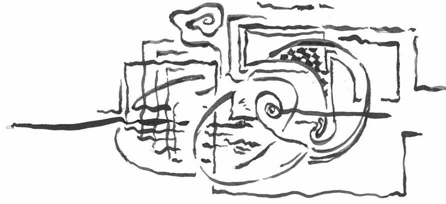

Discuss, commit errors, make mistakes,
but for God's sake think –
even if you should be wrong –
but think your own thoughts.

Gotthold Ephraim Lessing

Chapter 4

Limits of Computability or Why Do There Exist Tasks That Cannot be Solved Automatically by Computers

4.1 Aim

In Chapter 3 we discovered that there exist different infinite sizes. For instance, the number of real numbers is a larger infinity than the number of natural numbers. An infinite set is exactly as large as $\mathbb{N}$ if one can number the elements of A as the first one, the second one, the third one, etc. Here we aim to show that computing tasks exist that cannot be solved by any algorithm. The idea of our argument is simple. We show that the number of different tasks (computing problems) is a larger infinity than the number of all programs. Hence, there exist problems that cannot be algo-

rithmically solved, and so their solution cannot be automatically found by means of computers. But it is not satisfactory to prove the existence of algorithmically unsolvable problems. One could think that all algorithmically unsolvable problems are so artificial that none of them is really interesting for us. Therefore, we strive to show that there are concrete problems of serious interest in practice that cannot be algorithmically solved.

This chapter is the hardest one of this book, and so do not worry or be frustrated when you do not get a complete understanding of all details. Many of the graduate students at universities do not master this topic in detail. It is already valuable if one is able to understand and correctly interpret the computer science discoveries presented in what follows. To gain full understanding of the way in which these results were discovered usually requires multiple reading and discussions of the proof ideas. How many confrontations with this hard topic you perform is up to you.

It is important to know that one can successfully study the topics of all following chapters even in the case when one does not understand all the arguments of Chapter 4.

4.2 How Many Programs Exist?

How many programs do we have? The first simple answer is "Infinitely many." Clearly, for each program A, there is another program B that is longer by a row (by an instruction) than A. Hence, there are infinitely many program lengths and so infinitely many programs must exist. Our main question is whether the number of programs is equal to $|\mathbb{N}|$ or not. First we aim to show that the number of different programs is the same infinite size as the number of natural numbers. We show it by giving a numbering of programs.

Let us start by thinking about the number of texts that can be written by a computer or a typewriter. Each text can be viewed as a sequence of **symbols** of the keyboard used. We have to take into

account all uppercase and lowercase letters of the latin alphabet. Additionally, one is allowed to use symbols such as

?, !, ·, $, /, +, *, etc.

Moreover, every keyboard contains a key for the character blank. For instance, we use a blank to separate two words or two sentences. We often use the symbol ␣ to indicate the occurrence of the character blank. Since blank has its meaning in texts, we consider it as a symbol (letter). From this point of view, texts are not only words such as

"computer" or "mother"

and not only sentences such as

"Computer␣science␣is␣full␣of␣magic",

but also sequences of keyboard characters without any meaning such as

xyz*-+?!abe/ .

This means that we do not expect any meaning of a text. Semantics does not play any role in our definition of the notion of a text. A text is simply a sequence of symbols that does not need to have any interpretation. In computer science the set of symbols used is called an **alphabet**, and we speak about **texts over an alphabet** if all texts considered consist of symbols of this alphabet only.

Because blank is considered as a symbol, the content of any book may be viewed as a text. Hence, we can fix the following:

Every text is finite, but there is no upper bound on the length of a text. Therefore, there are infinitely many texts.

Let us observe the similarity to natural numbers. Each natural number has a finite decimal representation as a sequence of digits (symbols) $0, 1, 2, 3, 4, 5, 6, 7, 8,$ and 9. The length of the decimal representation grows with the number represented, and so there

is no bound on the representation length of natural numbers.[1]. Hence, natural numbers can be viewed as texts over the alphabet of decimal digits and so the number of texts over the alphabet $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$ is equal to $|\mathbb{N}|$. If one uses the binary alphabet $\{0, 1\}$ to represent natural numbers, one sees that the number of texts over the binary alphabet is also equal to $|\mathbb{N}|$

It appears that the size of the alphabet does not matter, and so that one can conjecture that

> *"The number of all texts over the characters of a keyboard is equal to* $|\mathbb{N}|$*."*

This is true, and we prove it by enumerating the texts. It is sufficient to show that one can order all texts in an infinite list. The ordering works in a similar way to creating a dictionary, but not exactly in the same way. Following the sorting rules of the dictionary, we have to take first the texts $a, aa, aaa, aaaa$, etc., and we will never order texts containing a symbol different from a, because there are infinitely many texts consisting of the letter a only. Therefore, we have to change the sorting approach a little bit. To order all texts in a list, we first apply the following rule:

> *Shorter texts are always before longer texts.*

This means that our infinite list of all texts starts with all texts of the length 1, then the texts of the length 2 follow, after that the texts of the length 3, etc. What still remains is to fix the order of the texts of the same length for any length. If one uses the letters of the Latin alphabet only, then one can do it in the same way as is used in dictionaries. This means to start with texts that begin with the letter a, etc. Since we also have a lot of special symbols on our keyboard such as ?, !, *, +, etc., we have to order the symbols of our keyboard alphabet first. Which order of symbols we use is our choice, and for our claim about the cardinality of the set of all texts over the keyboard alphabet it does not matter.

[1] This means that one cannot restrict the representation length by a concrete number. For instance, if one upperbounds the representation length by n, then one would have at most 10^n different representations available, and this is not enough to represent all infinitely many natural numbers.

Fig. 4.1 depicts a possible ordering of the symbols of the keyboard alphabet. Having an order of the alphabet symbols,

One sorts the texts of the same length in the same way as in dictionaries[2].

1	2	3	...	25	26	27	28	...	51	52	53	54	...	61	62
a	b	c	...	y	z	A	B	...	Y	Z	1	2	...	9	0

63	64	65	66	67	68	69	70	71	72	73	74	75	...	167
+	”	*	ç	&	!	.	:	,	;	?	$	£	...	␣

Fig. 4.1

This means that for the same text length, we start with texts beginning with the first symbol of our alphabet ordering. For instance, taking the order of symbols depicted in Fig. 4.1, the numbering of texts starts with

1 a
2 b
3 c
⋮
167 ␣
⋮

Then, the texts of length 5 are ordered as follows:

aaaaa
aaaab
aaaac
⋮
aaaa␣
aaaba
aaabb
aaabc

[2]Usually, we speak about the lexicographical order.

$\vdots$

Why did we take the time to show that the number of texts is equal to $|\mathbb{N}|$? Because

Each program is a text over the keyboard alphabet.

Programs are nothing more than special texts that are understandable for computers. Therefore, the number of all programs is not larger than the number of all texts over the keyboard alphabet, and so we can claim

The number of programs is equal to $|\mathbb{N}|$.

What we really have showed is that the number of programs is infinite and not larger than $|\mathbb{N}|$. The equality between $|\mathbb{N}|$, and the number of programs is the consequence of the fact that $|\mathbb{N}|$ is the smallest infinite size. But we did not prove this fact. Hence, if we want to convince the reader and provide a full argumentation of this fact, then we have to find a matching between $\mathbb{N}$ and programs. As we already know, any numbering provides a matching. And

One gets a numbering of all programs by erasing all texts that do not represent any program from our infinite list of all texts over the keyboard alphabet.

It is important to observe that the deletion of texts without an interpretation as programs can even be done automatically. One can write programs, called **compilers**, that get texts as inputs and decide whether a given text is a program in the programming language considered or not. It is worth noting that

A compiler can check the syntactical correctness of a text as a program but not the semantical correctness.

This means that a compiler checks whether a text is a correctly written sequence of computer instructions, i.e., whether the text is a program. A compiler does not verify whether the program is an algorithm, i.e., whether the program does something reasonable

or not, or whether the program can repeat a loop infinitely many times.

Hence, we are allowed to number all programs, and so to list all programs in a sequence

$$P_0, P_1, P_2, P_3, \ldots, P_i, \ldots$$

where $\boldsymbol{P_i}$ denotes the **i-th program**.

Why is it important for us that the number of programs and so the number of algorithms is not larger than $|\mathbb{N}|$? The answer is that the number of all possible computing tasks is larger than $|\mathbb{N}|$, and so there are more problems than algorithms. The immediate consequence is that there exist problems that cannot be solved by any algorithm (for which no solution method exists).

We have already showed in Chapter 3 that the number of problems is very large. For each real number c, one can consider the following computer task Problem(c).

Problem(c)

Input: a natural number n

Output: a number c up to n decimal digits after the decimal point

We say that an **algorithm A_c solves Problem(c)** or that **A_c generates c**, if, for any given $n \in \mathbb{N}$, A_c outputs all digits of c before the decimal point and the first n digits of c after the decimal point.

For instance,

- For $c = \frac{4}{3}$ and an input $n = 5$, the algorithm $A_{\frac{4}{3}}$ has to give the output 1.33333.
- For $\sqrt{2}$, the algorithm $A_{\sqrt{2}}$ has to generate the number 1.4142 for the input $n = 4$ and the number 1.414213 for the input $n = 6$.
- For $c = \pi$, the algorithm A_π has to provide the output 3.141592 for $n = 6$.

Exercise 4.1 What is the output of an algorithm $A_{\frac{17}{6}}$ generating 17/6 for the input $n = 12$? What are the outputs of an algorithm A_π that generates π for inputs $n = 2, n = 0, n = 7$, and $n = 9$?

Exercise 4.2 (challenge) Can you present a method for the generation of π up to an arbitrarily large number of digits after the decimal point?

In Chapter 3, we proved that the number of real numbers is larger than $|\mathbb{N}|$, i.e., that $|\mathbb{R}| > |\mathbb{N}|$. Since the number of algorithms is not larger than $|\mathbb{N}|$, the number of real numbers is larger than the number of algorithms. Therefore, we can conclude that

> *There exists a real number c such that Problem(c) is not algorithmically solvable.*

Thus, we proved that there are real numbers that cannot be generated by any algorithm. Do we understand exactly what this means? Let us try to build our intuition in order to get a better understanding of this result. The objects as natural numbers, rational numbers, texts, programs, recipes, and algorithms have something in common.

> *All these objects have a finite representation.*

But this is not true for real numbers. If one can represent a real number in a finite way, then one can view this representation as a text. Since the number of different texts is smaller than the number of real numbers, there must exist a real number without any finite representation.

What does it exactly mean? To have a constructive description of a real number e means that one is able to generate e completely digit by digit. Also, if the number e has an infinite decimal representation, one can use the description to unambiguously estimate the digit on any position of its decimal representation. In this sense, the finite description of e is complete. In other words, such a finite description of e provides an algorithm for generating e. For instance, $\sqrt{2}$ is a finite description of the irrational number

$e = \sqrt{2}$, and we can compute this number with an arbitrarily high precision by an algorithm.[3] Therefore, we are allowed to say:

> *Real numbers having a finite representation are exactly the numbers that can be algorithmically generated, and there exist real numbers that do not possess a finite representation and so are not computable (algorithmically generable).*

Exercise 4.3 What do you mean? Are there more real numbers with finite representations than real numbers without any finite representation, or vice versa? Justify your answer!

We see that there are tasks that cannot be solved by algorithms. But we are not satisfied with this knowledge. Who is interested in asking for an algorithm generating a number e that does not have any finite representation? How can one formulate such a task in a finite way? Moreover, when only tasks of this kind are not algorithmically solvable, then we are happy and forget about this "artificial" theory and dedicate our time to solving problems of practical relevance. Hence, you may see the reason why we do not stop our investigation here and are not contented with our achievements. We have to continue our study in order to discover, whether there are interesting computing tasks with a finite description that cannot be automatically solved by means of computers.

4.3 YES or NO, That Is the Question, or Another Application of Diagonalization

Probably the simplest problems considered in computer science are decision problems. A decision problem is to recognize whether a given object has a special property we are searching for or not. For instance, one gets a digital picture and has to decide whether a chair is in the picture. One can also ask whether a person is in the picture, or even whether a specific person (for instance, Albert Einstein) is in the picture. The answer has to be unambiguous

[3] For instance, by the algorithm of Heron.

"YES" or "NO". Other answers are not allowed and we force that the answer is always correct.

Here, we consider a simple kind of decision problems. Let M be an arbitrary subset of $\mathbb{N}$, i.e., let M be a set that contains some natural numbers. We specify the **decision problem** $(\mathbb{N}, \boldsymbol{M})$as follows.

Input: a natural number n from $\mathbb{N}$

Output:

"YES" if n belongs to M
"NO" if n does not belong to M

For instance, one can take PRIME as M, where

$$\text{PRIME} = \{2, 3, 5, 7, 11, 13, 17, 19, \ldots\}$$

is the infinite set of all primes. Then, ($\mathbb{N}$, PRIME) is the problem to decide whether a given natural number n is prime or not. The problem $(\mathbb{N}, \mathbb{N}_{even})$ is to decide whether a given nonnegative integer is even or not.

For each subset M of $\mathbb{N}$ we say that an **algorithm *A* recognizes *M*** or that an **algorithm *A* solves the decision problem** $(\mathbb{N}, \boldsymbol{M})$, if, for any input n, A computes

(i) the answer "YES" if n belongs to M, and

(ii) the answer "NO" if n does not belong to M ($n \notin M$).

Sometimes one uses the digit "1" instead of "YES" and the digit "0" instead of "NO". If A answers "YES" for an input n, then we say that **algorithm *A* accepts the number *n***. If A outputs "NO" for an input n, then we say that **algorithm *A* rejects the number *n***. Thus, an algorithm recognizing PRIME accepts each prime and rejects each composite number.

If there exists an algorithm solving a decision problem $(\mathbb{N}, M)$, then we say that the problem $(\mathbb{N}, M)$ is **algorithmically solvable** or that

the problem $(\mathbb{N}, M)$ is decidable.

Clearly, the decision problem $(\mathbb{N}, \mathbb{N}_{even})$ is decidable. It is sufficient to verify whether a given natural number is even or odd. The problem ($\mathbb{N}$, PRIME) is also decidable because we know how to check whether a natural number is a prime or not, and it is not too complicated to describe such a method as an algorithm.

Exercise 4.4 The naive method for primality testing is to divide the given number n by all numbers between 2 and $n-1$. If none of these $n-2$ numbers divides n, then n is a prime. To test primality in this way means to perform a lot of work. For the number 1000002 one has to execute 1000000 divisibility tests. Can you propose another method that can verify primality by performing an essentially smaller number of divisibility tests?

Exercise 4.5 (challenge) Write a program in the programming language TRANSPARENT of Chapter 2 that solves the problem ($\mathbb{N}$, QUAD) where

$$\text{QUAD} = \{1, 4, 9, 16, 25, \ldots\}$$

is the set of all squares i^2.

First, we aim to show the existence of decision problems that are not algorithmically solvable. Such decision problems are called

undecidable or **algorithmically unsolvable**.

We already recognized that we can list all programs as $P_0, P_1, P_2, \ldots$ and later we will see that one can do it by an algorithm. To list all algorithms by an algorithm is not so easy. Therefore, we begin our effort by proving a stronger result than we really need. We show that there are decision problems that cannot be solved by any program. What does "solved by a program" mean? What is the difference between algorithmic solvability and solvability by a program?

Remember that each algorithm can be written as a program, but it does not hold that each program is an algorithm. A program can perform a pointless work. A program can perform infinite work for some inputs without producing any result. But an algorithm must always finish its work in a *finite time* and produce a *correct result*.

Let M be a subset of $\mathbb{N}$. We say that a **program *P* accepts the set *P***, if, for any given natural number n,

(i) P outputs "YES", if n belongs to M, and

(ii) P outputs "NO" or works **infinitely long** if n does not belong to M.

For a program P, $\boldsymbol{M(P)}$ denotes the set M accepted by P. In this way, P can be viewed as a finite representation of the potentially infinite set $M(P)$.

Immediately, we see the difference between the recognition of M by an algorithm and the acceptance of M by a program. For inputs from M both the algorithm and the program are required to work correctly and provide the right answer "YES" in a finite time (see requirement (i)). In contrast to an algorithm, for numbers not belonging to M, a program is allowed to work infinitely long and so never produce any answer. In this sense algorithms are special programs that never run infinite computations. Therefore, it is sufficient to show that there is no program accepting a set M, and the direct consequence is that there does not exist any algorithm that recognizes M (i.e., solves the decision problem $(\mathbb{N}, M)$).

To construct such a "hard" subset M of $\mathbb{N}$, we use the diagonalization method from Chapter 3 again. For this purpose, we need the following infinite representation of subsets of natural numbers (Fig. 4.2).

	0	1	2	3	4	...	i	$i+1$	...
M	0	1	0	0	1	...	1	0	...

Fig. 4.2

M is represented as an infinite sequence of bits. The sequence starts with the position 0 and has 1 at the i-th position if and only if the number i is in M. If i is not in M, then the bit 0 is on the i-th position of the sequence. The set M in Fig. 4.2 contains the numbers $1, 4$, and i. The numbers $0, 2, 3$, and $i+1$ are not in M. The binary representation of $\mathbb{N}_{even}$ looks as follows

101010101010101010 ...

The representation of PRIM starts with the following bit sequence:

0011010100010100 ...

Exercise 4.6 Write the first 17 bits of the binary representation of QUAD.

Now, we again build a two-dimensional table that is infinite in both directions. The columns are given by the infinite sequence of all numbers:

$0, 1, 2, 3, 4, 5, \ldots, i, \ldots.$

The rows are given by the infinite sequence of all programs:

P_0, P_1, P_2, P_3, ..., P_i, ...

that reads an input number only once and their only possible outputs are "YES" and "NO". One can recognize such programs by determining whether they contain only one instruction of reading and whether the only output instructions are writing the text "YES" or the text "NO". Each such program P_i unambiguously defines a set $M(P_i)$ of all natural numbers that are accepted[4] by P_i. Those numbers, for which P_i outputs "NO" or works infinitely long do not belong to $M(P_i)$.

The rows of our table are the binary representations of sets $M(P_i)$. The k-th row (see Fig. 4.3) contains the binary representation of the set $M(P_k)$ that is accepted by the program P_k. The intersection of the i-th row and the j-th column contains "1" if P_i accepts the number j (if P_i halts on the input j with the output "YES"). The symbol "0" lies in the intersection of the i-th row and the j-th column, if P_i outputs "NO" or works infinitely long for the input j. Hence

The infinite table contains in its rows the representation of ***all*** *subsets of* $\mathbb{N}$ *that can be accepted by a program.*

Next we aim to show that there is at least one subset of $\mathbb{N}$ missing in the table, i.e., that there is a subset of $\mathbb{N}$, whose binary

[4] P_i finishes the work on them with printing "YES".

	0	1	2	3	4	5	6	...	i	...	j	...
$M(P_0)$	0	1	1	0	0	1	0		1		0	
$M(P_1)$	0	1	0	0	0	1	1		0		0	
$M(P_2)$	1	1	1	0	0	1	0		1		1	
$M(P_3)$	1	0	1	0	1	0	1		1		0	
$M(P_4)$	0	0	0	1	1	0	1		0		1	
$M(P_5)$	1	1	1	1	1	1	1		1		1	
$M(P_6)$	1	0	1	0	0	0	1		0		1	
$\vdots$												...
$M(P_i)$	0	1	1	0	0	1	0		1			
$\vdots$												...
$M(P_j)$	1	0	1	0	1	1	1				0	
$\vdots$												$\vdots$

Fig. 4.3

	0	1	2	3	4	5	6	...	i	...	j	...
DIAG	1	0	0	1	0	0	0		0		1	...

Fig. 4.4

representation differs from each row of the table (Fig. 4.3). We show it by constructing a sequence of bits, called DIAG, that does not occur in any row of the table. The construction of the bit sequence DIAG and the corresponding set M(DIAG) is done by the diagonalization method.

First, see the binary value a_{00} in the intersection of the 0-th row and the 0-th column. If $a_{00} = 0$ (Fig. 4.3), i.e., if 0 does not belong to $M(P_0)$, then we set the 0-th position d_0 of DIAG to 1. If $a_{00} = 1$ (i.e., if 0 is in $M(P_0)$), then we set $d_0 = 0$ (i.e., we do not take 0 to M(DIAG)). After this first step of the construction of DIAG we fixed only the value of the first position of DIAG, and due to this we are sure that DIAG differs from the 0-th row of the table (i.e., from $M(P_0)$) at least with respect to the membership of the 0-th element.

Analogously, we continue in the second construction step. We consider the second diagonal square, where the first row intersects the first column. We aim to choose the first position d_1 of DIAG is such

a way that DIAG differs from the binary representation of $M(P_1)$ at least in the value of this position. Therefore, if $a_{11} = 1$ (i.e., if 1 is $M(P_1)$), we set d_1 to 0 (i.e., we do not take 1 into $M(\text{DIAG})$). If $a_{11} = 0$ (i.e., if 1 is not in $M(P_1)$), then we set $d_1 = 1$ (i.e., we take 1 into $M(\text{DIAG})$).

If $\bar{a}_{ij}$ represents the opposite value to a_{ij} for any bit in the intersection of the i-th row and the j-th column (the opposite value to 1 is the value $\bar{1} = 0$ and $\bar{0} = 1$ is the opposite value to 0), then, after two construction steps, we reached the situation as depicted in Fig. 4.5.

	0	1	2	3	4	$\cdots$	i	$i+1$	$\cdots$
DIAG	$\bar{a}_{00}$	$\bar{a}_{11}$	?	?	?	$\cdots$	?	?	$\cdots$

Fig. 4.5

The first two elements of DIAG are $\bar{a}_{00}$ and $\bar{a}_{11}$, and so DIAG differs from both $M(P_0)$ and $M(P_1)$. The remaining positions of DIAG are still not determined, and we aim to fix them in such a way that DIAG will differ from each row of the table in Fig. 4.3.

In general, we guarantee a difference between DIAG, and the i-th row of the table in Fig. 4.3 as follows. Remember that $\bar{a}_{ii}$ is the bit of the square in the intersection of the i-th row and the i-th column and that d_i denotes the i-th bit of DIAG. If $\bar{a}_{ii} = 1$ (i.e., if i belongs to $M(P_i)$), then we set $d_i = 0$ (i.e., we do not take i into $M(\text{DIAG})$). If $\bar{a}_{ii} = 0$ (i.e., if i is not in $M(P_i)$), then we set $d_i = 1$ (i.e., we take i into $M(\text{DIAG})$). Hence, $M(\text{DIAG})$ differs from $M(P_i)$.

By this approach DIAG is constructed in such a way that it does not occur in any row of the table. For the concrete, hypothetical table in Fig. 4.3, Fig. 4.4 shows the corresponding representation of DIAG. In general, one can outline the representation of DIAG as done in Fig. 4.6.

In this way we obtain that

	0	1	2	3	4	$\cdots$	i	$\cdots$
DIAG	$\bar{a}_{00}$	$\bar{a}_{11}$	$\bar{a}_{22}$	$\bar{a}_{33}$	$\bar{a}_{44}$	$\cdots$	$\bar{a}_{ii}$	$\cdots$

Fig. 4.6

> M(DIAG) *is not accepted by any program, and therefore the decision problem* ($\mathbb{N}$, M(DIAG)) *cannot be solved by any algorithm.*

One can specify M(DIAG) also in the following short way:

$$\begin{aligned} M(\text{DIAG}) &= \{n \in \mathbb{N} \mid n \text{ is not in } M(P_n)\} \\ &= \text{the set of all natural numbers } n, \\ & \text{ such that } n \text{ is not in } M(P_n). \end{aligned}$$

Exercise 4.7 Assume the intersection of the first 10 rows and the first 10 columns in the table of all programs contains values as written in Fig. 4.7. Estimate the first 10 positions of DIAG.

	0	1	2	3	4	5	6	7	8	9	$\cdots$
$M(P_0)$	1	1	1	0	0	1	0	1	0	1	
$M(P_1)$	0	0	0	0	0	0	0	0	0	0	
$M(P_2)$	0	1	1	0	1	0	1	1	0	0	
$M(P_3)$	1	1	1	0	1	1	0	0	0	0	
$M(P_4)$	1	1	1	1	1	1	1	0	1	0	
$M(P_5)$	0	0	1	0	0	1	0	1	1	0	
$M(P_6)$	1	0	0	0	1	0	1	0	0	0	
$M(P_7)$	1	1	1	1	1	1	1	1	1	1	
$M(P_8)$	0	0	1	1	0	0	1	1	0	0	
$M(P_9)$	1	0	1	0	1	0	1	0	1	0	
$M(P_{10})$	0	0	1	0	0	0	1	1	0	1	
$\vdots$											$\ddots$

Fig. 4.7

Exercise 4.8 (challenge) Consider

M(2-DIAG) = the set of all even numbers $2i$, such that $2i$ is not in $M(P_i)$.

Is the decision problem ($\mathbb{N}$, M(2-DIAG)) algorithmically solvable? Carefully explain your argument! Draw diagrams that would similarly to Fig. 4.3 and Fig. 4.4 show the construction of 2-DIAG.

Exercise 4.9 (challenge) Can you use the solution to Exercise 4.8 in order to define two other subsets of ℕ that are not algorithmically solvable? How many algorithmically unsolvable problems can be derived by diagonalization?

Exercise 4.10 (challenge) Consider

> $M(\mathrm{DIAG}_2)$ as the set of all even natural numbers $2i$ such that $2i$ is not in $L(P_{2i})$.

Can you say something about the algorithmical solvability of $(\mathbb{N}, M(\mathrm{DIAG}_2))$?

Now, we know that the decision problem $(\mathbb{N}, M(\mathrm{DIAG}))$ is not algorithmically solvable. But we are not satisfied with this result. The problem looks to be described in a finite way by our construction, though it is represented by an infinite sequence of bits. But our construction does not provide any algorithm for generating DIAG because, as we will see later, though the table in Fig. 4.3 really exists, it cannot be generated by an algorithm. Moreover, the decision problem $(\mathbb{N}, M(\mathrm{DIAG}))$ does not correspond to any natural task arising in practice.

4.4 Reduction Method or How a Successful Method for Solving Problem Can Be Used to Get Negative Results

We already know how to use the diagonalization method in order to describe algorithmically unsolvable problems. This provides a good starting position for us. In this section, we learn how to "efficiently" spread the proofs of algorithmic unsolvability to further problems. The main idea is to introduce the relation "easier or equally hard" or "not harder than" with respect to the algorithmic solvability.

Let U_1 and U_2 be two problems. We say that

> **U_1 is easier than or equally hard as U_2**

or that

U_1 **is not harder than** U_2

with respect to algorithmic solvability and write

$$U_1 \leq_{Alg} U_2,$$

if the algorithmic solvability of U_2 implies (guarantees) the algorithmic solvability of U_1.

What does it exactly mean? If

$$U_1 \leq_{Alg} U_2$$

holds, then the following situations are possible:

- *U_1 and U_2 are both algorithmically solvable.*
- *U_1 is algorithmically solvable, and U_2 is not algorithmically solvable.*
- *Both U_1 and U_2 are algorithmically solvable.*

The only situation that the validity of the relation $U_1 \leq_{Alg} U_2$ excludes is the following one[5]:

- *U_2 is algorithmically solvable and U_2 is not algorithmically solvable.*

Assume that the following sequence of relations

$$U_1 \leq_{Alg} U_2 \leq_{Alg} U_3 \leq_{Alg} \ldots \leq_{Alg} U_k$$

between the k problems $U_1, U_2, \ldots, U_k$ was proved. Moreover, assume that one can show by the diagonalization method that

U_1 is not algorithmically solvable.

What can be concluded from these facts? Since U_1 is the easiest problem among all problems of the sequence, all other problems $U_2, U_3, \ldots, U_k$ are at least as hard as U_1 with respect to algorithmic solvability, and so one can conclude that

[5]Remember the definition of the notion of implication in Chapter 1. The truthfulness of the implication "The solvability of U_2 implies the solvability of U_1" excludes exactly this one situation.

the problems $U_2, U_3, \ldots, U_k$ *are not algorithmically solvable.*

This is the way we want to walk around in order to prove the algorithmic unsolvability of further problems. Due to diagonalization we already have the initial problem U_1 for this approach. This problem is the decision problem $(\mathbb{N}, M(\text{DIAG}))$. The only question is, how to prove the validity of the relation $U_1 \leq_{Alg} U_2$ between two problems?

For this purpose, we apply the reduction method, which was developed in mathematics in order to solve new problems by clever application of known methods for solving other problems. We give two examples showing a simple application of the reduction method.

Example 4.1 Assume one has a method for solving quadratic equations in the so-called p, q-form

$$x^2 + px + q = 0,$$

i.e., quadratic equations with the coefficient 1 before the term x^2. The method for solving such quadratic equations is given by the so-called p-q-formula:

$$x_1 = -\frac{p}{2} + \sqrt{\left(\frac{p}{2}\right)^2 - q}$$

$$x_2 = -\frac{p}{2} - \sqrt{\left(\frac{p}{2}\right)^2 - q}.$$

If $\left(\frac{p}{2}\right)^2 - q < 0$ holds, then the quadratic equation in the p, q-form does not have any real solution.

Now, we are searching for a method for solving arbitrary quadratic equations

$$ax^2 + bx + c = 0 \ .$$

Instead of deriving a new formula[6] for this purpose, we reduce the problem of solving general quadratic equations to the problem of solving quadratic equations in the p, q-form.

[6] We presented such a formula in Chapter 2 and wrote a program computing solutions by this formula there.

We know that the solutions of an arbitrary equation do not change, if one multiplies both sides of the equation by the same number different form 0. Hence, we are allowed to multiply both sides of the general quadratic equation by $1/a$.

$$\begin{aligned} ax^2 + bx + c &= 0 \qquad\qquad \mid \cdot \frac{1}{a} \\ a \cdot \frac{1}{a} \cdot x^2 + b \cdot \frac{1}{a} x + c \cdot \frac{1}{a} &= 0 \cdot \frac{1}{a} \\ x^2 + \frac{b}{a} x + \frac{c}{a} &= 0. \end{aligned}$$

In this way we got a quadratic equation in the p, q-form and this equation can be solved by the method presented above. An algorithmic representation of this reduction is outlined in Fig. 4.8.

Part A is an algorithm that corresponds to the reduction. Here, one computes the coefficients p and q of the equivalent quadratic equation in the p, q-form. This is all one has to do in this algorithmic reduction. The coefficients p and q are the inputs for algorithm B for solving the quadratic equations in the form $x^2+px+q = 0$. B solves this equation for the given p and q. The output of B (either the solutions x_1 and x_2 or the answer "there is no solution") can be taken over as the output of the whole algorithm C for solving general quadratic equations. □

Exercise 4.11 Assume we have an algorithm B for solving linear equations in the form

$$ax + b = 0.$$

Design an algorithm for solving linear equations of the form

$$cx + d = nx + m$$

by reduction. The symbols c, d, n, and m remain for concrete numbers, and x is the unknown. Outline the reduction in a similar way to what we used in Fig. 4.8.

The reduction form in Example 4.1 is called **1-1-reduction** (one to one reduction). It is the simplest possible reduction, in which the input of a problem U_1 (a general quadratic equation) is directly

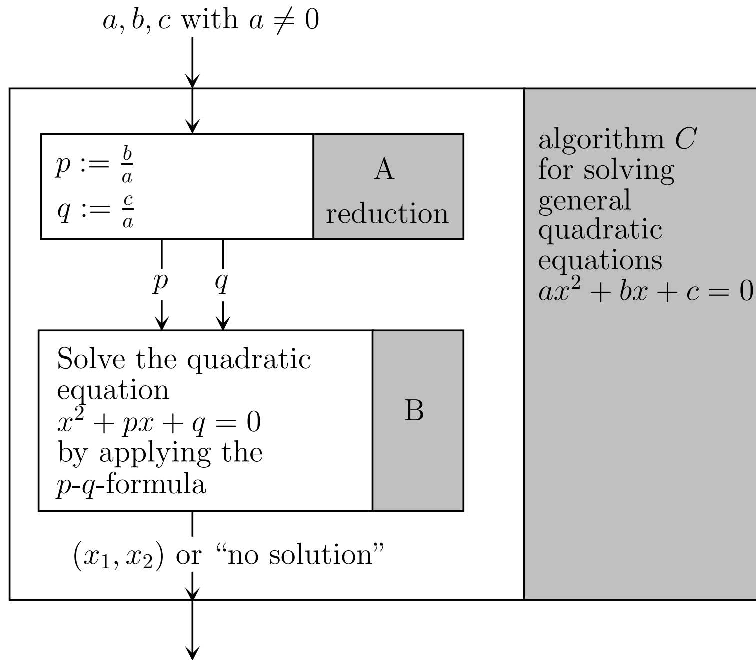

Fig. 4.8

transformed to an input of a problem U_2 (a quadratic equation in the p, q-form), and the result of the computation of the algorithm on the input of U_2 is taken one to one over as the result for the given input instance of U_1. This means that

$$U_1 \leq_{Alg} U_2 \tag{4.1}$$

holds. In other words, solving U_1 in an algorithmic way is not harder than solving U_2, because each algorithm B solving U_2 can be "modified" by reduction (Fig. 4.8 to an algorithm C that solves U_1.

Moreover, in the case of quadratic equations, we observe that U_2 (solving quadratic equations in p, q-form) is a special case of U_1 (solving general quadratic equations). Hence, each algorithm for U_1 is automatically an algorithm for U_2, and so

$$U_2 \leq_{Alg} U_1. \tag{4.2}$$

Following Equations (4.1) and (4.2) we may claim that U_1 and U_2 are equally hard. This means that either both problems are algorithmically solvable or both are not algorithmically solvable. Clearly, we know in this special case of solving quadratic equations that the first possibility is true.

In general, the reductions need not be so simple as the one presented. To prove

$$U_1 \leq_{Alg} U_2$$

one can need to apply the algorithm B solving U_2 several times for different inputs and additionally to work on the outputs of B in order to compute the correct results for U_1. To illustrate such a more general reduction, we present the following example.

Example 4.2 We assume that everybody is familiar with the Pythagorean theorem, which says that in any right-angled triangle (Fig. 4.9) the equality

$$c^2 = a^2 + b^2$$

holds. In other words

> *The square of the length of the longest side of any right-angled triangle is equal to the sum of the squares of the lengths of the two shorter sides.*

In this way, we obtained an algorithm $B_\triangle$ that for given lengths of two sides of a right-angled triangle computes the length of the third side. For instance, for known a and b, one can compute c by the formula

$$c = \sqrt{a^2 + b^2}.$$

If a and c are known, one can compute b as

$$b = \sqrt{c^2 - a^2}.$$

Let us denote by $U_\triangle$ the problem of computing the missing side length of a right-angled triangle.

Assume now a new task U_{Area}. For a given equilateral triangle (Fig. 4.10) with the side lengths m, one has to compute the area

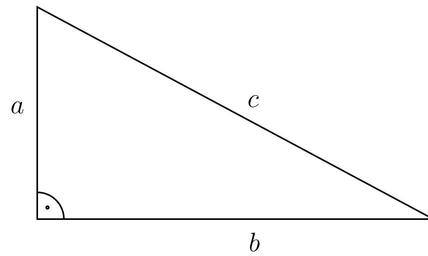

Fig. 4.9

of the triangle. We see (Fig. 4.10) that the area of such a triangle is

$$\frac{m}{2} \cdot h$$

where h is the height of the triangle (Fig. 4.10).

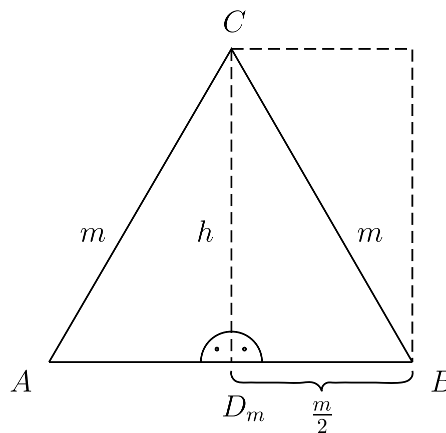

Fig. 4.10

We are able to show

$$U_{Area} \leq_{Alg} U_{\triangle}$$

by reducing solving U_{Area} to solving $U_{\triangle}$. How to do it is depicted in Fig. 4.11.

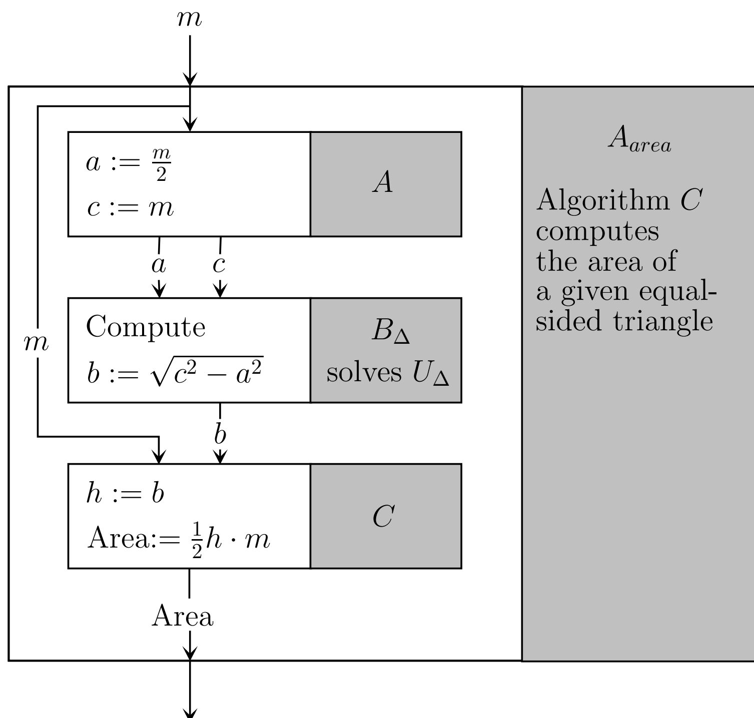

Fig. 4.11

We designed an algorithm A_{Area} for solving U_{Area} under the assumption that one has an algorithm $B_{\triangle}$ for solving $U_{\triangle}$ (for computing the missing size length in a right-angled triangle). We see in Fig. 4.11 that one needs the height h of the triangle in order to compute the area of the triangle. The height h is the length of the side CD of the right-angled triangle DBC. We also see that the length of the side DB is equal to m. Algorithm A in Fig. 4.11 uses these facts to compute the values a and c and send them as inputs for $B_{\triangle}$. The algorithm $B_{\triangle}$ computes the missing length b

of $\triangle DBC$, which is the height h of $\triangle ABC$. Finally, the algorithm C computes the area of $\triangle ABC$ from the values of m and b. □

Exercise 4.12 Consider the problem U_{Pyr} of computing the height of a pyramid with quadratic base of the size $m \times m$ and the lengths of the edges also equal to m. Solve this task by showing $U_{Pyr} \leq_{Alg} U_{\triangle}$. Show the reduction as we have done in Fig. 4.11.

[Hint: Consider the possibility of applying the algorithm $B_{\triangle}$ twice for different inputs.]

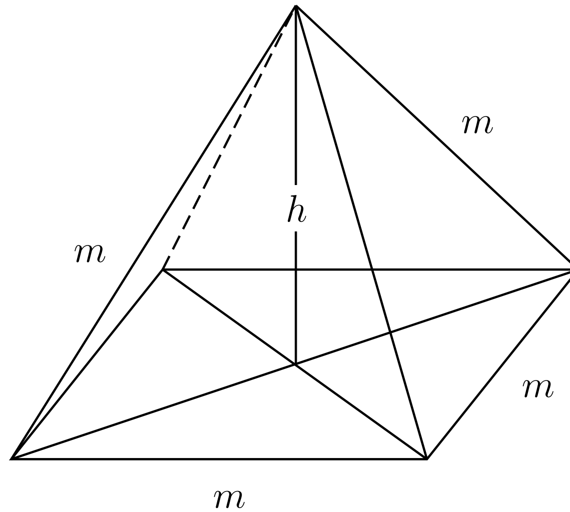

Fig. 4.12

Exercise 4.13 (challenge) Let U_{2lin} denote the problem of solving a system of two linear equations

$$\begin{aligned} a_{11}x + a_{12}y &= b_1 \\ a_{21}x + a_{22}y &= b_2 \end{aligned}$$

with two unknown x and y. Let U_{3lin} denote the problem of solving a system of three linear equations

$$\begin{aligned} a_{11}x + a_{12}y + a_{13}z &= b_1 \\ a_{21}x + a_{22}y + a_{23}z &= b_2 \\ a_{31}x + a_{32}y + a_{33}z &= b_3 \end{aligned}$$

with three unknown x, y, and z. Show that $U_{3lin} \leq_{Alg} U_{2lin}$.

We saw how reductions can be used to develop new methods for solving problems by using known methods for solving other problems. In this way, one uses reduction to extend the positive results about algorithmic solvability.

But our aim is not to use reduction as a means for designing new algorithms (i.e., for broadcasting positive messages about algorithmic solvability). We aim to use reduction as an instrument for spreading negative messages about algorithmic unsolvability. How can one reverse a method for designing positive results to a method for proving negative results? We outlined this idea already at the beginning of this chapter. If one is able to prove

$$U_1 \leq_{Alg} U_2$$

by a reduction and one knows that U_1 is not algorithmically solvable, then one can conclude that U_2 is not algorithmically solvable, too.

There is a simple difference between proving

$$U_1 \leq_{Alg} U_2$$

for spreading algorithmic solvability and for showing algorithmic unsolvability. For deriving a positive result, one already has an algorithm for U_2 and one tries to use it in order to develop an algorithm for U_1. To broaden negative results about unsolvability, we do not have any algorithm for U_2. We only *assume that there exists an algorithm solving* U_2. Under this assumption we build an algorithm that solves U_1. This means that we have to work with the hypothetical existence of an algorithm A_2 for U_2 and use it for designing of an algorithm for U_1.

Applying reduction to show the algorithmic solvability of a problem corresponds to a direct proof (direct argumentation), which was introduced in Chapter 1. Using reduction for proving the nonexistence of any algorithm for solving the problem considered corresponds to an indirect proof, whose schema was presented in

Section 1.2. To get a transparent connection to something known, we give an example from geometry first, and then we switch to algorithmics.

Example 4.3 One knows that it is impossible to partition an arbitrary given angle into three equal-sided angles by means of a ruler and a pair of compasses. In other words, there is no method as a sequence of simple construction steps executable by means of a ruler and a pair of compasses that would guarantee a successful geometric partitioning of any angle into three equal-sided angles. The prove of this negative result is far from being obvious and so we prefer here to trust mathematicians and believe it.

On the other hand, one may know from one's school time that there are simple methods for geometric doubling or halving of each angle.

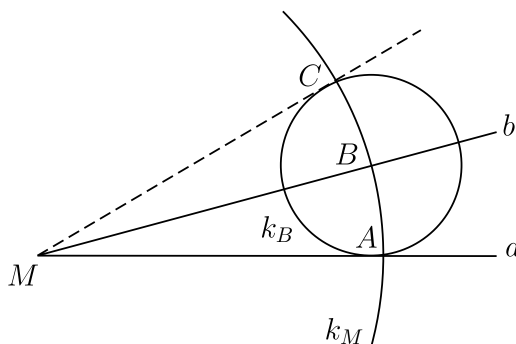

Fig. 4.13: Doubling an angle

For instance, Fig. 4.13 outlines how to double the angle $\angle ab$ between two lines a and b. An RC-algorithm (a ruler-compass-algorithm) for doubling an angle can work as follows:

1. Take an arbitrary positive distance r with the pair of compasses and draw a circle k_M with the center M (the intersection of a and b) and the radius r.

2. Denote by A the intersection of the circle k_m and the line a, and denote by B the intersection of k_M and the line b.
3. Take the distance $\overline{AB}$ between A and B in the pair of compasses and draw a circle k_M with the center B and the radius $\overline{AB}$.
4. Denote by C the intersection of the circles k_m and k_B that is different from A.
5. Connect the points M and C with a line.

We see that the angle $\angle AMC$ between the straight line a and the straight line connecting M and C is twice as large as the original angle $\angle ab = \angle AMB$.

Now, our aim is to show that there do not exist any CL-algorithms that can partition an arbitrary angle into 6 equal-sided parts. This does not look easier than to prove the nonexistence of any CL-algorithm for partitioning angles into three equal-sided angles. Our advantage is that we are not required to use this hard way of creating a nonexistence proof. We know already that one cannot partition angles into three equal-sided parts by means of a ruler and a pair of compasses. We use this fact to reach our aim efficiently.

How to proceed? We assume the opposite of what we want to prove and show under this assumption that one can partition angles into three equal-sided parts by a CL-algorithm. But this contradicts the already known fact about impossibility of partitioning angles into three equal-sided parts. More precisely, we assume that there is a CL-algorithm A_6 for partitioning angles into 6 equal-sided angles and use A_6 to design a CL-algorithm A_3 that partitions each given angle into three equal-sided angles. Since A_3 does not exist (as we already know), A_6 cannot exist either.

We describe the reduction of the problem of partitioning angles into three equal-sided angles to the problem of partitioning angles into 6 equal-sided parts as follows (Fig. 4.14). We assume that one has a CL-algorithm A_6 for partitioning angles into 6 equal-sided parts. We design a CL-algorithm A_3 that has an angle W as its

input. At the beginning, A_3 applies A_6 in order to partition W into 6 equal-sided angles $w_1, w_2, w_3, w_4, w_5, w_6$ (Fig. 4.15). Then, A_3 joins the angles w_1 and w_2 into the angle w_{12}. Similarly, A_3 joins w_3 and w_4 into an angle w_{34}, and w_5 and w_6 are joined into w_{56} (Fig. 4.15). We see that partitioning W into three angles w_{12}, w_{34}, and w_{56} corresponds to the partitioning of W.

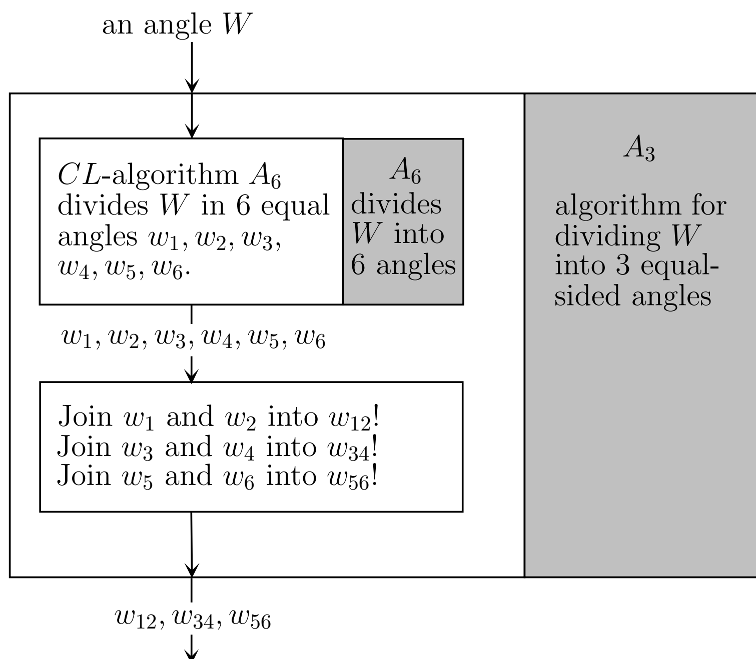

Fig. 4.14

Using the language of the indirect argumentation (indirect proof), the reduction in Fig. 4.14 corresponds to the following implication:

> *If there is a CL-algorithm for partitioning angles into 6 equal-sided parts, then there is a CL-algorithm for partitioning angles into 3 equal-sided angles.*

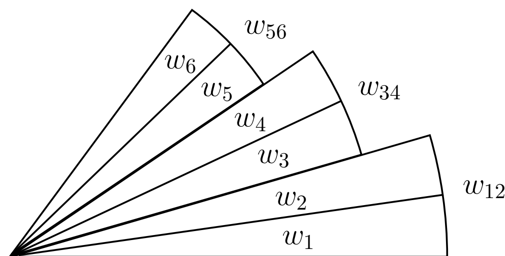

Fig. 4.15

Following the definition of the notion "implication", the truthfulness of this implication proved above (Fig. 4.14) excludes the second situation from the 4 possible situations listed in Fig. 4.16.

situation	6 parts	3 parts
1	possible	possible
2	possible	impossible
3	impossible	possible
4	impossible	impossible

Fig. 4.16

Taking into account that partitioning angles into three equal-sided parts is impossible, situations 1 and 3 are excluded, too. Hence, the only remaining possible situation is situation 4. Situation 4 contains the impossibility of partitioning angles into 6 equal-sided parts, and this completes our indirect argumentation for the nonexistence of a CL-algorithm partitioning angles into 6 equal-sided angles. □

Exercise 4.14 The problem of partitioning of an angle into three parts also has the following simplified representation. The task is to construct, for any given angle W, such an angle V by means of a linear and a pair of compasses that the size of V is one third of the size of W. One can prove that this simplification does not change anything on the CL-unsolvability of this problem. Use this fact to create in a similar way as in Fig. 4.14 reductions (proofs) showing the nonexistence of CL-algorithms for constructing angles of the size of

(i) one sixth

(ii) one ninth

of any given angle.

Now, we return from the world of CL-algorithms into the world of general algorithms. Our problem DIAG plays here a similar role as the problem of partitioning angles into three equal-sided parts does for CL-algorithms. Starting from algorithmic unsolvability of a problem, we want to conclude algorithmic unsolvability of another problem.

The reduction schema for $U_1 \leq_{Alg} U_2$ is outlined in Fig. 4.17.

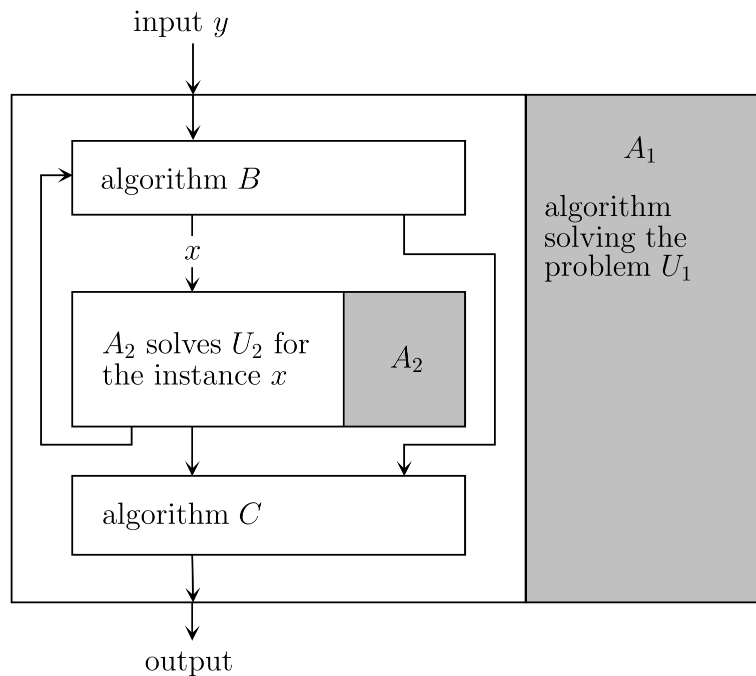

Fig. 4.17

The algorithm A_1 solving the problem U_1 is created as follows. First, the input instance y of U_1 is proceeded by an algorithm B that transforms y to a problem instance x of the problem U_2.

Following our assumption about the existence of an algorithm A_2 solving U_2, the algorithm A_2 computes the correct solution for the input x. As one may see in Fig. 4.17, A_2 can be used repeatedly several times. Finally, an algorithm C proceeds all outputs of A_2 and computes the final result for the problem instance y of U_1. We call attention to the important fact that A_2, B, and C are algorithms and therefore they provide their outputs in a final time. The number of requests on A_2 must be finite and so the loop containing B and A_2 can run only finitely many times. Therefore, we can conclude that A_1 is an algorithm for U_1 because it provides the correct output in a finite time for any input instance of U_1.

Next, we introduce two new decision problems that are of interest for program developers.

UNIV (the **universal problem**)
Input: a program P and an input $i \in \mathbb{N}$ for P
Output: YES, if P accepts the input i, i.e., if i is in $M(P)$.
NO, if P does not accept i
(i.e., if $i \notin M(P)$), which means that P either halts and rejects i or P works infinitely long on the input i.

HALT (the **halting problem**)
Input: a program P and a natural number i
Output: YES, if P halts on the input i
(i.e., if P finishes its work on i in a finite time).
NO, if P does not halt on i
(i.e., if P has an infinite computation on i repeating a loop infinitely many times).

The halting problem is one of the fundamental tasks in testing software products. We already know that only those programs can be considered to be algorithms that never get into an infinite computation. Hence, an important part of checking the correct functionality of programs is to verify whether they always (for every input) guarantee an output in a finite time. The halting problem HALT is a simple version of such testing. We are only asking, whether a given program P halts on a concrete input i. (The real

question is whether a given program halts on every possible input.) Later, we will see that even this simplified testing problem is algorithmically unsolvable.

The universal problem UNIV is directly related to verifying the correct functionality of a program solving a decision problem. We test whether P provides the correct result YES or NO on an input i. Now, somebody can propose the following simple way to solve UNIV. Simulate the work of P on i and look whether P outputs YES or NO. Certainly, one can do it if one has a guarantee that P halts on i (i.e., that P is an algorithm). But we do not have this guarantee. If P executes an infinite computation on i, we would simulate the work of P on i infinitely long and would never get the answer to our question, whether P accepts i or not. An algorithm for the universal problem is not allowed to work infinitely long on any input P and i, and so it is not allowed to execute an infinite simulation.

Following these considerations, we get the impression that the halting problem and the universal problem are strongly interlocked. Really, we show that these problems are equally hard.

First we show that

$$\text{UNIV} \leq_{Alg} \text{HALT}$$

i.e., that

UNIV is not harder than HALT with respect to algorithmical solvability.

What do we have to show? We have to show that the existence of an algorithm for HALT assures the existence of an algorithm that solves UNIV. Following our schema of indirect proofs, we assume that one has an algorithm A_{HALT} solving the halting problem. Using this assumption, we build an algorithm B that solves UNIV (Fig. 4.18).

The algorithm B works on any input (P, i) as follows:

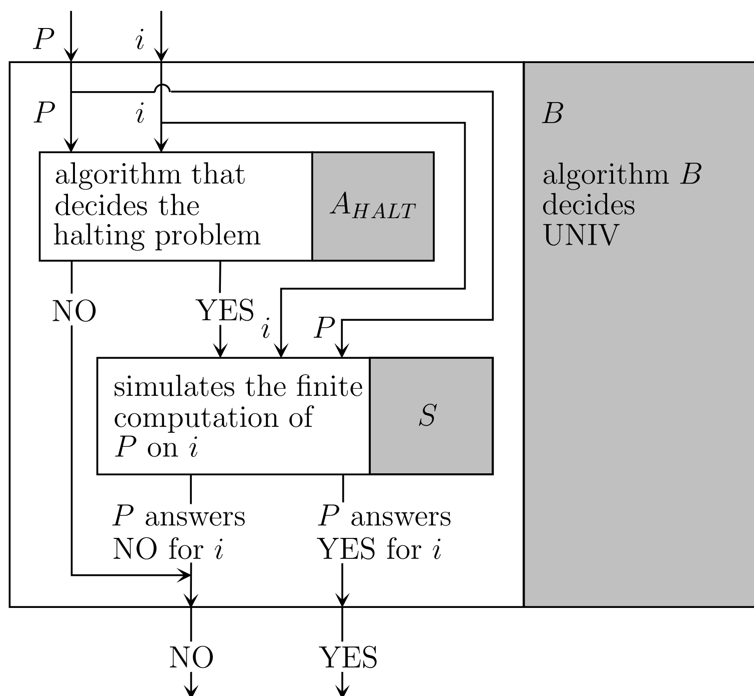

Fig. 4.18

1. B transfers its input (P, i) without any change to the algorithm A_{HALT}.
2. The algorithm A_{HALT} decides (in finite time) whether P halts on i or not. A_{HALT} answers YES if P halts on i. If P does not hold on i, A_{HALT} answers NO.
3. If A_{HALT} outputs NO, B is sure about the fact that P does not accept i (because P works on i infinitely long) and provides the final output NO saying that "i is not in $M(P)$".
4. If A_{HALT} outputs YES, then B simulates by a subprogram S (Fig. 4.18) the finite computation of P on i. Executing this finite simulation B sees whether P accepts i or not and outputs the corresponding claim.

Following the construction of B, we immediately see that B takes the right decision with respect to the membership of i in $M(P)$. We still have to verify whether B always finishes its work in a finite time. Under the assumption that A_{HALT} is an algorithm for HALT, we know that A_{HALT} provides outputs in a finite time, and so B cannot run for an infinitely long time in its part A_{HALT}. The simulation program S starts to work only if one has the guarantee that the computation of P on i is finite. Therefore, the simulation runs always in a finite time, and hence B cannot get into an infinite repetition of a loop in the part S. Summarizing, B always halts, and so B is an algorithm that solves the universal problem.

We showed above that UNIV is easier than or equally hard as HALT. Our aim is to show that these problems are equally hard. Hence, we have to show that the opposite relation

$$\text{HALT} \leq_{Alg} \text{UNIV}$$

holds, too. This means, we have to show that the algorithmic solvability of UNIV implies the algorithmic solvability of HALT. Let A_{UNIV} be an algorithm that decides UNIV. Under this assumption we design an algorithm D that solves HALT. For any input (P, i), the algorithm D works as follows (Fig. 4.19).

1. D gives P to a subprogram C that transforms P into P' in the following way. C finds all rows of P containing the instruction *output ("NO")* and exchanges "NO" for "YES". Hence, the constructed program never outputs "NO" and the following is true:

   Each finite computation of P finishes with the output YES and P' accepts exactly those natural numbers i on which P halts.

2. D gives P' and i as inputs to A_{UNIV} (Fig. 4.19). A_{UNIV} decides whether i is in $M(P')$ or not.
3. D takes over the answer YES or NO of A_{UNIV} as the final answer for its input (P, i).

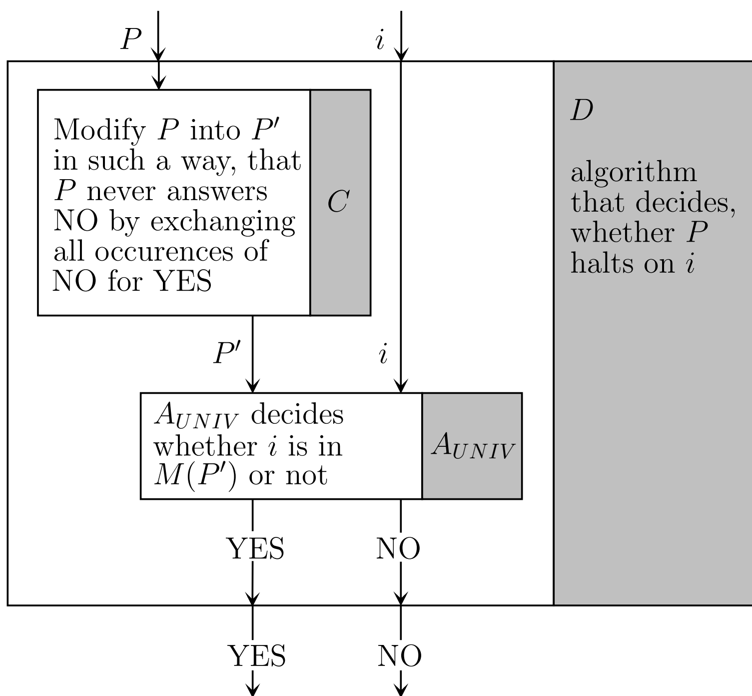

Fig. 4.19

Exercise 4.15 Provide a detailed explanation why D is an algorithm that solves the halting problem.

Exercise 4.16 (challenge) The reduction for $A_{\text{UNIV}} \leq_{Alg} A_{\text{HALT}}$ in Fig. 4.18 and the reduction $A_{\text{HALT}} \leq_{Alg} A_{\text{UNIV}}$ (Fig. 4.19) look different. Usually, one prefers the kind of reduction presented in Fig. 4.19 because it corresponds to the typical reduction in mathematics. Here, one transforms the input instance (P, i) of HALT to an input instance (P', i) of UNIV in such a way that the solution for the instance (P, i) of HALT is exactly the same as the solution for (P', i) of UNIV. Due to this, one can take the answer of A_{UNIV} for (P', i) as the final output for the input (P, i) of HALT. The schema of this reduction is the simple schema presented in Fig. 4.8 and Fig. 4.19. Find such a simple reduction for the proof of $A_{\text{UNIV}} \leq_{Alg} A_{\text{HALT}}$. This means, you have to algorithmically transform the instance (P, i) of UNIV into such an instance (P', i) that the output of A_{HALT} for (P', i) (i.e., the reduction for (P', i) of the halting problem) corresponds to the solution for the instance (P, i) of UNIV.

Above, we showed that the universal problem and the halting problem are equally hard with respect to algorithmic solvability. This means that either both problems are algorithmically solvable or both are algorithmically unsolvable. As we already mentioned, we aim to prove their unsolvability. To do that, it is sufficient to show that one of them is not easier than $(\mathbb{N}, M(\text{DIAG}))$. Here, we prove

$$(\mathbb{N}, M(\text{DIAG})) \ \leq_{Alg} \ \text{UNIV}.$$

We assume the existence of an algorithm A_{UNIV} for UNIV and use A_{UNIV} to create an algorithm A_{DIAG} that decides $(\mathbb{N}, M(\text{DIAG}))$. For any natural number i, the algorithm A_{DIAG} has to compute the answer YES if the i-th program P_i does not accept the number i and the answer NO if P_i accepts i.

For any input i, our hypothetical algorithm A_{DIAG} works on i as follows (Fig. 4.20):

1. A_{DIAG} gives i to a subprogram A_{gen} that generates the i-th program P_i.
2. A_{DIAG} gives i and P_i as inputs to the hypothetical algorithm A_{UNIV}. A_{UNIV} decides whether P_i accepts i (providing the answer YES) or not (providing the answer NO).
3. A_{DIAG} exchanges the answers of A_{UNIV}. If A_{UNIV} outputted YES (i.e., that i is in $M(P_i)$), then i does not belong to $M(\text{DIAG})$ and A_{UNIV} computes NO. If A_{UNIV} computed NO (i.e., i is not in $M(P_i)$), then i is in $M(\text{DIAG})$ and A_{UNIV} correctly answers YES (Fig. 4.20).

Following the description of the work of A_{DIAG} on i, we see that A_{DIAG} provides the correct answers under the assumption that A_{UNIV} and A_{gen} work correctly. The fact that A_{UNIV} is an algorithm solving UNIV is our assumption. The only open question is whether one can build an algorithm A_{gen} that, for each given natural number i, constructs the i-th program P_i in a finite time. A_{gen} can work as follows (Fig. 4.21). It generates texts over the keyboard alphabet one after the other in the order described at the beginning of this chapter. For each text generated, A_{gen} uses a compiler

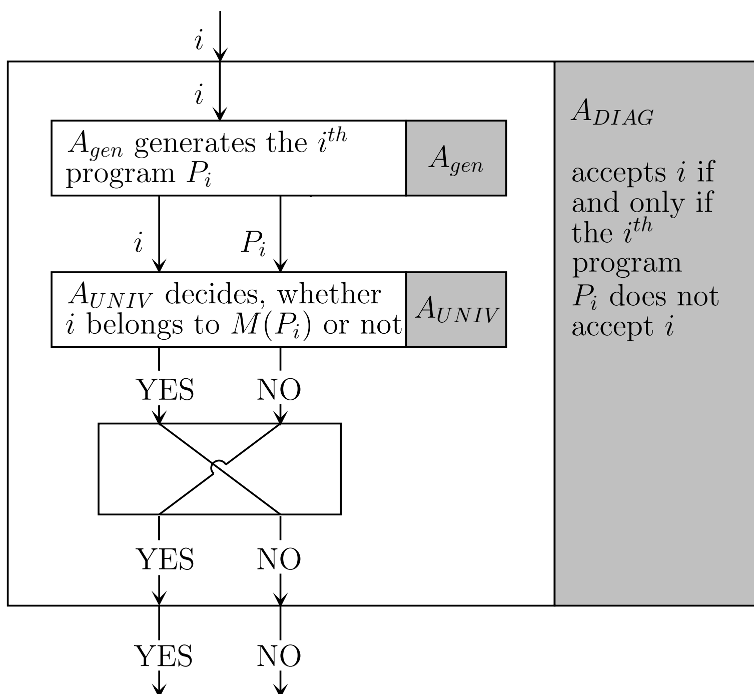

Fig. 4.20

in order to check whether the text is a program representation or not. Moreover, A_{gen} counts the number of positive compiler answers. After A_{gen} gets i positive answers, it knows that the last text generated is the i-th program P_i. The structure (flowchart) of the algorithm A_{gen} is outlined in Fig. 4.21.

Exercise 4.17 Show that $(\mathbb{N}, M(\text{DIAG})) \leq_{Alg}$ HALT is true by a reduction from $(\mathbb{N}, M(\text{DIAG}))$ to HALT.

Exercise 4.18 Let $M(\overline{\text{DIAG}})$ be the set of all natural numbers i, such that P_i accepts i. Hence, $M(\overline{\text{DIAG}})$ contains exactly those natural numbers that do not belong to $M(\text{DIAG})$. Show by a reduction that

$$(\mathbb{N}, M(\text{DIAG})) \leq_{Alg} (\mathbb{N}, M(\overline{\text{DIAG}})) \text{ and } (\mathbb{N}, M(\overline{\text{DIAG}})) \leq_{Alg} (\mathbb{N}, M(\text{DIAG})).$$

We showed that the decision problem $(\mathbb{N}, M(\text{DIAG}))$, the universal problem UNIV, and the halting problem HALT are not algorithmi-

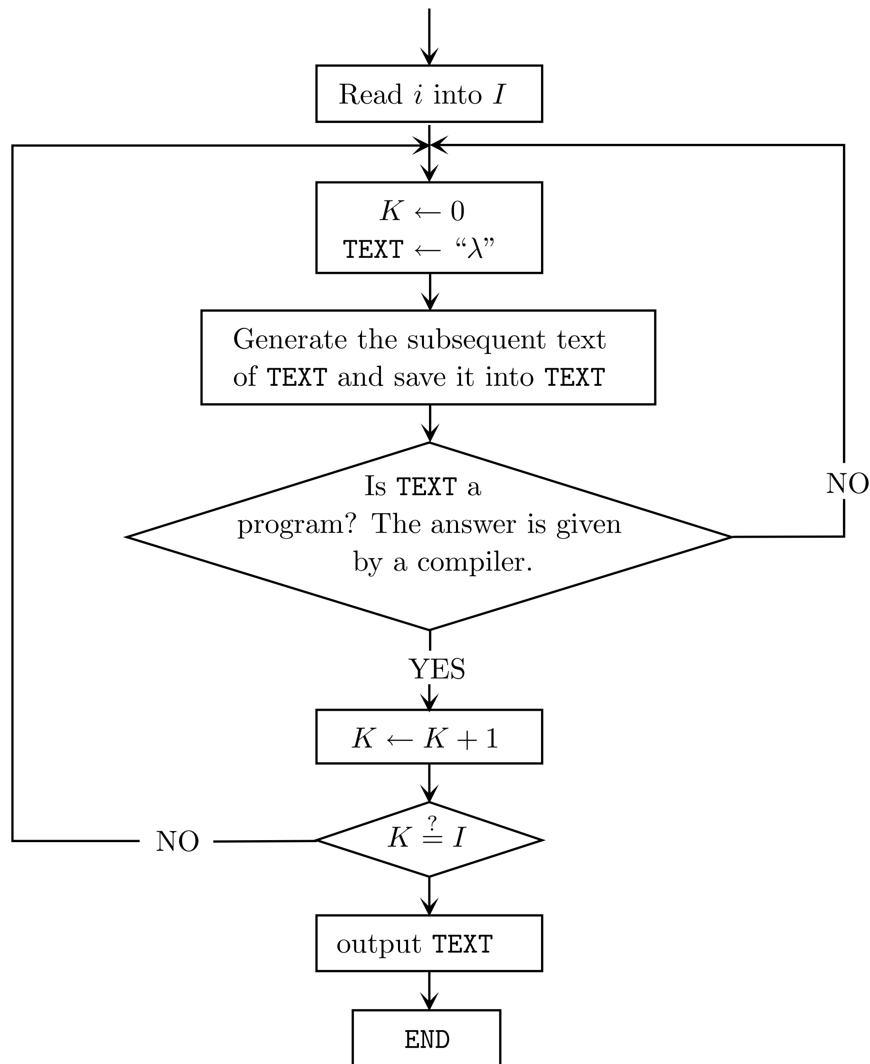

Fig. 4.21

cally solvable. The problems UNIV and HALT are very important for testing of programs and so are of large interest in software engineering. Unfortunately, the reality is that all important tasks related to program testing are algorithmically unsolvable. How

bad is this can be imagined by considering the unsolvability of the following simple task.

Let f_0 be a function from natural numbers to natural numbers that is equal to 0 for every input $i \in \mathbb{N}$. Such functions are called constant functions because the result 0 is completely independent of the input (of the argument). The following program

```
0 Output ← ''0''
1 End,
```

computes $f_0(i)$ for each i, and we see that it does not need to read the input value i because i does not influence the output. Despite of the simplicity of this task, there is no algorithm that is able to decide for a given program P whether P computes f_0 or not (i.e., whether P is a correct program for computing f_0). To understand it properly, note that input instances of this problem may also be very long complex programs doing additionally a lot of pointless work. The question is only whether at the end the correct result "0" is computed or not.

Exercise 4.19 (challenge) Let $\mathcal{M}_0$ be the set of all programs with $M(P) = \emptyset$. In other words, $\mathcal{M}_0$ contains all programs that, for each input, either outputs NO or executes an infinite computation. Prove that it is not algorithmically decidable, whether a given program P belongs to $\mathcal{M}_0$ (whether P does not accept any input) or not.

In Chapter 4, we learned something very important. *All syntactic questions and problems such as "Is a given text a program representation" are algorithmically decidable.* We are even able to construct the i-th program P_i for any given i. *All the semantic questions and problems about the meaning and the correctness of programs are not algorithmically solvable.*

4.5 Summary of the Most Important Discoveries

We destroyed the dream from the early 20th century about automating everything. We discovered *the existence of tasks that*

cannot be automatically solved by means of computers controlled by algorithms. This claim is true independently of current and future computer technologies.

Among the algorithmically unsolvable problems, one can find many tasks of interest such as

- Is a program correct? Does it fit the aim, which it was developed for?
- Does a program avoid infinite computations (endless repetitions of a loop)?

In computer science, there are several huge research communities focusing on testing programs.[7] Unfortunately, even such *simple questions as whether a given program computes a constant function are not algorithmically solvable.* The scientists in this area are satisfied if they are able to develop algorithms for testing some kinds of partial correctness of programs. For instance, they try to develop automatic testing of programs in some very special representation or at least to find some typical errors without any guarantee that all errors have been found.

For computing tasks related to program testing, one distinguishes between syntactic and semantic problems. **Syntactic** tasks are usually related to the correct representation of a program in the formalism of a given programming language, and these problem settings are algorithmically solvable. **Semantic** questions are related to the meaning of programs. For instance:

- What does a given program compute? Which problem does the program solve?
- Does the program developed solve the given problem?
- Does a given program halt on a given input or does a program always (for all inputs) halt?
- Does a given program accept a given input?

[7] This witnesses the importance of program testing in practice.

All nontrivial semantic tasks about programs are not algorithmically solvable.

To discover the above presented facts, we learned two proof techniques that can also be viewed as research methods, too. The first technique was the diagonalization method that we already applied in the study of infinities in Chapter 3. Using it, we were able to show that the number of problems is a larger infinity than the number of programs. Consequently, there exist algorithmically unsolvable problems. The first algorithmically unsolvable problem we discovered was the decision problem $(\mathbb{N}, M(\text{DIAG}))$. To extend its algorithmic unsolvability to other problems of practical interest, we used the reduction method. This method was used for a long time in mathematics in order to transform algorithmic solvability from problem to problem (i.e., for broadcasting positive messages about automatic solvability). The idea of the reduction method is to say that

P_1 is not harder than P_2, $P_1 \leq_{Alg} P_2$,

if, assuming the existence of an algorithm for P_2, one can build an algorithm solving P_1. If this is true, we say that P_1 *is reducible to P_2.*

The positive interpretation of the fact $P_1 \leq_{Alg} P_2$ is that algorithmic solvability of P_2 implies the algorithmic solvability of P_1. The negative meaning of $P_1 \leq_{Alg} P_2$ we used is that the algorithmic unsolvability of P_2 follows from the algorithmic unsolvability of P_1. We applied the reduction method in order to derive the algorithmic unsolvability of the halting problem and of the universal problem from the algorithmic unsolvability of the diagonal problem.

Solutions to Some Exercises

Exercise 4.3 The number of real numbers with a finite representation is exactly $|\mathbb{N}|$. We know that $2 \cdot |\mathbb{N}| = |\mathbb{N}|$ and even that $|\mathbb{N}| \cdot |\mathbb{N}| = |\mathbb{N}|$. Since $|\mathbb{R}| > |\mathbb{N}|$, we obtain

$$|\mathbb{R}| > |\mathbb{N}| \cdot |\mathbb{N}| .$$

Therefore, we see that the number of real numbers with a finite representation is an infinitely small fraction of the set of all real numbers.

Exercise 4.6 One can see the first 17 positions of the binary representation of QUAD in the following table:

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
QUAD	1	1	0	0	1	0	0	0	0	1	0	0	0	0	0	0	1	0

You can easily extend this table for arbitrarily many positions.

Exercise 4.7 The first 10 positions of DIAG for the hypothetical table in Fig. 4.7 are

$$\text{DIAG} = 0101000011.$$

Exercise 4.9 We aim to show that the set

> M(2-DIAG) = the set of all even positive integers $2i$, such that $2i$ is not in $M(P_i)$

is not decidable. The idea is again based on diagonalization as presented in Fig. 4.3. We create 2-DIAG in such a way that M(2-DIAG) differs from each row of the table. The only difference to constructing DIAG is that 2-DIAG differs from the i-th row of the table in the position $2i$ (instead of i in the case of DIAG). The following table in Fig. 4.22 provides a transparent explanation of the idea presented above.

	0	1	2	3	4	5	6	7	8	9	10	11	12	···
M(P_0)	[0]	0	1	1	0	1	1	0	1	1	1	1	0	
M(P_1)	1	0	[1]	1	0	0	0	0	1	0	1	1	0	
M(P_2)	1	1	1	1	[1]	1	1	1	0	0	0	1	0	
M(P_3)	0	1	0	1	0	0	[0]	0	1	1	1	0	0	
M(P_4)	1	0	1	0	1	0	1	0	[1]	0	1	0	1	
M(P_5)	0	1	0	1	1	0	0	1	0	1	[0]	1	1	
M(P_6)	0	0	0	0	0	0	0	0	0	0	0	0	[0]	···
⋮													⋮	⋱

Fig. 4.22

The positions in boxes correspond to the intersections of the i-th row with the $2i$-th column. The set 2-DIAG differs from the rows of the table at least in the values written on these positions. Hence, the first 13 positions of 2-DIAG with respect to the hypothetical table in Fig. 4.22 are

$$\text{2-DIAG} = \underline{1}0\underline{0}0\underline{0}0\underline{1}0\underline{0}0\underline{1}0\underline{1}\ldots$$

We see that we took 0 for each odd position of 2-DIAG, and we note that the values of these positions do not have any influence on the undecidability of 2-DIAG. The

underlined binary values starting with the 0-th position are related to the values in the boxes in table in Fig. 2.2. In this way, the underlined bit 1 on the 0-th position of 2-DIAG guaranteed that 2-DIAG is different from the 0-th row of the table. The underlined bit 0 on the second position guarantees that 2-DIAG does not lay in the second row, etc. The bit 1 on the 12-th position of 2-DIAG assures that 2-DIAG differs from the 6-th row.

Exercise 4.17 We aim to show that having an algorithm A_{HALT} solving HALT, one can algorithmically recognize the set $M(\text{DIAG})$. We start in a similar way as in Fig. 4.20, where the reduction $(|\mathbb{N}|, M(\text{DIAG})) \leq_{Alg}$ UNIV was outlined. Our task is to decide for a given i whether i belongs to $M(\text{DIAG})$ (i.e., whether P_i does not accept i) or not. We again use A_{gen} in order to generate the i-th program P_i and ask the algorithm A_{HALT} whether P_i halts on i or not (Fig. 4.23).

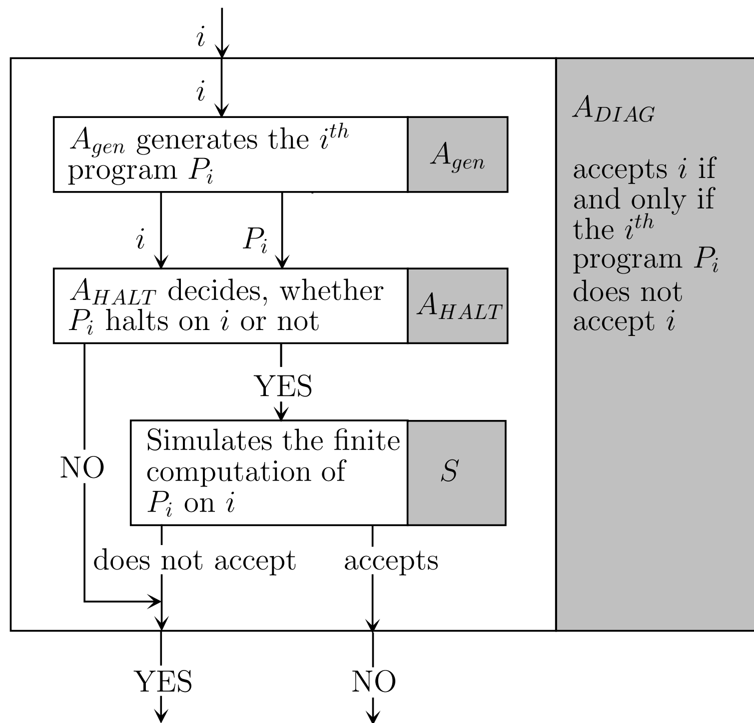

Fig. 4.23

If P_i does not halt on i, then i does not belong to $M(P_i)$ and one knows with certainty that $i \notin M(P_i)$ (P_i does not accept i), and so that the correct answer is YES ("$i \in M(\text{DIAG})$"). If P_i halts on i, then we continue similarly as in the

reduction presented in Fig. 4.18. We simulate the work of P_i on i in a finite time and convert the result of the simulation. If P_i does not accept i, then we accept i. If P_i accepts i, then we do not accept the number i (Fig. 4.23).