an algebraic characterization of semi-decidable sets in Section 5.1. Meer and Ziegler's result will be presented in Section 5.2 and an application thereof to algebraic circuits can be found in Section 5.3.

5.1 An algebraic characterization of semi-decidable sets

Lemma 5.1. *Let $A \subseteq \mathbb{R}^\infty$ be semi-decidable by a machine over $\mathbb{R}$ with i real machine constants. Then, there exist $c_1, \ldots, c_i \in \mathbb{R}$ such that $A = \bigcup_{n\in\mathbb{N}} A_n$ and all A_n are semi-algebraic over the field extension $K := \mathbb{Q}(c_1, \ldots, c_i)$.*

Proof. [Blu+98, Section 2.3] Let M be a machine with the machine constants $c_1, \ldots, c_i$

that semi-decides A and $\mathcal{N} = \{1, \ldots, N\}$ the set of nodes of M. By definition of M, we have $A = \Omega_M$ (recall that Ω_M is the halting set of M). For $T \in \mathbb{N}$, let Ω_T be the time-T halting set of M, i.e. the set of all $x \in \mathbb{R}^\infty$ such that M halts on input x after exactly T steps.

For $x \in \Omega_T$, let $\pi(x) = (1, \eta^1, \ldots, \eta^{T-1}, N)$ be the computation path of x (i.e. the nodes traversed during the computation of M on input x). Then,

$$\pi_T = \{\pi(x) \mid x \in \Omega_T\}$$

is the set of time-T halting paths. Clearly, π_T is finite for each $T \in \mathbb{N}$. By letting

$$\mathcal{V}_\pi = \{x \in \mathbb{R}^\infty \mid \pi(x) = \pi\},$$

we obtain the set of all inputs that take the computation path π (called the *path set* of π). We will now show that this set is basic semi-algebraic. Let z_j^t be a value that is stored in the jth register of M on input $x = (x_1, \ldots, x_n)$ after t steps of computation. Clearly, z_j^t can be written as a polynomial (or rational) function on $(x_1, \ldots, x_n)$, i.e. $z_j^t = f_j^t(x_1, \ldots, x_n)$. Furthermore, all coefficients of z_j^t are in the field $\mathbb{Q}(c_1, \ldots, c_i)$ since the only available coefficients for the polynomials associated with a computation node of M are $c_1, \ldots, c_i$. For some computation path $\pi = (\eta^0, \ldots, \eta^k)$, let

$$L_\pi = \{j \mid \eta_j \text{ is a branch node and } \eta_{j+1} = \beta^-(\eta_j)\}$$

and

$$R_\pi = \{j \mid \eta_j \text{ is a branch node and } \eta_{j+1} = \beta^+(\eta_j)\}.$$

Now, if $f_j^t \in K[x_1, \ldots, x_n]$ describes the value at the jth register after t steps of computation on input $x \in \mathbb{R}^n$, we can write the path set of π as

$$\mathcal{V}_\pi = \bigcup_{n \in \mathbb{N}} \{x \in \mathbb{R}^n \mid f_1^t(x) < 0 \text{ for each } t \in L_\pi \text{ and } f_1^t(x) \geq 0 \text{ for each } t \in R_\pi\},$$

proving that for each $n \in \mathbb{N}$, the set $\mathcal{V}_\pi^n := \mathcal{V}_\pi \cap \mathbb{R}^n$ is basic semi-algebraic over $\mathbb{Q}(c_1, \ldots, c_i)$. Thus, we can write the halting sets of M as a countable union of path sets:

$$\Omega_T = \bigcup_{n \in \mathbb{N}} \bigcup_{\pi \in \pi_T} \mathcal{V}_\pi^n \text{ and } \Omega_M = \bigcup_{t \in \mathbb{N}} \Omega_t$$

Since all unions are countable and $\mathcal{V}_\pi^n$ is basic semi-algebraic for each n, π, we have shown that $A = \Omega_M$ is a countable union of semi-algebraic sets. □

The following result from transcendental number theory is a useful tool for finding sets of algebraically independent numbers.

Theorem 5.2 (Lindemann-Weierstrass, 1985)**.** *If $a_1, \ldots, a_n$ are algebraic numbers that are linearly independent over $\mathbb{Q}$, then $\exp(a_1), \ldots, \exp(a_n)$ are algebraically independent over $\mathbb{Q}$.*

Proof. See [NT20, Corollary 2.6.1]. □

The method to obtain algebraically independent numbers using Theorem 5.2 is based on the observation that for two primes $p \neq q$, $\sqrt{p}$ and $\sqrt{q}$ are linearly independent in the vector space $\mathbb{Q}(\sqrt{p}, \sqrt{q})$ over $\mathbb{Q}$ whose elements have the form

$$a + b\sqrt{p} + c\sqrt{q} + d\sqrt{pq} \ (a, b, c, d \in \mathbb{Q}).$$

Assume $b\sqrt{p} + c\sqrt{q} = 0$ for some $b, c \neq 0$. Squaring this equation yields

$$b^2 p + c^2 q + 2bc\sqrt{pq} = 0.$$

Now, since $b^2 p$ and $c^2 q$ are rational, we let $z = \frac{b^2 p + c^2 q}{2bc}$ and obtain $\sqrt{pq} = -z$. Since $z \in \mathbb{Q}$, we have $\sqrt{pq} \in \mathbb{Q}$, contradicting that $\sqrt{pq}$ is irrational.

5.2 Restricted access to real constants

Theorem 5.3 [MZ06, Theorem 5]**.** *Let $c_1, \ldots, c_i$ be algebraically independent over $\mathbb{Q}$. Then, the finite set $A = \{c_1, \ldots, c_i\} \subseteq \mathbb{R}$ is decidable with i real machine constants, but not semi-decidable with $i - 1$ real constants.*

Theorem 5.2 yields an easy way to obtain any number of algebraically independent values $c_1, \ldots, c_n$ over $\mathbb{Q}$: Let $p_1, \ldots, p_n$ be primes with $p_i \neq p_j$ for $i \neq j$. Then,

$\exp(\sqrt{p_1}), \dots, \exp(\sqrt{p_n})$ are algebraically independent over $\mathbb{Q}$. This proves the existence of a set A as in the theorem for each $i \in \mathbb{N}$.

Hence, Theorem 5.3 proves that for each $i \in \mathbb{N}$, there is a problem that can be decided by a machine using i real constants, but not by a machine using $i - 1$ real constants. In other words, machines over $\mathbb{R}$ become strictly stronger if additional constants are allowed.

Proof. To see that A is decidable using i real constants, construct a machine that compares its input to each of the machine constants $c_1, \dots, c_i$.

We will now prove the second part of the claim. First note that $\operatorname{trdeg} \mathbb{Q}(A) = i$ since $\{c_1, \dots, c_i\}$ is a transcendence basis of $\mathbb{Q}(A)$. Suppose that A is semi-decidable by a machine with $i - 1$ real constants. By Lemma 5.1, A is a countable union of semi-algebraic sets over some field extension $\mathbb{Q}(\tilde{c}_1, \dots, \tilde{c}_{i-1}) = K$. Since A is a finite set, the union is in fact finite. Let $A = A_1 \cup \dots \cup A_k$ where A_j is semi-algebraic over K for each j.

Because A is finite, the A_j have to be finite as well. Hence, each field extension $K(A_j)/K$ is algebraic by Lemma 2.9. Thus, $K(A)/K$ is an algebraic field extension which implies that $\operatorname{trdeg} \mathbb{Q}(A) \leq i - 1$ (this is an upper bound for the transcendence degree of K since there exists a transcendence basis S of K with $S \subseteq \{\tilde{c}_1, \dots, \tilde{c}_{i-1}\}$). This, however, contradicts the fact that $\operatorname{trdeg} \mathbb{Q}(A) = i$ and therefore, A cannot be semi-decidable by a machine with $i - 1$ constants. □

This result has interesting implications on Post's problem as we can naturally define "slices" of the real halting problem in the following way.

Problem:	$\mathbb{H}_i$
Input:	A description $\langle w, c_1, \dots, c_i \rangle$ of a BSS machine M over $\mathbb{R}$ with constants $c_1, \dots, c_i \in \mathbb{R}$ and $w \in \mathbb{Q}$
Question:	Does M halt on input 0?

This yields an infinite family of problems that are BSS-undecidable but not sufficient for the real halting problem to be decidable if one of these problems is allowed as an oracle [MZ06, Corollary 7]. Interestingly, this is not the only explicit solution to the real version of Post's problem. Another solution is the set $\mathbb{Q}$ which is BSS-undecidable but not BSS-complete (i.e. the real halting problem cannot be reduced to $\mathbb{Q}$) [MZ05].

5.3 Application to algebraic circuits

It is well-known that non-uniform boolean circuit families are able to solve Turing-undecidable problems since the characteristic function of each problem $A \subseteq \mathbb{N}$ can be encoded into the definition of the circuit family by using an unary encoding for its inputs (c.f. [Vol99, Lemma 2.2]). In particular, let $\mathbb{H}_U$ be the unary version of the discrete halting problem which is defined as follows.

Problem:	$\mathbb{H}_U$
Input:	$1^n, n \in \mathbb{N}$
Question:	Does the Turing machine defined by n halt on input n?

Obviously, $\mathbb{H}_U$ is Turing-undecidable. However, it can be solved by the non-uniform circuit family $C = (C_n)_{n \in \mathbb{N}}$ where C_n outputs the constant value $\chi_{\mathbb{H}_U}(n)$.

Similar arguments do not work for algebraic circuit families as each problem that can be encoded into the input length is a subset of $\mathbb{N}$ and therefore decidable by Corollary 3.3. However, the preceding result gives us a tool to generalize the above observation, i.e. show that there are non-uniform algebraic circuit families for BSS-undecidable problems.

Theorem 5.4. *There exists a BSS-undecidable problem $A \subseteq \mathbb{R}$ that is decidable by a non-uniform circuit family over $\mathbb{R}$.*

Proof. Let $c_1, c_2, \ldots$ be an infinite sequence of algebraically independent numbers. Define $A_i = \{(c, \ldots, c) \in \mathbb{R}^i \mid c \in \{c_1, \ldots, c_i\}\}$ and $A = \bigcup_{i \in \mathbb{N}} A_i$. Suppose there exists a machine M that decides A and let k be the number of real constants of M. We can now define a machine M' with k real constants that decides the set $\{c_1, \ldots, c_{k+1}\}$: On

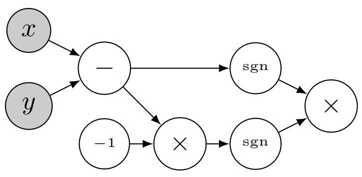


Figure 5.1: A circuit for $x = y$. The circuit outputs 1 if and only if both sign nodes output 1 which is equivalent to $x \geq y \wedge y \geq x$.

input x, M' checks whether $|x| = 1$ and simulates M on input $(x, \dots, x) \in \mathbb{R}^{k+1}$. This, however, contradicts Theorem 5.3 and therefore, A cannot be decidable.

To see that A is decidable by a non-uniform circuit family, let $C = (C_n)_{n \in \mathbb{N}}$ where C_n, on input $(x_1, \dots, x_n)$, checks whether $x_1 = \cdots = x_n$ using a layer of sub-circuits as in Figure 5.3 and outputs 1 if $x_1 = c_k$ for some $1 \leq k \leq n$ (this can be done using n constant gates).

Clearly, C_n decides A_n and therefore, because $A \cap \mathbb{R}^n = A_n$, C decides A. This circuit family is in fact a $\mathrm{NC}^1_{\mathbb{R}}$ family, thus proving $A \in \mathrm{NC}^1_{\mathbb{R}}$. □

6 Real computation on discrete inputs

The inputs for BSS machines over $\mathbb{R}$ that were discussed before are uncomputable, i.e. there is no way to represent these inputs using Turing machines. The aim of this chapter is to investigate the power of BSS machines over $\mathbb{R}$ in a setting where we only allow computable, i.e. discrete, inputs. More formally, we are interested in complexity classes of the form

$$\mathcal{C} \cap \Sigma^* := \{A \cap \Sigma^* \mid A \in \mathcal{C}\}$$

where $\mathcal{C}$ is a complexity class for BSS machines over $\mathbb{R}$ and Σ is an alphabet. For instance, in the case where $\mathcal{C}$ is the class of all decidable problems over $\mathbb{R}$, we know from Corollary 3.3 that $\mathcal{C} \cap \{0,1\}^*$ is the class of all languages over $\{0,1\}^*$ (since each such language is decidable over $\mathbb{R}$). Obviously, this contains Turing-undecidable languages and thus, BSS machines cannot be simulated by Turing machines.

The aim of the following studies is to investigate the reason for the superior power of BSS machines compared to Turing machines. Recall that the reason for the above result is that a machine constant c can be used to encode the solution for any discrete problem. We already studied the impact of real machine constants on the recursive power of BSS machines in Chapter 5. However, apart from the existence of such machine constants, the decision algorithm from Chapter 3 also requires the BSS machine to be able to actually access the information stored in c (as in the proof of Lemma 3.2). We consider the following restricted versions of BSS machines:

- *Additive machines [Blu+98, Chapter 21].* Additive machines are defined as in Definition 2.17 except that now in computation nodes, only additions and subtractions are allowed, i.e. $g_m = x \circ y, \circ \in \{+, -\}$ for each node m. Here, x, y are either coordinates of the state space or machine constants.

- *Branching on equality.* Instead of allowing our machines to make their decision in a branch node based on the question "$x \geq 0$?" for some x (*branching on order*), we may only allow the question "$x = 0$?".

Both multiplication and branching on order were used in the proof of Lemma 3.2. Therefore, it is a natural question to ask whether machines over $\mathbb{R}$ keep their super-recursive power even without the ability to use these operations.

Apart from these restrictions, there are also variants that assign a higher cost to iterative multiplications and different variants of nondeterminism [Blu+98, Chapter 5, 20], but these restrictions mostly affect the runtime rather than the recursive power of BSS machines (although this is not generally true for arbitrary rings – for instance, problems over $\mathbb{Z}$ that are decidable in nondeterministic polynomial time are not necessarily decidable over $\mathbb{Z}$, cf. [Blu+98, Chapter 5, Remark 5]).

6.1 Simulations for additive machines

In 1994, Koiran proved that additive machines retain their super-recursive power on discrete inputs (i.e. inputs from Σ^* where Σ is an alphabet) if and only if they are allowed to branch on order. More specifically, they proved that $\mathrm{P}^{<}_{\mathrm{add}} \cap \Sigma^* = \mathrm{P/poly}$ and $\mathrm{P}^{=}_{\mathrm{add}} \cap \Sigma^* = \mathrm{P}$. Here, $\mathrm{P}^{<}_{\mathrm{add}}$ ($\mathrm{P}^{=}_{\mathrm{add}}$) denotes the class of decision problems that are decidable in polynomial time by additive machines over $\mathbb{R}$ that branch on order (equality).

Theorem 6.1 [Koi94, Theorem 1]**.** $\mathrm{P}^{=}_{\mathrm{add}} \cap \Sigma^* = \mathrm{P}$

In the following proof, $\mathrm{bin}(n)$ denotes the binary representation of the integer n.

Proof. First note that $\mathrm{P} \subseteq \mathrm{P}^{=}_{\mathrm{add}}$ trivially holds because each base operation of Turing machines can be simulated by an additive machine in constant time.

We will now prove $\mathrm{P}_{\mathrm{add}}^{=} \subseteq \mathrm{P}$. Let $A \in \mathrm{P}_{\mathrm{add}}^{=}$ via the additive machine M with the machine constants $c_1, \ldots, c_k$. Let x be an input for M with $|x| = n$. We assume without loss of generality that $c_k = 1$. The key observation is that at any time, all registers of M hold values that can be represented as

$$\sum_{i=1}^{k} a_i c_i$$

where $a_i \in \mathbb{Z}$ and $|\mathrm{bin}(a_i)| \leq p(n)$ for some polynomial p. The latter can be proven by induction on the steps of M (for instance, one can prove $x_i^t \leq 2^t \cdot \hat{c}$ where $\hat{c} = \max\{|c_1|, \ldots, |c_k|\}$ and x_i^t is the value that is stored in the ith register of M after t steps).

Let V be the vector space over $\mathbb{Q}$ spanned by $c_1, \ldots, c_k$ and let m be the dimension of V. We can assume without loss of generality that $(c_1, \ldots, c_m)$ is a base of V. Let $(b_{ij}) \in \mathbb{Q}^{k \times m}$ be a matrix such that

$$c_i = \sum_{j=1}^{m} b_{ij} c_j.$$

Then, we can write the value of any register x at a given time during the computation as

$$\begin{aligned} x &= \sum_{i=1}^{k} a_i \sum_{j=1}^{m} b_{ij} c_j \\ &= \sum_{i=1}^{k} \sum_{j=1}^{m} a_i b_{ij} c_j \\ &= \sum_{j=1}^{m} (\sum_{i=1}^{k} a_i b_{ij}) c_j \end{aligned}$$

and therefore, the test "$x = 0$?" can be performed by a Turing machine by checking if $\sum_{i=1}^{k} a_i b_{ij} = 0$ for all $1 \leq j \leq m$. Hence, M can be simulated by a Turing machine M' in polynomial time. □

Obviously, the proof is also valid if we drop the restriction that M and M' run in polynomial time. Therefore, additive machines that branch on equality checks, restricted to discrete inputs, are just as powerful as Turing machines. We now move on to prove that additive machines that branch on order are still strictly more powerful than Turing machines. For that, it suffices to show that P/poly $\subseteq \mathrm{P}^{<}_{\mathrm{add}}$ as P/poly already contains undecidable problems such as the unary halting problem.

Theorem 6.2 [Koi94, Theorem 9]**.** $\mathrm{P}^{<}_{\mathrm{add}} \cap \Sigma^* = \mathrm{P/poly}$

Proof. To see that P/poly $\subseteq \mathrm{P}^{<}_{\mathrm{add}}$, we can use the same argument as in Lemma 3.2. The polynomial advice $f(|x|)$ is encoded in a real machine constant a with

$$a = 0.f(1)\#f(2)\#f(3)\#\dots$$

(in binary representation). The first digit a_1 of $f(1)$ can be accessed by checking $a \geq \frac{1}{2}$. To access the second digit a_2, the same check is applied to $2a - a_1$. The multiplication $2a$ is the same as $a + a$ and therefore computable by an additive machine. By iterating this process, the machine can access the advice for its input. This reduces the task to the simulation of a deterministic polynomial time algorithm which is possible in polynomial time due to Theorem 6.1.

For $\mathrm{P}^{<}_{\mathrm{add}} \subseteq$ P/poly, let M be a $\mathrm{P}^{<}_{\mathrm{add}}$-machine. We will approximate the machine constants $c_1, \dots, c_k$ of M by rational values $\overline{c}_1, \dots, \overline{c}_k$ such that the branching checks behave exactly as in the computation of M.

Let $B(x)$ be the set of all values that appear in a branch node during the computation of M on input x and $B_n = \bigcup_{x \in \Sigma^n} B(x)$. Clearly, B_n is finite for each $n \in \mathbb{N}$.

Let $L_n = \{b \in B_n \mid b < 0\}$. There exists some $\varepsilon > 0$ such that $|b| \geq \varepsilon$ for all $b \in L_n$. Thus, for each $x \in B_n$, we have that $x < 0$ if and only if $x' < 0$ for all x' with $|x - x'| < \varepsilon$. The same argument holds if $x > 0$. The case $x = 0$ can be eliminated using the technique from Theorem 6.1.

Hence, we can approximate the c_i without changing the computation path of M by choosing $\overline{c}_1, \dots, \overline{c}_k$ such that the error at any given step of the computation is at most ε regardless of the input. This only requires $|c_i - \overline{c}_i|$ to be sufficiently small. We can use the polynomial advice to enable the Turing machine to use these approximations (i.e. $f(n) = (\overline{c}_1, \dots, \overline{c}_k)$ where $\overline{c}_i$ depends on n for $i = 1, \dots, k$).

It remains to be shown that $|f(n)|$ grows only polynomially with respect to n. To do this, we reformulate the problem as a system of linear inequalities. Each $b \in B_n$ can be written as some linear combination

$$b = a_1c_1 + \cdots + a_kc_k,$$

where $a_j \in \mathbb{Z}$ for $j = 1, \ldots, k$. Hence, for each $b \in B_n$, the approximations have to satisfy an equation

$$\begin{cases} a_1\overline{c}_1 + \cdots + a_k\overline{c}_k < 0, & \text{if } b \in L_n \\ a_1\overline{c}_1 + \cdots + a_k\overline{c}_k \geq 0, & \text{otherwise.} \end{cases}$$

Since M is a $\mathrm{P}^{<}_{\mathrm{add}}$-machine, there is a polynomial p such that M halts after at most $p(n)$ steps for each input x with $|x| = n$, yielding $|a_j| \leq 2^{p(n)}$. We can deduce from Lemma 2.14 that there exists a solution $\overline{c} = (\overline{c}_1, \ldots, \overline{c}_k)$ to this system with size$(\overline{c})$ polynomial in $k \cdot |\mathrm{bin}(2^{p(n)})| = k \cdot p(n)$, proving that there exists an advice function f such that $|f(n)|$ grows only polynomially with respect to n. $\square$

This proof shows that multiplication as in the proof of Lemma 3.2 is not a necessary prerequisite for BSS machines to decide Turing-undecidable problems. In fact, each problem $A \subseteq \Sigma^*$ is decidable by an additive BSS machine that branches on order (using the same argument as in the above proof, but without the restriction to polynomial time).

The technique for proving $\mathrm{P}^{=}_{\mathrm{add}} = \mathrm{P}$ can also be used to obtain a similar result for parallel computation. Let $\mathrm{NC}^i_{\mathrm{add}}$ denote the class of problems that can be decided by a family of additive $\mathrm{NC}^i_{\mathbb{R}}$ circuits. Additive circuits are defined like algebraic circuits, but with the following adjustments:

- all arithmetic nodes are either associated with $+$ or $-$
- on input x, sign nodes compute the function $\begin{cases} 1, & x = 0 \\ 0, & \text{otherwise} \end{cases}$

Theorem 6.3. $\mathrm{NC}^i_{\mathrm{add}} \cap \Sigma^* \subseteq \mathrm{AC}^i$

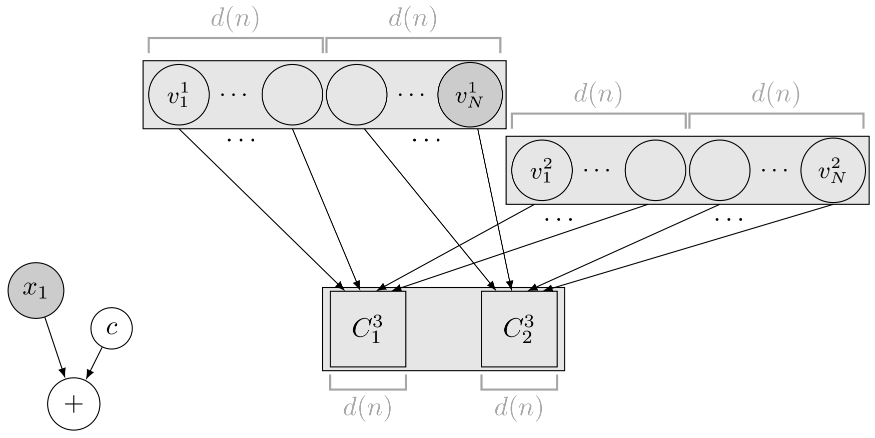


Figure 6.1: An algebraic circuit (left) and a corresponding boolean circuit (right). The node labeled x_1 is the input node and the node labeled c is a constant node. Since the circuit has one constant node, each linear combination for intermediate results has the form $a_1c + a_2$. Thus, the addition node has two sub-circuits (one circuit for each a_i).

Proof. Let $A \in \mathrm{NC}^i_{\mathrm{add}}$ via the family of algebraic circuits $C = (C_n)_{n\in\mathbb{N}}$, i.e. $f_C(x) = \chi_A(x)$ for all $x \in \{0,1\}^*$ where χ_A is the characteristic function of A. Moreover, let $c_1, \dots, c_k$ be the constants appearing in C_n, $d(n)$ an upper bound for the depth of C_n and $s(n)$ an upper bound for the size of C_n.

The idea is that each intermediate result x appearing during the computation of C_n can be represented by a linear combination

$$x = a_1c_1 + \cdots + a_kc_k + a_{k+1}$$

where $a_i \in \mathbb{Z}$ (as in the proof of Theorem 6.1). During the computation of C, we have $|\mathrm{bin}(a_i)| \leq d(n)$ for each a_i. Therefore, we can use $k+1$ blocks of $d(n)$ nodes to store x in the boolean circuit C'_n (see Figure 6.1 for an example). In the following, let $N = (k+1)d(n)$. We construct an equivalent circuit C'_n by modifying C_n in the following way:

- Each input node g is replaced by a sequence $v_1^g, \dots, v_N^g$ of nodes where $v_1^g, \dots, v_{N-1}^g$ are constant nodes holding the value 0 and $v_N^g = g$.

- Each constant node g holding the constant c_j is replaced by the nodes $v_1^g, \dots, v_N^g$ where

$$v_i^g \text{ is } \begin{cases} \text{a constant } 1, & \text{if } i = j \cdot d(n) \\ \text{a constant } 0, & \text{otherwise.} \end{cases}$$

- Each arithmetic node g labeled $\circ \in \{+, -\}$ with the predecessors h_1 and h_2 is replaced by $k+1$ circuits C_i^g that compute

$$\mathrm{val}\left(v_{(i-1)d(n)+1}^{h_1}, \dots, v_{i \cdot d(n)}^{h_1}\right) \circ \mathrm{val}\left(v_{(i-1)d(n)+1}^{h_2}, \dots, v_{i \cdot d(n)}^{h_2}\right)$$

  where $\mathrm{val}(x)$ is defined by $\mathrm{bin}(x) \mapsto x$. Each C_i^g can be realized in constant depth and polynomial size [Vol99, Section 1.1]. Let $o_1^i, \dots, o_{d(n)}^i$ be the output nodes of C_i^g. For each $1 \leq j \leq N$, let $y_1 \cdot d(n) + y_2$ be the unique representation of j where $1 \leq y_2 \leq d(n)$. Then, we set $v_j^g = o_{y_2}^{y_1+1}$.

- Each sign node g with the predecessor h is replaced by the nodes $v_1^g, \dots, v_N^g$ where $v_1^g, \dots, v_{N-1}^g$ are constant nodes holding the value 0 and v_N^g is the output node of a circuit that computes $\neg(v_1^h \vee \dots \vee v_N^h)$.

- Let h be the predecessor of the output node in C_n. Then, v_N^h becomes the output node of C_n'.

On inputs $x \in \{0,1\}^n$, the new circuit C_n' always computes the same output as C_n. Each node in C_n is replaced by a sub-circuit with polynomially many nodes and constant depth. Thus, we have proven $A \in \mathrm{AC}^i$. $\square$

This proof is, of course, valid for the non-uniform case. Moreover, it can easily be verified that the above construction can be computed in polynomial time. Thus, the resulting circuit family C' is P-uniform if and only if the algebraic circuit family C is $\mathrm{P}_{\mathrm{add}}^{=}$-uniform.

6.2 Simulations for multiplicative machines

As we have seen in the previous section, additive BSS machines are not stronger than Turing machines unless they are allowed to branch on order rather than on equality. This leads to the hypothesis that the super-recursive power of unrestricted BSS machines is entirely based on their ability to branch on order. Consequently, one might assume that BSS machines with the ability to evaluate polynomials but without the ability to branch on order can still be simulated by Turing machines. The following theorem shows that this is indeed the case.

Theorem 6.4. *Let M be a BSS machine over $\mathbb{R}$ that branches on equality and $\Sigma = \{0,1\}^*$. If M accepts the language A, then there exists a Turing machine M' such that M' accepts the language $A \cap \Sigma^*$.*

We need a result from computer algebra to prepare the proof of Theorem 6.4. If R is the polynomial ring $K[x_1, \ldots, x_m]$ for some field K and $J \subseteq R$ is an ideal, the ideal membership problem (IMP) is defined as follows.

Problem:	IMP(R, J)
Input:	An element $a \in R$
Question:	Does $a \in J$ hold?

Lemma 6.5. *Let K be a field and $J \subseteq K[x_1, \ldots, x_m] =: R$ an ideal. Then,* IMP(R, J) *is decidable.*

Proof. See [GG13, Chapter 21] or [GP08, Chapter 1.8.1]. □

Theorem 6.6. *Let K be a field, $f \in K[x_1, \ldots, x_m], a = (a_1, \ldots, a_m) \in K^m$ and $J \subseteq K[x_1, \ldots, x_m]$ an ideal. Then, the question "$f(a) = 0$?" is decidable.*

Proof. Let ev_a be the evaluation homomorphism

$$ev_a \colon K[x_1, \ldots, x_k] \to K, f(x_1, \ldots, x_k) \mapsto f(a_1, \ldots, a_k).$$

Since ev_a is a homomorphism, $\ker ev_a$ is an ideal $J \subseteq K[x_1, \ldots, x_k]$. Hence, $f(c_1, \ldots, c_k) = 0$ if and only if $f \in J$. By Lemma 6.5, this is decidable. □

We will use this result to simulate branch nodes in the following proof. The key observation is that we can use the same technique as in Theorem 4.1 to avoid calculating with real numbers. Instead of storing the real values from the registers of the BSS machine, we store a polynomial that needs to be evaluated in order to get the actual value in the register.

Proof. (of Theorem 6.4) Let $x = (x_1, \ldots, x_n)$ and M be a BSS machine as in the claim of the theorem. Moreover, let $c_1, \ldots, c_k$ be the machine constants of M. To obtain a Turing machine M' with the desired properties, we use a similar construction as in the proof of Theorem 4.1: For $1 \leq i \leq n$, let $f_i \in \mathbb{Q}[z_1, \ldots, z_k]$ be the constant polynomials $f_i(z_1, \ldots, z_k) = x_i$. On input x, M' first stores the string

$$f_0 \# f_1 \# \cdots \# f_n$$

on its tape where $\#$ is a separator symbol and $f_0(z_1, \ldots, z_k) = n$. Throughout this proof, we identify polynomials with their representation over Σ. Note that M' does *not* store the value of f_i, but a representation of the polynomials. Let f be the polynomial that corresponds to s_1 in a state

$$(\ldots, s_0.s_1, s_2, \ldots)$$

of M. To keep track of the shift operations of M, we can use a special symbol that marks the beginning of f's representation. For instance, the first symbol of f_1 will be marked in the initial configuration of M'. To simulate a step of M, M' will behave as follows:

- To simulate a *computation node*, M' adjusts the polynomials on its tape as in the proof of Theorem 4.1. This is possible because each polynomial over $\mathbb{Q}$ has a finite representation.

- To simulate a *shift node*, M' moves the special symbol on its tape that marks the first symbol of f_1.

- To simulate a *branch node*, M' checks whether $g(c_1, \dots, c_k) = 0$. Here, g denotes the polynomial that corresponds to the register of M with coordinate 1. By Theorem 6.6, the check $g(c_1, \dots, c_k) = 0$ is computable.
- To simulate an *output node*, M checks whether $g(c_1, \dots, c_k) = 0$ for g as above. If yes, M' rejects. Otherwise, M' accepts.

Clearly, M' accepts the same inputs as M. □

Let $\mathcal{C}$ denote the class of all problems that are decidable by a BSS machine which branches on equality. Theorem 6.4 shows that $\mathcal{C} \cap \Sigma^*$ contains only Turing-decidable problems. Thus, we can conclude that BSS machines are not able to exploit their ability to perform exact computations on real numbers to gain more recursive power than Turing machines unless they are allowed to branch on order.

Again, we may take the runtime of the simulated BSS machine M into account and investigate the simulation's slow-down. Let $\mathrm{P}_{\mathbb{R}}^{=}$ denote the class of problems that are decidable in polynomial time by a BSS machine that branches on equality. One can show that $\mathrm{P}_{\mathbb{R}}^{=} \cap \{0,1\}^* \subseteq \mathrm{BPP}$ [Koi97, Theorem 9], but an efficient simulation of BSS machines with branches on equality has not yet been found. With regard to the result $\mathrm{P}_{\mathrm{add}}^{=} \cap \Sigma^* = \mathrm{P}$ (Theorem 6.1), this presumably means that multiplication does have a significant impact on the runtime of BSS machines.

This led to the study of a variant of BSS machines that assigns a higher cost to its base operations depending on how complicated the representation of the operation's result by a polynomial $f(x_1, \dots, x_n, c_1, c_k)$ with respect to the input $x = (x_1, \dots, x_n)$ and the machine constants $c_1, \dots, c_k$ is. A natural measure is the length of the polynomial's binary representation which is otherwise the bottleneck in the simulation of BSS machines (as the polynomial can grow exponentially with respect to the runtime of M). These machines are called *weak BSS machines* (see [Koi97, Section 3.1] for a precise definition) and it has been proven that these machines can efficiently be simulated by Turing machines [Koi97, Theorem 8].

7 Conclusion

Since their introduction in 1989, several surprising, partly undesired properties of BSS machines have been found, probably the most notable result being that the undecidable halting problem is no longer undecidable if we establish BSS machines over $\mathbb{R}$ as our model of computation.

In Chapter 2, we have introduced the BSS model of computation for arbitrary rings. While this model is a useful way to capture decidability questions which are not expressible for Turing machines, we have seen that the respective results heavily depend on minor details in the definition of BSS machines.

These details were the subject of Chapter 5 and 6 where we studied certain restrictions on the instruction set as well as restrictions on the allowed machine constants. Building on Koiran's proof from 1994 who proved that additive BSS machines that branch on an equality relation can be efficiently simulated by Turing machines while additive machines that branch on an order relation cannot be simulated in a uniform way at all, we found that BSS machines are able to decide Turing-undecidable problems if and only if they are allowed to branch on an order relation rather than an equality relation. Our proof used a construction from Michaux which we presented in the 4th chapter. This construction was originally used by Michaux to prove the surprising result that space does not play an important role for BSS machines over $\mathbb{R}$ as each BSS machine over $\mathbb{R}$ can be simulated in linear space. In addition, we used Koiran's proof to find an efficient way to simulate additive algebraic circuits by boolean circuits.

The 5th chapter concerned itself with a restricted version of BSS machines where only a limited number of irrational machine constants is allowed. This restriction was based on the observation that BSS machines can use irrational machine constants to encode the solution of Turing-undecidable problems. Meer and Ziegler proved that BSS machines gain strictly more recursive power for each additional irrational machine constant that is allowed. We used this proof to show that there is a BSS-undecidable problem which is decidable by a non-uniform algebraic circuit family, thus generalizing the respective result from classical circuit complexity and justifying the notion of uniformity for algebraic circuit families. Ultimately, the consequence of the results in Chapter 5 and 6 is that BSS machines over $\mathbb{R}$ can decide any problem over $\{0,1\}^*$ if they are allowed to

- branch on an order relation (i.e. get an answer to the question "$x \geq 0$?") and
- use at least one irrational machine constant.

Otherwise, their recursive power does not exceed that of Turing machines.

In the 3rd chapter, we presented a way to encode BSS machines over $\mathbb{R}$ by a sequence in $\mathbb{R}^\infty$. Next, we constructed a universal BSS machine as proposed by Blum, Shub and Smale, allowing us to formulate a generalization of the discrete halting problem to BSS machines which is undecidable even by BSS machines using the same argument as in the discrete case.

While BSS machines over $\mathbb{R}$ are, for now, a purely theoretical model and an idealization of physical computers, it is possible to come up with reasonable restrictions that limit their recursive power (even with respect to complexity) sufficiently that they can be efficiently simulated by Turing machines. Further research in algebraic computation with the help of tools from algebra and topology will certainly contribute to a better understanding of classical complexity theory in the future.

Bibliography

[Blu+98] L. Blum et al. *Complexity and Real Computation.* Springer New York, 1998.

[BSS89] Leonore Blum, Michael Shub, and Stephen Smale. “On a theory of computation and complexity over the real numbers: NP-completeness, recursive functions and universal machines”. In: *Bulletin of the American Mathematical Society* 21 (1989), pp. 1–46.

[Bos20] Siegfried Bosch. “Transzendente Erweiterungen”. In: *Algebra.* 9th ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2020, pp. 377–429.

[CM99] Felipe Cucker and Klaus Meer. “Logics Which Capture Complexity Classes Over The Reals”. In: *J. Symb. Log.* 64.1 (1999), pp. 363–390.

[GG13] Joachim von zur Gathen and Jürgen Gerhard. *Modern Computer Algebra (3. ed.)* 3rd ed. Cambridge University Press, 2013.

[Göd31] Kurt Gödel. “Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I”. In: *Monatshefte für Mathematik und Physik* 38.1 (1931), pp. 173–198.

[GP08] Gert-Martin Greuel and Gerhard Pfister. *A Singular Introduction to Commutative Algebra.* 2nd ed. Springer Berlin Heidelberg, 2008.

[Koi94] Pascal Koiran. “Computing over the Reals with Addition and Order”. In: *Theor. Comput. Sci.* 133.1 (1994), pp. 35–47.

[Koi97] Pascal Koiran. “A Weak Version of the Blum, Shub, Smale Model”. In: *Journal of Computer and System Sciences* 54.1 (1997), pp. 177–189.

[Lan87] Serge Lang. *Linear Algebra.* 3rd ed. Springer, 1987.

[MP07] C. Maclean and D. Perrin. *Algebraic Geometry: An Introduction*. Universitext. Springer London, 2007.

[MZ05] Klaus Meer and Martin Ziegler. “An Explicit Solution to Post’s Problem over the Reals”. In: *Fundamentals of Computation Theory*. Ed. by Maciej Liśkiewicz and Rüdiger Reischuk. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 467–478.

[MZ06] Klaus Meer and Martin Ziegler. “Uncomputability below the Real Halting Problem”. In: *Proceedings of the Second Conference on Computability in Europe: Logical Approaches to Computational Barriers*. CiE’06. Swansea, UK: Springer-Verlag, 2006, pp. 368–377.

[Mic89] Christian Michaux. “Une remarque a propos des machines sur R introduites par Blum, Shub et Smale”. In: *CR Acad. Sci. Paris* 309.1 (1989), pp. 435–437.

[NT20] Saradha Natarajan and Ravindranathan Thangadurai. “Early Transcendence Results from Nineteenth Century”. In: *Pillars of Transcendental Number Theory*. Springer Singapore, 2020, pp. 21–33.

[Sch86] Alexander Schrijver. *Theory of Linear and Integer Programming*. Wiley, 1986.

[Tur37] Alan Turing. “On Computable Numbers, with an Application to the Entscheidungsproblem”. In: *Proceedings of the London Mathematical Society* 42.1 (1937), pp. 230–265.

[Vol99] Heribert Vollmer. *Introduction to Circuit Complexity*. Texts in Theoretical Computer Science. Springer Berlin Heidelberg, 1999.