Chapter 5 from the book

HILBERT's TENTH PROBLEM

by Yuri Matiyasevich

Russian original published by NAUKA Publishers
Moscow, 1993; ISBN 5-02-014356-X

English translation published by the MIT Press
Cambridge, London, 1993; ISBN 0-262-13295-8

With kind permission of the MIT Press, Foreword to the English translation, written by Martin Davis, English translation of the Preface and Chapters 1 and 5 of the book are put on the WWW site with URL

`http://logic.pdmi.ras.ru/~yumat/H10Pbook`

and mirrored on

`http://www.informatik.uni-stuttgart.de/ifi/ti/personen/Matiyasevich/H10Pbook`

Updated Bibliography is available via URL

`http://logic.pdmi.ras.ru/Hilbert10`

5 Hilbert's Tenth Problem Is Unsolvable

In this chapter we discuss Turing machines, abstract computers of a simple kind, and establish a connection between them and Diophantine equations. This will enable us to give a negative solution of Hilbert's Tenth Problem.

5.1 Turing machines

The number-theoretic techniques developed in the previous chapters will permit us to tackle Hilbert's Tenth Problem from the point of view of computability theory. But first we must clarify what is to be understood by the word "process" when Hilbert speaks of "a process according to which it can be determined by a finite number of operations whether the equation is solvable in rational integers." There are many ways in which this needed clarification can be provided, but the fundamental idea is always the same. We are able to convince ourselves that we have a general method for solving problems of a particular kind if, by using this method, we can solve any individual problem of that kind without employing our creative abilities, that is, so to speak, mechanically. This last can be formalized by saying that we can construct a machine that will solve problems of the class without our intervention, i.e., automatically.

Turing machines are abstract computers. Although they can be described in purely mathematical set-theoretic terminology, we will adopt the convention of describing Turing machines as though they were physical devices.

A Turing machine has memory in the form of a *tape* divided into *cells*. The tape has a single end; to make matters definite, we assume it to be on the left. To the right, the tape is potentially infinite. This means that, in contrast with actual physical computers, a computation by a Turing machine will never lead to an abnormal termination with the diagnostic "INSUFFICIENT MEMORY." On the other hand, any particular computation will require only finitely many cells.

Each cell will either be empty or will contain a single *symbol* from a finite set of symbols $\mathrm{A} = \{\alpha_1, \dots, \alpha_w\}$, called an *alphabet*. Different machines may have different alphabets. One of the symbols will play a special role in that it will always mark the left-most cell and appear nowhere else. We use the symbol "$\star$" for this marker. In addition, we need a symbol to denote an empty cell, and we follow tradition in using the letter "Λ" for this purpose.

Symbols on the tape are read and written by a *head*, which at each moment of discrete time scans one of the cells. The head can move along the tape to the left and to the right.

At each moment, the machine is in one of finitely many *states* that, following tradition, we will denote by $\mathrm{q}_1, \ldots, \mathrm{q}_v$. One of the states is declared to be *initial*, and we shall always suppose that this is q_1. In addition, one or more states are declared to be *final*.

The next action of a machine is totally determined by its current state and the symbol scanned by the head. In a single step, the machine can change the symbol in the cell, move the head one cell to the left or to the right, and pass into another state. The actions are defined by a set of *instructions* of the form

$$\mathrm{q}_i\alpha_j \Longrightarrow \alpha_{A(i,j)}D(i,j)\mathrm{q}_{Q(i,j)}, \tag{5.1.1}$$

where

(a) q_i, the current state, is not permitted to be final;
(b) α_j is the symbol scanned by the head, $\alpha_j \in \mathrm{A}$ unless the current cell is empty in which case $\alpha_j = \Lambda$.
(c) $\alpha_{A(i,j)}$ is the symbol to be written, $\alpha_{A(i,j)} \in \mathrm{A}$ (the case $\alpha_{A(i,j)} = \alpha_j$ is not excluded);
(d) $D(i,j)$ represents motion by the head—there are three options:

(i) L, the head moves one cell to the left,
(ii) R, the head moves one cell to the right,
(iii) S, the head stands still;

(e) $\mathrm{q}_{Q(i,j)}$ is the new state (the case $\mathrm{q}_{Q(i,j)} = \mathrm{q}_i$ is not excluded).

We refer to the part of an instruction preceding the $\Longrightarrow$ as its left-hand side, and the part following the $\Longrightarrow$ as its right-hand side. Following our convention, if $\alpha_j = \star$, then $\alpha_{A(i,j)} = \star$ and $D(i,j) \neq \mathrm{L}$; on the other hand, if $\alpha_j \neq \star$, then $\alpha_{A(i,j)} \neq \star$. For each pair consisting of a non-final state and an element of $\mathrm{A} \cup \{\Lambda\}$, there must be *exactly* one instruction with the corresponding left-hand side.

To begin with, some initial segment of the tape is occupied by symbols from A with no gaps, the rest of the potentially infinite tape is empty, the head scans one of the cells, and the machine is in the initial state q_1. Then the work of the machine proceeds step by step in accordance with the instructions. The work will stop after an instruction in which $\mathrm{q}_{Q(i,j)}$ is a final state is executed. It is possible that the machine will never reach a final state and that the work will continue indefinitely.

The input information is determined by the initial content of the tape and the position of the head on the tape. The result of the computation, the output, is determined when the work stops by the content of the tape, the position of the head on the tape, and the final state. The acts of interpreting the input and output information are events external to the Turing machine. For example, natural numbers could be written on the tape in binary notation using two symbols α and β, α being zero and β one, or vice versa; it is clear that in these two cases the same machine would, in general, compute two different functions.

5.2 Composition of machines

We need to establish the existence of a number of machines with particular properties. In principle, we could exhibit the full system of instructions for each of these machines, but this would turn out to be cumbersome and impracticable. Instead, we shall explicitly provide instructions only for several of the simplest machines and describe two methods for constructing more complex machines from given machines. This will enable us to obtain machines with the required properties.

All the machines that we are going to construct will have the same alphabet,

$$\{\star, 0, 1, 2, 3, \lambda\}. \tag{5.2.1}$$

There will be two final states, q_2 and q_3, and we shall interpret reaching q_2 as the answer "YES" and reaching q_3 as the answer "NO." The cells containing the symbol "λ" will play the role of *proxies* for empty cells, in the following sense: only empty cells and cells containing the symbol "λ" may be situated to the right of a cell containing the symbol "λ", and for any state q_i, the instructions with left-hand sides $\mathrm{q}_i\lambda$ and $\mathrm{q}_i\Lambda$ will have identical right-hand sides.

The first method for constructing a new Turing machine M from two given machines M_1 and M_2 is as follows:

(a) *In all instructions of machine M_1, the final state q_2 is replaced by q_{v+1}, where v is the number of states of machine M_1* (it should be recalled that final states can occur only in the right-hand sides of instructions).
(b) *In all instructions of machine M_2, every non-final state q_i is replaced by q_{v+i}* (in particular, q_1 is replaced by q_{v+1}).
(c) *The set of instructions of the new machine M consists of the instructions of both of the given machines, modified as described above.*

The action of machine M clearly consists of the consecutive execution of the actions of machines M_1 and M_2 *as originally constituted*, provided that machine M_1 halted

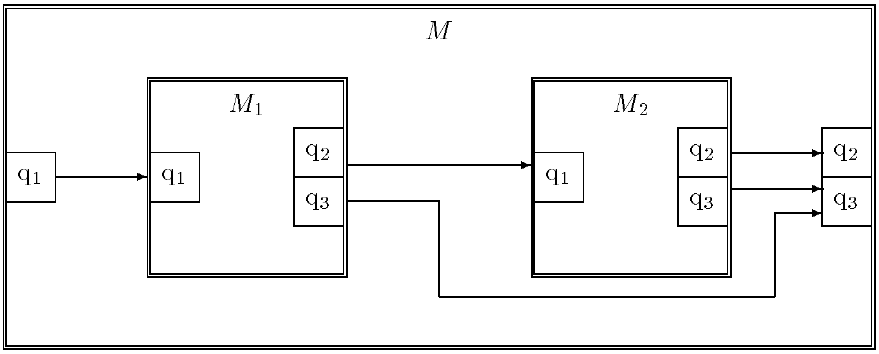

Figure 5.1. The first method of composition

in state q_2. To denote machine M, we shall use any one of the three notations

$$M_1 ; M_2$$

$$M_1 \textbf{ and } M_2$$

or

$$\textbf{if } M_1 \textbf{ then } M_2$$

depending on which seems most natural in a given context.

It easy to see that the composition of Turing machines described above is an associative operation, so notation like $M_1; M_2; M_3$ is unambiguous.

The second method for constructing a new Turing machine M from two given machines M_1 and M_2 is as follows:

(a) *In all instructions of machine M_1, the final state q_2 is replaced by q_{v+1}, where v is the number of states of machine M_1, and the final state q_3 is replaced by q_2.*
(b) *In all instructions of machine M_2, every non-final state q_i is replaced by q_{v+i}, and the final state q_2 is replaced by q_1.*
(c) *The set of instructions of the new machine M consists of the instructions of both of the given machines, modified as described above.*

The Turing machine constructed in this way will be denoted by

$$\textbf{while } M_1 \textbf{ do } M_2 \textbf{ od}$$

The action of this machine consists in performing in turn the actions of machines

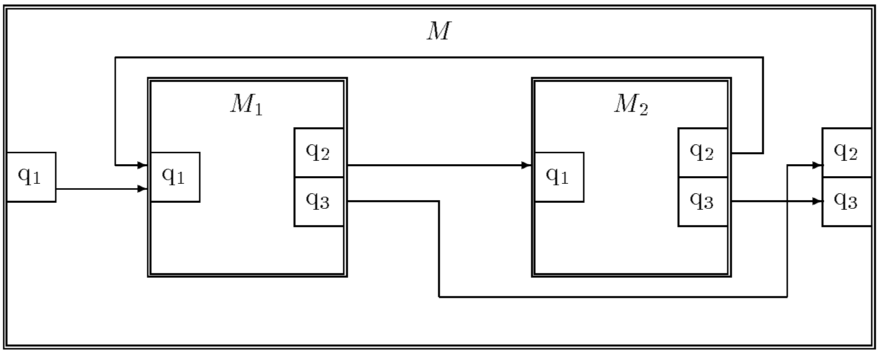

Figure 5.2. The second method of composition

M_1 and M_2 *as originally constituted* until one of them enters the final state q_3.

The notation introduced above resembles a primitive programming language. In fact, every such "program" denotes a particular Turing machine.

5.3 Basis machines

In this section we construct a number of specific Turing machines and describe their actions. While Turing machines can be thought of as working with strings of characters, we are most interested in dealing with numbers. For this we adopt a number of conventions for representing natural numbers on the tape. First of all, we shall use unary notation, namely, *the representation of a number m* will occupy $m+1$ consecutive cells. The leftmost of these will contain the symbol "0", while the remainder will contain the symbol "1". Naturally, the cell adjacent to the rightmost cell of such a representation cannot contain the symbol "1".

A representation of the tuple

$$\langle a_1, \ldots, a_n \rangle \tag{5.3.1}$$

will consist of the representations of the numbers a_1, ... , a_n arranged one after another without gaps, beginning with the second cell of the tape. (According to our agreement, the first cell always contains the end-marker "$\star$".) The other cells may either be empty or, according to our convention from Section 5.2, contain the symbol "λ". The representation with no cells containing λ will be called *canonical.*

$\star$	0	1	1	0	1	0	0	1	1	1	λ		…

Figure 5.3. A representation of the tuple $\langle 2, 1, 0, 3\rangle$

We begin with a number of extremely simple machines that will be presented explicitly.

Machine LEFT with instructions

$$\begin{aligned} \mathrm{q}_1\star &\Longrightarrow \star\mathrm{Sq}_2 \\ \mathrm{q}_1 0 &\Longrightarrow 0\mathrm{Lq}_2 \\ \mathrm{q}_1 1 &\Longrightarrow 1\mathrm{Lq}_2 \\ \mathrm{q}_1 2 &\Longrightarrow 2\mathrm{Lq}_2 \\ \mathrm{q}_1 3 &\Longrightarrow 3\mathrm{Lq}_2 \\ \mathrm{q}_1\lambda &\Longrightarrow \lambda\mathrm{Lq}_2 \\ \mathrm{q}_1\Lambda &\Longrightarrow \lambda\mathrm{Lq}_2 \end{aligned} \tag{5.3.2}$$

moves the head one cell to the left unless it was already scanning the leftmost cell marked by the symbol "$\star$".

Machine RIGHT with instructions

$$\begin{aligned} \mathrm{q}_1\star &\Longrightarrow \star\mathrm{Rq}_2 \\ \mathrm{q}_1 0 &\Longrightarrow 0\mathrm{Rq}_2 \\ \mathrm{q}_1 1 &\Longrightarrow 1\mathrm{Rq}_2 \\ \mathrm{q}_1 2 &\Longrightarrow 2\mathrm{Rq}_2 \\ \mathrm{q}_1 3 &\Longrightarrow 3\mathrm{Rq}_2 \\ \mathrm{q}_1\lambda &\Longrightarrow \lambda\mathrm{Sq}_2 \\ \mathrm{q}_1\Lambda &\Longrightarrow \lambda\mathrm{Sq}_2 \end{aligned} \tag{5.3.3}$$

moves the head one cell to the right unless it was scanning an empty cell or a cell containing the symbol "λ".

Machine WRITE(0) with instructions

$$\begin{aligned} \mathrm{q}_1\star &\Longrightarrow \star\mathrm{S}\mathrm{q}_2 \\ \mathrm{q}_1 0 &\Longrightarrow 0\mathrm{S}\mathrm{q}_2 \\ \mathrm{q}_1 1 &\Longrightarrow 0\mathrm{S}\mathrm{q}_2 \\ \mathrm{q}_1 2 &\Longrightarrow 0\mathrm{S}\mathrm{q}_2 \\ \mathrm{q}_1 3 &\Longrightarrow 0\mathrm{S}\mathrm{q}_2 \\ \mathrm{q}_1 \lambda &\Longrightarrow 0\mathrm{S}\mathrm{q}_2 \\ \mathrm{q}_1 \Lambda &\Longrightarrow 0\mathrm{S}\mathrm{q}_2 \end{aligned} \tag{5.3.4}$$

writes the symbol "0" to the cell scanned by the head unless it is the leftmost cell containing the marker "$\star$". Similar actions are performed by the machines WRITE(1), WRITE(2), WRITE(3), and WRITE(λ), the instructions for which can be obtained from (5.3.4) by replacing the symbol "0" in the right-hand sides of the instructions by "1", "2", "3", and "λ", respectively.

Machine READ(0) with instructions

$$\begin{aligned} \mathrm{q}_1\star &\Longrightarrow \star\mathrm{S}\mathrm{q}_3 \\ \mathrm{q}_1 0 &\Longrightarrow 0\mathrm{S}\mathrm{q}_2 \\ \mathrm{q}_1 1 &\Longrightarrow 1\mathrm{S}\mathrm{q}_3 \\ \mathrm{q}_1 2 &\Longrightarrow 2\mathrm{S}\mathrm{q}_3 \\ \mathrm{q}_1 3 &\Longrightarrow 3\mathrm{S}\mathrm{q}_3 \\ \mathrm{q}_1 \lambda &\Longrightarrow \lambda\mathrm{S}\mathrm{q}_3 \\ \mathrm{q}_1 \Lambda &\Longrightarrow \lambda\mathrm{S}\mathrm{q}_3 \end{aligned} \tag{5.3.5}$$

determines whether the cell scanned by the head contains the symbol "0" or not and then halts in state q_2 or q_3, accordingly; by our convention these correspond respectively to "YES" or "NO" answers. In a similar manner, machines READ(1), READ(2), READ(3) and READ($\star$) determine the presence of symbols "1", "2", "3", and "$\star$", respectively. Machine READ(λ) with instructions

$$\begin{aligned}
q_1\star &\Longrightarrow \star S q_3\\
q_1 0 &\Longrightarrow 0 S q_3\\
q_1 1 &\Longrightarrow 1 S q_3\\
q_1 2 &\Longrightarrow 2 S q_3\\
q_1 3 &\Longrightarrow 3 S q_3\\
q_1 \lambda &\Longrightarrow \lambda S q_2\\
q_1 \Lambda &\Longrightarrow \lambda S q_2
\end{aligned} \tag{5.3.6}$$

acts a bit differently because we have decided to consider cells containing the symbol "λ" as proxies for empty cells.

We also need the following two machines that make almost no changes on the tape. Machine STOP with instructions

$$\begin{aligned}
q_1\star &\Longrightarrow \star S q_3\\
q_1 0 &\Longrightarrow 0 S q_3\\
q_1 1 &\Longrightarrow 1 S q_3\\
q_1 2 &\Longrightarrow 2 S q_3\\
q_1 3 &\Longrightarrow 3 S q_3\\
q_1 \lambda &\Longrightarrow \lambda S q_3\\
q_1 \Lambda &\Longrightarrow \lambda S q_3
\end{aligned} \tag{5.3.7}$$

goes directly into the final state q_3 from the state q_1, without moving the head. On the other hand, machine NEVERSTOP with instructions

$$\begin{aligned}
q_1\star &\Longrightarrow \star S q_1\\
q_1 0 &\Longrightarrow 0 S q_1\\
q_1 1 &\Longrightarrow 1 S q_1\\
q_1 2 &\Longrightarrow 2 S q_1\\
q_1 3 &\Longrightarrow 3 S q_1\\
q_1 \lambda &\Longrightarrow \lambda S q_1\\
q_1 \Lambda &\Longrightarrow \lambda S q_1
\end{aligned} \tag{5.3.8}$$

never changes its state and hence never halts.

The action of machine

$$\text{READNOT}(0) = \textbf{while}\ \text{READ}(0)\ \textbf{do}\ \text{STOP}\ \textbf{od}$$

is the opposite of the action of machine READ(0); namely, it recognizes the absence of the symbol "0" in the cell observed by the head. Machines READNOT(2), READNOT(3), READNOT($\star$), and READNOT(λ) are obtained by replacing READ(0) by READ(2), READ(3), READ($\star$), and READ(λ), respectively.

Machine

$$\text{STAR} = \textbf{while}\ \text{READNOT}(\star)\ \textbf{do}\ \text{LEFT}\ \textbf{od}$$

puts the head into the leftmost cell (marked by "$\star$").

Machine

$$\text{VACANT} = \text{STAR}; \textbf{while}\ \text{READNOT}(\lambda)\ \textbf{do}\ \text{RIGHT}\ \textbf{od}$$

puts the head into the leftmost cell containing the symbol "λ" if such exists; otherwise it puts the head into the leftmost empty cell.

Machine

$$\text{JUMP} = \textbf{while}\ \text{READNOT}(0)\ \textbf{do}\ \text{RIGHT}\ \textbf{od}$$

moves the head to the right until the first cell containing the symbol "0" is reached; if all cells containing "0" are to the left of the head, then the machine will never halt.

A sequence of machines FIND is defined by the recurrent relations

$$\text{FIND}(1) = \text{STAR}; \text{JUMP}$$
$$\text{FIND}(k+1) = \text{FIND}(k); \text{RIGHT}; \text{JUMP}$$

Clearly, machine FIND(k) moves the head into the cell containing the symbol "0" that begins the representation of the element a_k from tuple (5.3.1).

Machine

$$\text{LAST} = \text{VACANT}; \textbf{while}\ \text{READNOT}(0)\ \textbf{do}\ \text{LEFT}\ \textbf{od}$$

acts on tuple (5.3.1) in the same way as machine FIND(n); namely, it moves the head into the cell containing the symbol "0" that begins the representation of the last element a_n. Thus, in an appropriate situation, the single machine LAST will perform the action of any one of the infinite sequence of machines FIND(k).

Machine

$$\text{NEW} = \text{VACANT}; \text{WRITE}(0)$$

transforms tuple (5.3.1) into tuple $\langle a_1, \ldots, a_n, 0\rangle$.

Machine

$$\text{INC} = \text{VACANT}; \text{WRITE}(1)$$

transforms tuple (5.3.1) into the tuple

$$\langle a_1, \ldots, a_{n-1}, a_n + 1\rangle. \tag{5.3.9}$$

On the other hand, machine

$$\text{DEC} = \text{VACANT}; \text{LEFT}; \textbf{if } \text{READ}(1) \textbf{ then } \text{WRITE}(\lambda)$$

transforms tuple (5.3.9) into tuple (5.3.1). When applied to a tuple with zero as its last element, it does not alter anything on the tape, but it halts in the state q_3, indicating that the last element has not been decreased.

Machine

$$\begin{aligned}\text{DELETE} = \text{VACANT};\ & \\ & \textbf{while } \text{READNOT}(0) \textbf{ do } \text{WRITE}(\lambda); \text{LEFT } \textbf{od}; \\ & \text{WRITE}(\lambda)\end{aligned}$$

truncates tuple (5.3.1), yielding the tuple $\langle a_1, \ldots, a_{n-1}\rangle$.

Machine

$$\text{MARK}(2) = \textbf{while } \text{RIGHT}; \text{READ}(1) \textbf{ do } \text{WRITE}(2) \textbf{ od}$$

replaces consecutive occurrences of the symbol "1" by the symbol "2". The machine MARK(2) is typically used after one of the machines FIND; thus, machine FIND(k); MARK(2) selects the element a_k from the representation of tuple (5.3.1).

Machine

$$\text{MARK}(3) = \textbf{while } \text{RIGHT}; \text{READ}(1) \textbf{ do } \text{WRITE}(3) \textbf{ od}$$

behaves in a similar manner.

Machine

$$\begin{aligned}\text{THEREIS}(2) = \text{STAR};\ & \\ & \textbf{while } \text{READNOT}(2) \textbf{ do} \\ & \quad \textbf{if } \text{READNOT}(\lambda) \textbf{ then } \text{RIGHT} \\ & \textbf{od}\end{aligned}$$

determines whether the symbol "2" occurs anywhere on the tape; if so, it halts in state q_2; otherwise it halts in state q_3. Our decision to construct all our machines in such a way that a cell containing the symbol "2" cannot be situated to the right of a cell containing the symbol "λ" is essential if machine THEREIS(2) is to perform correctly.

Machine

$$\text{THEREWAS}(2) = \textbf{if } \text{THEREIS}(2) \textbf{ then } \text{WRITE}(1)$$

not only determines the presence of the symbol "2" but also, having found a cell containing this symbol, replaces it by the symbol "1".

Machines THEREIS(3) and THEREWAS(3) are defined and act analogously to machines THEREIS(2) and THEREWAS(2), respectively.

Machines THEREWAS(2) and THEREWAS(3) restore the symbol "1" to just one cell. In contrast, machine

$$\begin{aligned}\text{RESTORE} = {} & \textbf{while } \text{THEREIS}(2) \textbf{ do } \text{THEREWAS}(2) \textbf{ od};\\ & \textbf{while } \text{THEREIS}(3) \textbf{ do } \text{THEREWAS}(3) \textbf{ od}\end{aligned}$$

restores all symbols "1" that machines MARK(2) and MARK(3) had earlier replaced by "2" or "3".

Machine

$$\begin{aligned}\text{APPEND}(k) = {} & \text{FIND}(k); \text{MARK}(2);\\ & \textbf{while } \text{THEREWAS}(2) \textbf{ do } \text{INC} \textbf{ od}\end{aligned}$$

transforms tuple (5.3.1) into $\langle a_1, \ldots, a_{n-1}, a_n + a_k\rangle$. Machine

$$\text{COPY}(k) = \text{NEW}; \text{APPEND}(k)$$

transforms tuple (5.3.1) into $\langle a_1, \ldots, a_n, a_k\rangle$. Finally, machine

$$\text{ADD}(k, l) = \text{COPY}(k); \text{APPEND}(l)$$

transforms tuple (5.3.1) into $\langle a_1, \ldots, a_n, a_k + a_l\rangle$.

For $k \neq l$, machine

$$\begin{aligned}\text{MULT}(k, l) = {} & \text{NEW}; \text{FIND}(k); \text{MARK}(3);\\ & \textbf{while } \text{THEREWAS}(3) \textbf{ do } \text{APPEND}(l) \textbf{ od}\end{aligned}$$

transforms tuple (5.3.1) into $\langle a_1, \ldots, a_n, a_k a_l\rangle$. To obtain tuple $\langle a_1, \ldots, a_n, a_k^2\rangle$, one can use machine

$$
\begin{aligned}
\text{MULT}(k,k) = {} & \text{COPY}(k); \text{LAST}; \text{MARK}(3);\\
& \textbf{while } \text{THEREWAS}(3) \textbf{ do}\\
& \quad \textbf{while } \text{THEREWAS}(3) \textbf{ do } \text{APPEND}(k) \textbf{ od}\\
& \textbf{od}
\end{aligned}
$$

For $k \neq l$, machine

$$
\begin{aligned}
\text{NOTGREATER}(k,l) = {} & \text{FIND}(k); \text{MARK}(2); \text{FIND}(l); \text{MARK}(3);\\
& \textbf{while } \text{THEREIS}(2) \textbf{ and } \text{THEREIS}(3) \textbf{ do}\\
& \quad \text{THEREWAS}(2); \text{THEREWAS}(3)\\
& \textbf{od};\\
& \textbf{while } \text{THEREIS}(2) \textbf{ do}\\
& \quad \text{RESTORE}; \text{STOP}\\
& \textbf{od};\\
& \text{RESTORE}
\end{aligned}
$$

compares a_k and a_l and stops in state q_2 or q_3 depending on which of the two inequalities $a_k \le a_l$ or $a_k > a_l$ holds. Similarly, machines

$$\text{EQUAL}(k,l) = \text{NOTGREATER}(k,l) \textbf{ and } \text{NOTGREATER}(l,k)$$

and

$$\text{NOTEQUAL}(k,l) = \textbf{while } \text{EQUAL}(k,l) \textbf{ do } \text{STOP} \textbf{ od}$$

determine whether a_k and a_l are equal.

Machine

$$
\begin{aligned}
\text{NEXT} = {} & \text{LAST}; \text{WRITE}(1); \text{RIGHT};\\
& \textbf{while } \text{READ}(\lambda) \textbf{ do } \text{WRITE}(1); \text{LAST}; \text{RIGHT} \textbf{ od};\\
& \text{WRITE}(0)
\end{aligned}
$$

transforms tuple (5.3.1) into the tuple $\langle a_1, \dots, a_{n-2}, b, c\rangle$, where $\langle b, c\rangle$ is the pair that follows immediately after the pair $\langle a_{n-1}, a_n\rangle$ in Cantor's list of all pairs (3.1.1). Finally, machine

$$
\begin{aligned}
\text{DECODE} = {} & \text{LAST}; \text{MARK}(2); \text{NEW}; \text{NEW};\\
& \textbf{while } \text{THEREWAS}(2) \textbf{ do } \text{NEXT} \textbf{ od}
\end{aligned}
$$

transforms tuple (5.3.1) into $\langle a_1, \dots, a_n, b, c\rangle$, where $\langle b, c\rangle$ is the pair with Cantor number a_n.

5.4 Turing machines can recognize Diophantine sets

In this section we establish our first connection between Diophantine sets and Turing machines. Namely, we show that *given any parametric Diophantine equation*

$$D(a_1, \dots, a_n, x_1, \dots, x_{m+1}) = 0, \tag{5.4.1}$$

one can construct a Turing machine M that will eventually halt, beginning with a representation of the tuple

$$\langle a_1, \dots, a_n\rangle, \tag{5.4.2}$$

if and only if equation (5.4.1) *is solvable in the unknowns* $x_1, \dots, x_{m+1}$.

We begin by constructing an auxiliary machine M_1 that, given a tuple

$$\langle a_1, \dots, a_n, y_0\rangle, \tag{5.4.3}$$

determines whether y_0 is the Cantor number of a tuple $\langle x_1, \dots, x_{m+1}\rangle$ that does not satisfy equation (5.4.1). This machine M_1 is itself constructed by concatenating a number of other machines using the operation ";". First, we use m copies of the machine DECODE, so that the tuple (5.4.3) is transformed into the tuple

$$\langle a_1, \dots, a_n, y_0, x_1, y_1, \dots, x_m, y_m\rangle, \tag{5.4.4}$$

where

$$y_{k-1} = \mathrm{Cantor}(x_k, y_k), \tag{5.4.5}$$

and hence

$$y_0 = \mathrm{Cantor}_{m+1}(x_1, \dots, x_m, y_m). \tag{5.4.6}$$

Equation (5.4.1) can be rewritten in the form

$$C_{\mathrm{L}}(a_1, \dots, a_n, x_1, \dots, x_m, y_m) = C_{\mathrm{R}}(a_1, \dots, a_n, x_1, \dots, x_m, y_m), \tag{5.4.7}$$

where C_{L} and C_{R} are polynomials whose coefficients are natural numbers and the unknown x_{m+1} has been replaced by y_m. The computation of the values of C_{L}

and C_{R} can be represented as a sequence of elementary operations

$$\begin{aligned} z_1 &= \alpha_1 \circ_1 \beta_1, \\ &\vdots \\ z_k &= \alpha_k \circ_k \beta_k, \end{aligned} \tag{5.4.8}$$

where each of α_i and β_i, $i = 1, \dots, k$, is one of the quantities $1, a_1, \dots, a_n, x_1, \dots, x_m, y_m, z_1, \dots, z_{k-1}$, where each $\circ_i$ is one of the symbols $+$ or $\cdot$, and where the values of C_{L} and C_{R} occur among the quantities $z_1, \dots, z_k$. It is clear that the machines NEW, INC, ADD, and MULT can be combined to transform tuple (5.4.4) into the tuple

$$\langle a_1, \dots, a_n, y_0, x_1, y_1, \dots, x_m, y_m, 1, z_1, \dots, z_k \rangle. \tag{5.4.9}$$

To complete the construction of machine M_1, it suffices to apply the machine NOTEQUAL with appropriate arguments. Thus, machine M_1 will halt in state q_2 if y_0 from tuple (5.4.2) is *not* the Cantor number of a solution of equation (5.4.1), and in state q_3 if it is.

Let M_2 denote the machine DELETE; DELETE; ... DELETE, where the machine DELETE is iterated $2m + k + 1$ times. Clearly, M_2 transforms tuple (5.4.9) into the original tuple (5.4.3).

Now we can define the machine M as

$$M = \text{NEW}; \textbf{while } M_1 \textbf{ do } M_2; \text{INC} \textbf{ od} \tag{5.4.10}$$

which behaves as follows: It begins (using machine NEW) by transforming tuple (5.4.2) into the tuple

$$\langle a_1, \dots, a_n, 0 \rangle. \tag{5.4.11}$$

If machine M_1 determines that 0 is the Cantor number of a solution of equation (5.4.1), then machine M immediately halts. Otherwise, machines M_2 and INC act to construct the tuple

$$\langle a_1, \dots, a_n, 1 \rangle, \tag{5.4.12}$$

and machine M_1 proceeds to determine whether 1 is the Cantor number of a solution, and so on. If equation (5.4.1) has a solution, then machine M will eventually halt; if equation (5.4.1) has no solution, then machine M will continue forever.

5.5 Diophantine simulation of Turing machines

In the previous section we showed that Diophantine equations are "semidecidable"; i.e., if an equation has a solution, then this fact can be revealed purely mechanically. This "semidecidability" is an intuitively obvious property, and the formal proof given in the previous section does not contribute much to our knowledge about Diophantine equations. Instead, the proof gives us evidence that Turing machines are sufficiently powerful, in spite of their primitive set of instructions.

In fact, the main role of the previous section was to motivate the introduction of a new notion. Namely, we shall say that a set $\mathfrak{M}$ of n-tuples of natural numbers is *Turing semidecidable* if there is a Turing machine M that, beginning in state q_1, with a tape containing the canonical representation of the tuple $\langle a_1, \dots, a_n \rangle$, and with its head scanning the leftmost cell on the tape, will eventually halt if and only if $\langle a_1, \dots, a_n \rangle \in \mathfrak{M}$. In this case, we say that M *semidecides* $\mathfrak{M}$.

In the previous section we proved that every Diophantine set is Turing semidecidable. The aim of the present section is to establish the converse implication: *every Turing semidecidable set is Diophantine.*

Let M be a Turing machine that semidecides a set $\mathfrak{M}$ consisting of n-tuples of natural numbers. Let

$$\{\alpha_1, \dots, \alpha_w\} \tag{5.5.1}$$

be the alphabet of M. As the machine M carries out its operations, at each moment the symbols of (5.5.1) occupy only a finite initial segment of the tape of length, say, l, and we can therefore represent the tape by the tuple

$$\langle s_1, s_2, \dots, s_m, \dots, s_{l-1}, s_l \rangle \tag{5.5.2}$$

consisting of the subscripts of the symbols occurring in the cells (in particular, s_1 is always equal to the subscript of the symbol "$\star$" in (5.5.1)).

The current state q_i and the position of the head can be represented by a tuple of the same length,

$$\langle 0, \dots, 0, i, 0, \dots, 0 \rangle, \tag{5.5.3}$$

in which all elements but one are zero; the only non-zero element is equal to the subscript of the state, and its position corresponds to the position of the head.

The triple consisting of the current contents of the tape, the state, and the position of the head will be called the *configuration*. Clearly, tuples (5.5.2) and (5.5.3) uniquely determine the configuration. To represent these tuples, we shall use positional coding with a fixed base β that must be no less than 3, greater than v, the number of states of machine M, and greater than w, the number of symbols in the alphabet (5.5.1). The pair $\langle p, t\rangle$ is called a *configuration code* if p and t are ciphers of tuples (5.5.3) and (5.5.2), respectively, to the base β. A configuration code does not specify the length of tuples (5.5.2) and (5.5.3), but this quantity is of no significance because tuple (5.5.3) has only one non-zero element, while zero elements of tuple (5.5.2) will be interpreted as empty cells. Thus, the configuration is uniquely determined by a configuration code, and our use of the ciphers of tuples (5.5.2) and (5.5.3) (rather than their codes) corresponds to the potential infinity of the tape.

Using this terminology, our first goal will be to construct a Diophantine equation

$$D(p, t, x_1, \ldots, x_m) = 0 \tag{5.5.4}$$

such that *if* $\langle p, t\rangle$ *is the code of a configuration, then equation* (5.5.4) *is solvable in* $x_1, \ldots, x_m$ *if and only if machine* M*, beginning in this configuration, eventually halts.* We shall not be concerned about whether or not equation (5.5.4) is solvable when $\langle p, t\rangle$ is not a configuration code.

Naturally, we begin by simulating a single step of a Turing machine. Thus, let machine M proceed directly from the configuration with code $\langle p, t\rangle$ to the configuration with code

$$\langle \mathrm{NextP}(p, t), \mathrm{NextT}(p, t)\rangle.$$

We will now verify that NextP and NextT are Diophantine functions.

We need to make this statement more precise. So far we have defined the functions NextP and NextT only when $\langle p, t\rangle$ is a code of a configuration with a non-final state. We now set

$$\mathrm{NextP}(p, t) = 0, \tag{5.5.5}$$

$$\mathrm{NextT}(p, t) = t \tag{5.5.6}$$

if $\langle p, t\rangle$ is a configuration code with a final state. We shall understand the statement that NextP and NextT are Diophantine as meaning that there exist Diophantine functions that are equal to NextP and NextT when $\langle p, t\rangle$ is a configuration code, but which may have any value, or even be undefined, when it is not.

Clearly, the functions NextP and NextT are determined by the functions A, D, and Q from the set (5.1.1) of instructions of machine M. Following (5.5.5) and (5.5.6), we extend these functions, setting $A(i,j) = j$, $D(i,j) = S$, and $Q(i,j) = 0$ when q_i is a final state.

The fact that the function NextT is Diophantine is almost obvious. Indeed, each element of the tuple with cipher $\mathrm{NextT}(p,t)$ is uniquely determined by the elements in the same position from the tuples with ciphers p and t. Let us define the function A by

$$\mathrm{A}(i,j) = \begin{cases} A(i,j) & \text{if } 0 < i \le v,\ 0 \le j \le w, \\ j & \text{otherwise.} \end{cases} \tag{5.5.7}$$

(Recall that v is the number of states of the machine M and w is the number of symbols in its alphabet.) Now we can define NextT by

$$t' = \mathrm{NextT}(p,t) \iff \exists w\,[t' = \mathrm{A}[\beta](p,t,w)], \tag{5.5.8}$$

where $\mathrm{A}[\beta]$ is the extension of A to tuples introduced in Section 3.6.

The fact that the function NextP is also Diophantine is a bit less evident. Observe that the kth element of the tuple with cipher $\mathrm{NextP}(p,t)$ is uniquely determined by the $(k-1)$th, kth, and $(k+1)$th elements of the tuples with ciphers p and t. The technique introduced in Section 3.6 only allows us to extend functions element-wise, but it is not difficult to overcome this obstacle. Put

$$p^{\mathrm{R}} = p\beta, \qquad p^{\mathrm{L}} = p \operatorname{div} \beta, \tag{5.5.9}$$

$$t^{\mathrm{R}} = t\beta, \qquad t^{\mathrm{L}} = t \operatorname{div} \beta. \tag{5.5.10}$$

Clearly, if t is the cipher of tuple (5.5.2), then t^{R} and t^{L} are the ciphers of the tuples

$$\langle 0, s_1, \ldots, s_{m-1}, \ldots, s_{l-1}, s_l \rangle \tag{5.5.11}$$

and

$$\langle s_2, s_3, \ldots, s_{m+1}, \ldots, s_l, 0 \rangle, \tag{5.5.12}$$

respectively (it is the mth elements that are shown explicitly in tuples (5.5.2), (5.5.11), and (5.5.12)). Similarly, if p is the cipher of tuple (5.5.3), then p^{R} and p^{L} are the ciphers of the tuples

$$\langle 0, \ldots, 0, 0, i, \ldots, 0 \rangle \tag{5.5.13}$$

and

$$\langle 0, \ldots, i, 0, 0, \ldots, 0 \rangle, \tag{5.5.14}$$

where the non-zero element in (5.5.3) has been shifted by one position to the right or the left, respectively.

Now every element of the tuple with cipher $\mathrm{NextP}(p,t)$ is uniquely determined by the elements in the same position in the tuples with ciphers p^{L}, p, p^{R}, t^{L}, t, t^{R} in a manner determined by the functions D and Q. To be more precise, let us introduce the function DQ by

$$\mathrm{DQ}(i^{\mathrm{L}},i,i^{\mathrm{R}},j^{\mathrm{L}},j,j^{\mathrm{R}}) = \begin{cases} Q(i^{\mathrm{L}},j^{\mathrm{L}}) & \text{if } i^{\mathrm{L}}>0,\ \ i=i^{\mathrm{R}}=0, \text{ and } D(i^{\mathrm{L}},j^{\mathrm{L}})=\mathrm{L}, \\ Q(i,j) & \text{if } i^{\mathrm{L}}=0,\ \ i>0,\ \ i^{\mathrm{R}}=0, \text{ and } D(i,j)=\mathrm{S}, \\ Q(i^{\mathrm{R}},j^{\mathrm{R}}) & \text{if } i^{\mathrm{L}}=i=0,\ i^{\mathrm{R}}>0, \text{ and } D(i^{\mathrm{R}},j^{\mathrm{R}})=\mathrm{R}, \\ 0 & \text{otherwise} \end{cases} \tag{5.5.15}$$

(the "otherwise" also includes the cases when some of the numbers i^{L}, i, or i^{R} are greater than v and when some of the numbers j^{L}, j, or j^{R} are greater than w, because Q and D are not defined in those cases). Now we are in position to give a Diophantine representation for NextP:

$$p' = \mathrm{NextP}(p,t) \iff \exists w\,[p' = \mathrm{DQ}[\beta](p\beta, p, p \operatorname{div} \beta, t\beta, t, t \operatorname{div}\beta, w)]. \tag{5.5.16}$$

The functions NextP and NextT simulate a single step of our Turing machine. Multiple steps are similarly simulated by the 3-argument functions AfterP and AfterT defined as iterations of functions NextP and NextT:

$$\begin{aligned} \mathrm{AfterP}(0,p,t) &= p, \\ \mathrm{AfterT}(0,p,t) &= t, \\ \mathrm{AfterP}(k+1,p,t) &= \mathrm{NextP}(\mathrm{AfterP}(k,p,t),\mathrm{AfterT}(k,p,t)), \\ \mathrm{AfterT}(k+1,p,t) &= \mathrm{NextT}(\mathrm{AfterP}(k,p,t),\mathrm{AfterT}(k,p,t)). \end{aligned} \tag{5.5.17}$$

Clearly, if machine M passes from configuration $\langle p,t\rangle$ to configuration $\langle p',t'\rangle$ in k steps, then $p' = \mathrm{AfterP}(k,p,t)$ and $t' = \mathrm{AfterT}(k,p,t)$. Next we will show that the functions AfterP and AfterT are also Diophantine.

Let us consider all of the intermediate configurations

$$\langle p_0,t_0\rangle,\ldots,\langle p_k,t_k\rangle, \tag{5.5.18}$$

where

$$\langle p_0, t_0\rangle = \langle p, t\rangle, \tag{5.5.19}$$

$$\langle p_{i+1}, t_{i+1}\rangle = \langle \mathrm{NextP}(p_i, t_i), \mathrm{NextT}(p_i, t_i)\rangle, \tag{5.5.20}$$

$$\langle p', t'\rangle = \langle p_k, t_k\rangle. \tag{5.5.21}$$

Let the number l be so large that

$$p < \beta^{l-k-2}, \qquad t < \beta^{l-k-2}. \tag{5.5.22}$$

This implies that in the configuration coded by $\langle p, t\rangle$, no more than $l-k-2$ initial cells are occupied by symbols. In k steps machine M can fill no more than k additional cells, and hence for $i = 0, \ldots, k$,

$$p_i < \beta^{l-2}, \qquad t_i < \beta^{l-2}; \tag{5.5.23}$$

in particular,

$$p' < \beta^{l-2}, \qquad t' < \beta^{l-2}. \tag{5.5.24}$$

Let us construct a pair of *superconfigurations* $\langle p_{\mathrm{L}}, t_{\mathrm{L}}\rangle$ and $\langle p_{\mathrm{R}}, t_{\mathrm{R}}\rangle$ by concatenating the configurations (5.5.18) (recall that in Section 3.3 we showed that concatenation of tuples with equal bases is a Diophantine operation):

$$\langle p_{\mathrm{L}}, \beta, kl\rangle = \langle p_0, \beta, l\rangle + \cdots + \langle p_{k-1}, \beta, l\rangle, \tag{5.5.25}$$

$$\langle t_{\mathrm{L}}, \beta, kl\rangle = \langle t_0, \beta, l\rangle + \cdots + \langle t_{k-1}, \beta, l\rangle, \tag{5.5.26}$$

$$\langle p_{\mathrm{R}}, \beta, kl\rangle = \langle p_1, \beta, l\rangle + \cdots + \langle p_k, \beta, l\rangle, \tag{5.5.27}$$

$$\langle t_{\mathrm{R}}, \beta, kl\rangle = \langle t_1, \beta, l\rangle + \cdots + \langle t_k, \beta, l\rangle. \tag{5.5.28}$$

The pairs $\langle p_{\mathrm{L}}, t_{\mathrm{L}}\rangle$ and $\langle p_{\mathrm{R}}, t_{\mathrm{R}}\rangle$ do not code configurations because the tuples with ciphers p_{L} and p_{R} contain k non-zero elements and the tuples with ciphers t_{L} and t_{R} correspond to tapes with k cells marked by the symbol "$\star$".

The superconfigurations correspond to a *supermachine*, whose tape is divided by the symbols "$\star$" into k parts on each of which a separate head operates according to the instructions of the original machine M. Instead of k steps of the machine M, we can deal with a single step of the supermachine, during which each of the heads carries out the corresponding instruction.

Thus, the superconfiguration $\langle p_{\mathrm{R}}, t_{\mathrm{R}}\rangle$ is uniquely determined by the superconfiguration $\langle p_{\mathrm{L}}, t_{\mathrm{L}}\rangle$. Moreover, the functions NextP and NextT, as defined by (5.5.16) and (5.5.8) without modification, already describe the operation of the supermachine:

$$p_{\mathrm{R}} = \mathrm{NextP}(p_{\mathrm{L}}, t_{\mathrm{L}}), \qquad t_{\mathrm{R}} = \mathrm{NextT}(p_{\mathrm{L}}, t_{\mathrm{L}}). \tag{5.5.29}$$

Let us also consider the superconfiguration $\langle p_{\mathrm{M}}, t_{\mathrm{M}}\rangle$ defined by

$$\langle p_{\mathrm{M}}, \beta, (k-1)l\rangle = \langle p_1, \beta, l\rangle + \cdots + \langle p_{k-1}, \beta, l\rangle, \tag{5.5.30}$$

$$\langle t_{\mathrm{M}}, \beta, (k-1)l\rangle = \langle t_1, \beta, l\rangle + \cdots + \langle t_{k-1}, \beta, l\rangle. \tag{5.5.31}$$

Using this notation, equations (5.5.25)–(5.5.26) can be rewritten as

$$\langle p_{\mathrm{L}}, \beta, kl\rangle = \langle p, \beta, l\rangle + \langle p_{\mathrm{M}}, \beta, (k-1)l\rangle, \tag{5.5.32}$$

$$\langle t_{\mathrm{L}}, \beta, kl\rangle = \langle t, \beta, l\rangle + \langle t_{\mathrm{M}}, \beta, (k-1)l\rangle, \tag{5.5.33}$$

$$\langle p_{\mathrm{R}}, \beta, kl\rangle = \langle p_{\mathrm{M}}, \beta, (k-1)l\rangle + \langle p', \beta, l\rangle, \tag{5.5.34}$$

$$\langle t_{\mathrm{R}}, \beta, kl\rangle = \langle t_{\mathrm{M}}, \beta, (k-1)l\rangle + \langle t', \beta, l\rangle. \tag{5.5.35}$$

We have seen that for any configuration $\langle p, t\rangle$ and any positive k, if l satisfies (5.5.22), then there are numbers p_{L}, t_{L}, p_{M}, t_{M}, p_{R}, t_{R}, p', t' satisfying conditions (5.5.24), (5.5.29), and (5.5.32)–(5.5.35). We now show that these numbers are uniquely determined by these same conditions in terms of k, l, p, and t.

In fact, by (5.5.32) and (5.5.33), the first l elements of the tuples with codes $\langle p_{\mathrm{L}}, b, kl\rangle$ and $\langle t_{\mathrm{L}}, \beta, kl\rangle$ are uniquely determined. The functions NextP and NextT were defined by (5.5.16) and (5.5.8) in such a manner that the first m elements of the tuples with ciphers $\mathrm{NextP}(x, y)$ and $\mathrm{NextT}(x, y)$ are uniquely determined by the first $m+1$ elements of the tuples with ciphers x and y. Hence, by (5.5.29), the first $l-1$ elements of the tuples with codes $\langle p_{\mathrm{R}}, \beta, kl\rangle$ and $\langle t_{\mathrm{R}}, \beta, kl\rangle$ are also uniquely determined. Then it follows from (5.5.34) and (5.5.35) that the first $l-1$ elements of the tuples with codes $\langle p_{\mathrm{M}}, \beta, (k-1)l\rangle$ and $\langle t_{\mathrm{M}}, \beta, (k-1)l\rangle$ are also unique.

Returning to (5.5.32) and (5.5.33), we see that the first $2l-1$ elements of the tuples with codes $\langle p_{\mathrm{L}}, \beta, kl\rangle$ and $\langle t_{\mathrm{L}}, \beta, kl\rangle$ are uniquely determined. Hence, by (5.5.29), the first $2l-2$ elements of the tuples with codes $\langle p_{\mathrm{R}}, \beta, kl\rangle$ and $\langle t_{\mathrm{R}}, \beta, kl\rangle$ are unique, and, by (5.5.34) and (5.5.35), the first $2l-2$ elements of the tuples with codes $\langle p_{\mathrm{M}}, \beta, (k-1)l\rangle$ and $\langle t_{\mathrm{M}}, \beta, (k-1)l\rangle$ are unique as well.

Repeating this reasoning sufficiently often, we establish the uniqueness of all elements of the tuples with codes

$$\langle p_{\mathrm{L}}, \beta, kl\rangle, \langle t_{\mathrm{L}}, \beta, kl\rangle, \langle p_{\mathrm{M}}, \beta, (k-1)l\rangle, \langle t_{\mathrm{M}}, \beta, (k-1)l\rangle$$

and all, except possibly the last, elements of the tuples with codes

$$\langle p_{\mathrm{R}}, \beta, kl\rangle, \langle t_{\mathrm{R}}, \beta, kl\rangle, \langle p', \beta, l\rangle, \langle t', \beta, l\rangle.$$

In addition, the inequality (5.5.24) implies that the last elements of the tuples with codes $\langle p', \beta, l\rangle$ and $\langle t', \beta, l\rangle$ are equal to 0, and hence, by (5.5.34) and (5.5.35), so are the last elements of the tuples with codes $\langle p_{\mathrm{R}}, \beta, kl\rangle$ and $\langle t_{\mathrm{R}}, \beta, kl\rangle$.

Thus, we have shown that *for any configuration* $\langle p, t\rangle$*, any positive* k*, and for every* l *satisfying* (5.5.22)*, the system of Diophantine conditions* (5.5.24)*,* (5.5.29)*,* (5.5.32)–(5.5.35) *has exactly one solution in the variables* p_L*,* t_L*,* p_M*,* t_M*,* p_R*,* t_R*,* p'*,* t'*, and for this unique solution, we have* $p' = \mathrm{AfterP}(k, p, t)$ *and* $t' = \mathrm{AfterT}(k, p, t)$. This immediately implies that the functions AfterP and AfterT are Diophantine.

To construct the Diophantine equation (5.5.4), we need to take one additional small step. Let $\omega_1, \ldots, \omega_z$ be the subscripts of the final states of machine M. Now, the condition

$$\exists kr \left[\mathrm{Elem}\big(\mathrm{AfterT}(k, p, t), \beta, r\big) = \omega_1 \vee \cdots \vee \mathrm{Elem}\big(\mathrm{AfterT}(k, p, t), \beta, r\big) = \omega_z\right] \tag{5.5.36}$$

holds for the code $\langle p, t\rangle$ of a configuration if and only if the machine, beginning in that configuration, will eventually halt (with k being equal to the number of steps carried out by the machine). Condition (5.5.36) is Diophantine and hence can be transformed into the required equation (5.5.4).

However, equation (5.5.4) is not quite what we need in order to prove that the set $\mathfrak{M}$, semidecidable by machine M, is Diophantine, because the parameters of (5.5.4) are p and t from the code of a configuration, rather than $a_1, \ldots, a_n$ as required in a Diophantine representation of a set. Nevertheless, it is easy to see that in the initial configuration,

$$p = 1 \tag{5.5.37}$$

(the machine is in state q_1 with its head scanning the leftmost cell) and

$$\begin{aligned}\langle t, \beta, a\rangle = \langle \kappa, \beta, 1\rangle + \langle \mu, \beta, 1\rangle + \langle \mathrm{Repeat}(\nu, \beta, a_1), \beta, a_1\rangle + \cdots \\ + \langle \mu, \beta, 1\rangle + \langle \mathrm{Repeat}(\nu, \beta, a_n), \beta, a_n\rangle,\end{aligned} \tag{5.5.38}$$

where $a = a_1 + \cdots + a_n + n + 1$ is the number of occupied cells and κ, μ, and ν are the subscripts of the symbols "$\star$", "0", and "1", respectively in (5.5.1) (i.e., $\alpha_\kappa = \star$, $\alpha_\mu = 0$, $\alpha_\nu = 1$). Combining (5.5.4), (5.5.37), and (5.5.38) and regarding $a_1, \ldots, a_n$ as parameters, we obtain the required Diophantine representation of the set $\mathfrak{M}$.

5.6 Hilbert's Tenth Problem is undecidable by Turing machines

In Section 5.5 we introduced the notion of Turing semidecidable set, and we established the equivalence of that notion with the notion of Diophantine set. Now, as soon as we introduce one other notion, we shall be in a position to give a more precise formulation of Hilbert's Tenth Problem. Namely, we shall say that a set $\mathfrak{M}$ of n-tuples of natural numbers is *Turing decidable* if there is a Turing machine M that, beginning in state q_1, with the head scanning the leftmost cell of a tape containing the canonical representation of the tuple $\langle a_1, \ldots, a_n \rangle$, will eventually halt in state q_2 if $\langle a_1, \ldots, a_n \rangle \in \mathfrak{M}$ and in state q_3 otherwise.

It is natural to ask what relationship there is between Turing decidability and semidecidability. It is easy to see that *if a set $\mathfrak{M}$ is Turing decidable, then it is Turing semidecidable.* In fact, if M is a Turing machine yielding the decidability of $\mathfrak{M}$, then the machine

$$\textbf{while } M \textbf{ do } \textsc{stop} \textbf{ od}; \textsc{neverstop} \tag{5.6.1}$$

semidecides the set $\mathfrak{M}$.

It is also easy to see that *if a set is decidable, then its complement is also semidecidable.* For this, it suffices to consider the machine

$$\textbf{while } M \textbf{ do } \textsc{neverstop} \textbf{ od} \tag{5.6.2}$$

instead of (5.6.1).

It is not so evident that the converse is valid too: *if both a set $\mathfrak{M}$ and its complement are Turing semidecidable, then $\mathfrak{M}$ is Turing decidable.* A straightforward approach to a proof of this result might go as follows. Let M_2 and M_3 be Turing machines that semidecide the set $\mathfrak{M}$ and its complement, respectively. One can construct a third machine M that simulates the operations of the two machines M_2 and M_3 on its single tape, performing in turn the successive steps of each machine. (Naturally, simulating one step of M_2 or M_3 requires machine M to perform many steps.) Eventually, one of the two machines, M_2 or M_3, will halt, and then machine M will enter the final state q_2 or q_3, respectively.

The technical details involved in actually constructing a machine that simultaneously simulates two given machines are quite complicated. However, taking advantage of the equivalence, established above, of the class of Turing semidecidable sets with the class of Diophantine sets, we can give a shorter proof. Instead of simulating Turing machines, we construct a pair of Diophantine equations

$$D_2(a_1, \ldots, a_n, x_1, \ldots, x_m) = 0 \tag{5.6.3}$$

and

$$D_3(a_1, \ldots, a_n, x_1, \ldots, x_m) = 0, \tag{5.6.4}$$

respectively defining the set $\mathfrak{M}$ and its complement (without loss of generality, we assume that both equations have the same number of unknowns). The two equations can be combined into a single equation

$$\begin{aligned}(D_2^2(a_1, \ldots, a_n, x_1, \ldots, x_m) + (1 - y_m)^2)\\ \cdot (D_3^2(a_1, \ldots, a_n, x_1, \ldots, x_m) + y_m) = 0.\end{aligned} \tag{5.6.5}$$

Clearly, this equation has a solution for any value of the parameters, and moreover, $\langle a_1, \ldots, a_n \rangle \in \mathfrak{M}$ if and only if

$$y_m = 1 \tag{5.6.6}$$

in such a solution. Let M be machine (5.4.10) constructed with equation (5.6.5) substituted for equation (5.4.1). When M halts, the tape will contain the representation of a tuple of the form (5.4.9). A machine to decide $\mathfrak{M}$ can then be defined as $M; \textsc{equal}(l, l+1)$, where $l = n + 1 + 2m$; this machine determines whether equation (5.6.6) holds because the constant 1 and the value of y_m are the lth and $l+1$st elements, respectively, in the tuple (5.4.9).

Now we can reformulate Hilbert's Tenth Problem in the following, more explicit way: *is the set of codes of all solvable Diophantine equations (without parameters) Turing decidable*? This set was considered in Section 4.5, where it was denoted by $\mathfrak{H}_0$. There we established that $\overline{\mathfrak{H}_0}$, the complement of $\mathfrak{H}_0$, is not Diophantine. This implies that $\mathfrak{H}_0$ is not Turing decidable. In other words, *it is impossible to construct a Turing machine that, beginning with the representation of a number k on the tape, will halt after a finite number of steps in state* q_2 *or* q_3, *depending on whether the equation with code k is or is not solvable.*

The set $\mathfrak{H}_0$ is itself Diophantine, since it is defined by equation (4.6.7). This fact shows that Turing machines are incapable of deciding whether or not the equations belonging *to one particular family of Diophantine equations* have solutions, to say nothing of *arbitrary Diophantine equations.*

5.7 Church's Thesis

In the previous sections of this chapter we have obtained two, no doubt remarkable, results. Namely, we have established that

the class of Diophantine sets
is identical to (5.7.1)
the class of Turing semidecidable sets

and

Hilbert's Tenth Problem is Turing undecidable. (5.7.2)

However, these two rather technical results raise a number of new questions. While the definition of *Diophantine set* is quite natural, that of *Turing semidecidable set* is burdened by numerous technical details, many of which seem arbitrary. For example, we could use binary rather than unary notation to represent numbers. The tape could be infinite in both directions instead of in only one direction. Instead of a single head, there might be several, each executing its own set of instructions while sharing information about their respective scanned cells. Moreover, there might be several tapes. In fact, the memory need not even be linear; it could, for example, take the form of a plane divided into square cells. For each such modification of the notion of Turing machine, one can introduce a corresponding concept of semidecidability and pose the question of how that concept is related to Diophantine sets.

The reformulation of Hilbert's Tenth Problem used in Section 5.6 could be criticized as well, since it is based on a very special method for coding Diophantine equations. It would be more natural to write on the tape the number of unknowns, the degree, and the coefficients of an equation in unary or some positional notation. Hilbert did not impose any restrictions on the desired method for solving the Tenth Problem. Thus, if for some appropriate notation for polynomials, someone had succeeded in constructing a decision machine, that would certainly have provided a positive solution of Hilbert's Tenth Problem. So to what extent can the Turing undecidability established in Section 5.6 be considered as constituting a negative solution?

For every modification of the notion of Turing machine (or any other abstract computing device) and for every method of representing initial data on the tape, one could try to obtain results like (5.7.1) and (5.7.2). This could be done directly or indirectly. For the former, we would need to introduce suitable codings and to prove that the corresponding relations are Diophantine. For the latter, we can forget about Diophantine equations for the time being and prove that the class of semidecidable sets remains the same in spite of modifications in the definition (see, e.g., Exercises 5.6 and 5.7a). Such studies of relations between (semi)decidability on different abstract computing devices were in fact carried out quite apart from work on Hilbert's Tenth Problem, and long before its undecidability was established in any sense. These studies showed that the notions of semidecidable set and decidable set, which we introduced using Turing machines, are quite insensitive to the choice of a particular computing device and method of representing initial data. The very same classes arose for any "reasonable" choice. By "reasonable" we mean that the elementary steps of a computation should in fact be elementary, i.e., easily performable (for example, one cannot recognize in a single step whether a particular Diophantine equation has a solution).

Long before the appearance of the first mathematically rigorous notions of abstract computing devices such as Turing machines, there was the intuitive notion of an algorithm as a guaranteed method for the mechanical solution of problems of a definite kind. As a classical example, we can mention 's algorithm for finding the greatest common divisor of two positive integers. Whenever an algorithm was proposed, it was clear that in fact it was a universal method for solving problems of the appropriate kind. However, such intuitive notions do not suffice for a proof that there is no algorithm for problems of the kind being considered.

The initial data for an algorithm (in the intuitive sense) are selected from some countable set, and, essentially without loss of generality, we are only going to consider situations in which the initial data consist of natural numbers or tuples of natural numbers of some fixed length. (The techniques of Chapter 4 indicate how one could code more complicated structures.)

The result of carrying out an algorithm is also an object of the appropriate kind. We could have chosen to consider only algorithms whose outputs are natural numbers, but it is more in the spirit of the subject of this book to consider instead algorithms with the two outputs "YES" and "NO." Correspondingly, along with the intuitive concept of algorithm, two related concepts arise, namely, the intuitive concepts of decidable and semidecidable set. A set $\mathfrak{M}$ of n-tuples is decidable (in the intuitive sense) if there exists an algorithm (also in the intuitive sense) that halts on every n-tuple of natural numbers and reports "YES" or "NO" depending on whether the n-tuple belongs to the set or not. Similarly, for $\mathfrak{M}$ to be semidecidable we would need an algorithm that would report "YES" for every n-tuple in $\mathfrak{M}$ and that would either report "NO" or fail to halt if the tuple doesn't belong to $\mathfrak{M}$. Computability theory in its entirety could be expounded in terms of decidable and semidecidable sets, just as we can eliminate the notion of functions from mathematics and deal only with their graphs.

Between intuitive decidability and semidecidability there is the same relation as between Turing decidability and semidecidability: a set is decidable if and only if both it and its complement are semidecidable.

How is formal Turing (semi)decidability related to (semi)decidability in the intuitive sense? One relation is evident: Turing (semi)decidable sets are recognized by most mathematicians as (semi)decidable in the intuitive sense. The converse is known as

CHURCH'S THESIS: *Every set of n-tuples that is (semi)decidable in the intuitive sense is also Turing (semi)decidable.*

Here we have encountered something rarely found in mathematics, a *thesis*. What is it? It is not a theorem because it has no proof. It is not a conjecture because it cannot have a proof. It is not even an axiom that we are free to accept or reject. All of this is due to the fact that Church's Thesis is not a precise mathematical statement, because it relates the rigorous notion of Turing (semi)decidability with the non-rigorous notion of (semi)decidability in the intuitive sense.

What are the arguments in favor of Church's Thesis? They may be very different in appearance but all of them are essentially the same: *so far no one has found an example of a set that would be recognized by most mathematicians as (semi)decidable in the intuitive sense and for which we cannot construct an appropriate Turing machine.* Of course, this assertion should not be understood literally, i.e., as though for every set recognized to be (semi)decidable someone had actually constructed a corresponding machine. In fact, there are numerous tools for indirectly proving the existence of such a machine.

What is Church's Thesis for? On the one hand, it can serve as a guiding star: as soon as we have established (semi)decidability in the intuitive sense, our chance of finding the corresponding Turing machine should be considered very high. In fact, professional mathematicians usually content themselves with establishing intuitive (semi)decidability and do not condescend to formal proofs.

On the other hand, Church's Thesis plays a role in mathematics similar to that played elsewhere by the law of conservation of energy. Namely, as long as no exception to the law is found, it is unreasonable to set out to construct a perpetual motion machine. Similarly, once the Turing undecidability of a set has been proved, one need not spend one's time seeking a universal method for recognizing the elements of that set. In particular, according to Church's Thesis, the quite special result (5.7.2) gives us the moral right to cease the hunt (so far carried out in vain) for a "process" of the sort Hilbert asked for in his Tenth Problem.

In Chapter 7 we shall discuss the question of whether (5.7.2) constitutes a complete solution of Hilbert's Tenth Problem from another point of view.

Exercises

1. Construct a Turing machine DOUBLE that will transform tuple (5.3.1) into the tuple $\langle a_1, \dots, a_{n-1}, 2a_n \rangle$ for any n.

2. Construct a Turing machine ERASE that will transform tuple (5.3.1) into the tuple $\langle a_2, \dots, a_n \rangle$.

3. Let the *binary representation of a number* on tape consist of consecutive cells containing the symbols "0" and "1" (with the most significant digits to the left) followed by a cell containing the symbol ",", and let the *binary representation of a tuple* consist of the binary representations of its elements in order without gaps. Construct, for each n, a Turing machine BIN(n) that will transform the binary representation of an n-tuple into its unary representation.

4. Using the binary representation of tuples from Exercise 5.3, one can introduce the notion of a *binary Turing semidecidable set* in a natural manner. The existence of the machines BIN shows that every Diophantine set is binary Turing semidecidable. Show that the converse is true as well.

5. Let us say that a natural number valued function F defined on a set of n-tuples of natural numbers is *Turing computable* if there is a Turing machine M that, beginning with the canonical representation of the tuple $\langle a_1, \dots, a_n \rangle$ on its tape, will eventually halt if and only if $F(a_1, \dots, a_n)$ is defined, and for which, moreover, the tape will contain a representation of the singletong $\langle F(a_1, \dots, a_n) \rangle$ when it halts. Show that a function is Turing computable if and only if it is Diophantine.

6. Show that every set (semi)decidable by a Turing machine with a tape that is infinite in both directions is also (semi)decidable by a Turing machine with a tape possessing an end.

7. After the introduction of Turing machines, many other kinds of abstract computing devices were proposed. Certain of these can be used instead of Turing machines for establishing the relationship between "Diophantine" and "semidecidable." As an example, we can mention *register machines*. The memory of such a machine consists of finitely many *registers*, each of which can contain an arbitrarily large natural number. The behavior of a machine is specified by a program consisting of finitely many labeled *instructions*. The instructions may be of four types:

I. $\mathrm{L}j$: $\mathrm{R}i \leftarrow \mathrm{R}i + 1$;

II. $\mathrm{L}j$: $\mathrm{R}i \leftarrow \mathrm{R}i - 1$;

III. $\mathrm{L}j$: **if** $\mathrm{R}i = 0$ **then goto** $\mathrm{L}k$;

IV. $\mathrm{L}j$: **stop**.

An instruction of type I increases the contents of a register by 1; an instruction of type II decreases it by 1. An instruction of type III is a conditional branch breaking the natural order in which instructions are executed. Finally, an instruction of type IV stops the machine. It is assumed that programs are written in such a way that instructions of type II will execute only when the corresponding register does not contain 0. The notion of a *set semidecidable by a register machine* is introduced in a natural way: the elements of an n-tuple are placed into n input registers (while the other registers contain 0), the machine begins by executing the first instruction, and the n-tuple belongs to the set if and only if the machine eventually halts.

(a) Prove that the notions of Turing semidecidability and semidecidability on register machines are equivalent.
(b) Give a direct proof of the equivalence between "Diophantine" and "semidecidable by a register machine" by simulating register machines by Diophantine equations.

8. Show that every Diophantine set can be represented in the form

$$a \in \mathfrak{D} \iff \exists x_1 \dots x_m \left[\sum_{k=1}^{l} \lambda(k) x_1^{P_{k1}(a)} \dots x_m^{P_{km}(a)} = 0 \right],$$

where P_{kj} are polynomials with natural number coefficients and $\lambda(k) = \pm 1$.

Commentary

Alan Turing introduced the abstract computing devices that are now named for him in his classical paper [1936]. A very similar notion was also introduced by Emil L. Post [1936]. Since that time numerous modifications of the Turing-Post machine have been proposed. The version used in Section 5.1 was chosen as being particularly suitable for simulation by Diophantine equations.

Various authors have proposed other approaches for making the general notion of algorithm precise. (An exposition of the history of computability theory (in Russia the subject is often referred to as *the theory of algorithms*) can be found in Uspenskiĭ and Semënov [1987].) All of these approaches led to equivalent notions of decidable and semidecidable sets (the latter are more often called "recursively enumerable"). Alonzo Church [1936] was the first to realize that a single and, at first sight, very special definition can be adequate for the fundamental notion of computability. Church's Thesis has many equivalent formulations depending on the choice of a particular kind of abstract computing device, and the formulation from Section 5.7 is sometimes called *Turing's Thesis.* Kolmogorov and Uspenskiĭ [1958] attempted to give the most general definition of an abstract computing device satisfying the requirement that every step should be elementary, and proved its equivalence to more traditional models of computing devices, in particular to Turing machines.

The first papers aimed at proving the algorithmic unsolvability of Hilbert's Tenth Problem appeared in the early 1950's (see the Commentary to Chapter 1). Even at that time, there was no difficulty in proving that all Diophantine sets are semidecidable (for any standard definition of the latter notion). At this same time Martin Davis [1953] set forth the daring hypothesis that the converse is also true, i.e., that *every semidecidable set is Diophantine*, and thus that the number-theoretic notion of Diophantine set coincides with the notion of semidecidable set from computability theory.

Davis [1950, 1953] took the first step towards proving his hypothesis; namely, he showed that every semidecidable set of natural numbers can be represented in the form

$$a \in \mathfrak{M} \iff \exists z \, \forall y \leq z \, \exists x_1 \dots x_m \, [D(a, x_1, \dots, x_m, y, z) = 0]. \tag{1}$$

Such representations are said to be in *Davis normal form.*

Examples of sets that are semidecidable but not decidable have been known since the 1930's, and hence Davis's hypothesis implied a negative solution of Hilbert's Tenth Problem. However, efforts to prove the hypothesis encountered difficulties.

In this situation, it was natural to seek other approaches that, without proving Davis's hypothesis, would nevertheless provide a negative solution of Hilbert's Tenth Problem.

One possible way to weaken Davis's hypothesis is as follows: In the definition of a family of equations given in Section 1.4, it is required that the function D in (1.4.1) should be a polynomial in *all* the variables $a_1, \ldots, a_n, x_1, \ldots, x_m$. Instead of this, it would be sufficient to require polynomial dependence on $x_1, \ldots, x_m$ only. It would suffice that the dependence on $a_1, \ldots, a_n$ be via any algorithm that would provide, for any given values of $a_1, \ldots, a_n$, a Diophantine equation in $x_1, \ldots, x_m$. It is clear that even under such a weakened restriction on D, a positive solution of Hilbert's Tenth Problem would have implied the decidability of the set represented by (1.4.2).

In particular, one could consider the class of exponential Diophantine equations in which unknowns do not occur in the exponents. Such an approach was proposed by A. I. Mal'tsev [1968], but for reasons that are not entirely clear, although he allowed parameters to occur in the exponents, he did not permit parameters to occur polynomially like the unknowns. Thus, Davis's hypothesis does not formally give a solution to the problem posed by Mal'zev; however, a solution can be easily deduced from it (see Exercise 5.8).

Another weakening of Davis's hypothesis was proposed by S. Yu. Maslov [1967].

Davis's hypothesis was finally proved in 1970, but the chronological order in which the proof was obtained was different from the logical order followed in Chapters 1–5. One of the most crucial steps in the historical development consisted in obtaining an exponential Diophantine representation for an arbitrary semidecidable set. This result was first obtained by Davis and Putnam in a conditional form. Their result was announced in an abstract [1959a] and presented with full details in a report [1959b] that is not readily available.

In their proof Davis and Putnam used the still-unproved (1992) conjecture that there exist arbitrarily long arithmetical progressions consisting entirely of prime numbers. It was Julia Robinson who managed to eliminate the conjecture. The new result was announced in her short communication [1960] and a lecture [1962] and then published in the now-classic paper by Davis, Putnam, and Robinson [1961]. In that paper, an exponential Diophantine representation was obtained by eliminating the bounded universal quantifier from the Davis normal form (1). This technique will be presented in Chapter 6.

After it was established that every semidecidable set has an exponential Diophantine representation, it was sufficient to show that exponentiation was Diophantine in order to prove Davis's hypothesis. The history of that stage was presented in the Commentary to Chapter 2.

The proof that exponentiation was Diophantine was not only the final link in the proof of Davis's hypothesis, it also opened new routes for the construction of Diophantine representations of semidecidable sets. In particular, the Davis representation (1) ceased being a necessary step. Earlier it had been necessary because the method used for elimination of the bounded universal quantifier preceding the Diophantine equation in (1) led to an exponential Diophantine equation, and thus the process could not be iterated. Now that we can transform exponential Diophantine equations into Diophantine equations, we can begin with an arithmetic representation of a semidecidable set containing arbitrarily many universal quantifiers. A technique for obtaining arithmetical representations had been found long before by Kurt Gödel [1931].

On the other hand, the fact that exponentiation was Diophantine also made it possible to construct Diophantine representations without any use of bounded universal quantifiers, as was in fact done in Chapters 1–5. An approach of this kind was implemented for the first time by Matiyasevich [1976]. This construction was based on Turing machines, and the main difficulties were connected with the necessity of coding strings of symbols and sequences of their transformations by numbers. In Jones and Matiyasevich [1983, 1984, 1991] and Matiyasevich [1984], another construction is presented in which the initial representation of a semidecidable set is based on register machines (see Exercise 5.7). This model of abstract computing devices was proposed in Lambek [1961], Melzak [1961], Minsky [1961, 1967], and Shepherdson and Sturgis [1963]. Register machines are particularly suitable for constructing Diophantine representations. Like Turing machines, they have very primitive instructions and, in addition, they deal directly with numbers.

The method of Diophantine simulation of Turing machines described in Section 5.5 is different from the method given in Matiyasevich [1976] and is presented here for the first time. For yet another proof via Turing machines, see van Emde Boas [1983].

We conclude with a quotation from Hilbert's lecture [1900] that reveals his attitude to "negative" results:

> Occasionally it happens that we seek the solution under insufficient hypotheses or in an incorrect sense, and for this reason do not succeed. The problem then arises: to show the impossibility of the solution under the given hypotheses, or in the sense contemplated. Such proofs of im-

possibility were effected by the ancients, for instance when they showed that the ratio of the hypotenuse to the side of an isosceles triangle is irrational. In later mathematics, the question as to the impossibility of certain solutions plays a preëminent part, and we perceive in this way that old and difficult problems, such as the proof of the axiom of parallels, the squaring of circle, or the solution of equations of the fifth degree by radicals have finally found fully satisfactory and rigorous solutions, although in another sense than that originally intended. It is probably this important fact along with other philosophical reasons that gives rise to conviction (which every mathematician shares, but which no one has as yet supported by a proof) that every definite mathematical problem must necessary be susceptible of an exact settlement, either in the form of an actual answer to the question asked, or by the proof of the impossibility of its solution and therewith the necessary failure of all attempts.