Algorithmic Logic
for
Software Construction and Analysis

Grażyna Mirkowska Andrzej Salwicki

24 May 2009

Contents

Preface

This book is about logic in programming. At present it is a domain calling the interest of many researchers, and covered by many monographs and papers. Therefore we had to choose among many topics. Logic is today a tool used in many fields of computer science: in software engineering, in data bases, in expert systems, in robotics, in the theory of parallel computations and almost everywhere. Does exist one logic which enables formulation and solving questions that come from such diverse fields? Obviously not. One can find in use: classical logic (logic of first-order formulas), intuitionistic logic, modal logics, logics based on resolution principle, temporal logic, various logics of programs (here we find: algorithmic logics, dynamic logics, Floyd-Hoare logic), multimodal logics, infinitary logics, logics of higher orders etc. One can foresee the appearance and development of new logical calculi. For example, in order to describe the phenomena of integrated circuits (abbr. VLSI) a completely new, two-dimensional logical formalism is needed, which will differ from the present, linear formalisms. Classical predicate logic and the resolution method lie at the foundations of PROLOG programming language. Various non-standard logics e.g. logic of defaults, logic of beliefs, have to find applications in the future systems of artificial inteligence. It is widely believed that various dialects of temporal logic are the appropriate tools for the analysis of concurrent programs. Number of papers devoted to Logic In Computer Science is so big that it justified the organization of annual conferences LICS.

Our practice as programmers motivates us to think of the tools needed in software engineering. In our opinion production of software requires a logical apparatus proper for the process of specification, analysis and implementation of software systems. It turns out, that the classical logic is not quite appropriate for the description of phenomena which appear in the software production. For it is from its very nature static and therefore it does not reflects the dynamics of algorithmic processes. Hence, we decided to present yet another logical calculus, which, in our opinion can help the programmers in their everyday work. We offer the algorithmic logic of the

simplest programs and its applications in the specification of data structures and algorithms, in analysis of semantical properties of programs and in the definition of semantics of programming languages.

We do not assume that the reader has any specific knowledge of topics discussed in the book. We assume only the acquaintance with the basic facts of logic on the level of high school. A minimal practice in programming and algorithmics is desirable. Well, we would like to believe, that the present book can introduce the reader into the problems of programming, but the acquisition of personal experience in design and running programs is indispensable.

We do not develop neither the classical propositional calculus nor the classical predicate calculus. Both the systems are included into algorithmic logic. Moreover, the literature devoted to classical logics is rich and every library has many valuable positions. We would like to recommend: the compact monograph of Lyndon [33], the book by A.Grzegorczyk [21] and the book of H.Rasiowa [44], which is especially recommended for those who came to computer science without any mathematical background.

The book is addressed to the students of computer science, to programmers having the ambition to improve their working environment, to the designers of big software systems and new programming languages. We can not promise that after reading the book you will write better programs (more effective, more ingenious etc.). Certainly, the reader will have better understanding how intellectual his work is and which tools can be used in need. Let us hope that the knowledge of problems discussed in the book will lead to improvement of the way of documenting your work. This would be your substantial success, measurable in economic terms.

Beside algorithmic logic we present certain modal logic and its semantics patterned on Kripke models. Modal logic may inspire the reader to his own research. It found many diverse applications. In the book we present one such application in construction of a mathematical model of concurrent computations. The algorithmic logic itself can be viewed as a variant of modal logic.

The material contained in the book may be helpful during various courses in computer science departments. It may serve as a textbook for the lectures on theory of programming. It may be an auxiliary reading to the courses on: introduction to computer science, logic with elements of set theory for computer science, methods of programming (!), algorithms and data structures, semantics of programming languages. The book was used by us as a textbook for the lectures on theory of programming. During a one semester course one can present almost all material here contained. It is suggested to make more exercises in proving properties of programs. It was remarked that students

find a lot of satisfaction of their own successful attempts to axiomatize data structures. For those who will became interested in problems presented in this book we would recommend the more complete monographs on logic of programs and logic in general [3, 21, 23, 25, 29, 33, 40, 45].

Chapter 1

Introduction

1.1 Logic? What for?

The design and analysis of software systems must be based on inference methods which are independent of experiment, of the person creating an algorithm. The methods should, in a reliable way, enable to derive valid properties of algorithms.

Certainly, many programmers sense the necessity to base their work on solid theoretic foundations. Yet, before we shall try to justify our belief that logical methods are indispensable for the majority of programmers, we would like to remark, that we know personally a few excellent programmers and we know they do not use any formal methods. Inconsistency, you may say, the authors encourage us to read a completely useless book. However, the problem is not so simple, as it would like to seem at the first glance. These experts in programming, intuitively apply the formal laws that rule the behavior of programs, their computations. One can make, in this moment, an analogy with the mathematics. Many outstanding mathematicians lead fruitful research and obtain fascinating results, not knowing about mathematical logic nor the subtle results of metamathematical considerations. Still, every professional mathematician applies logical methods of inference, because they are tightly connected with all his work. Let us remark, that the mathematics develops itself since thousands of years and that it was mathematical research which stimulated the research in logic. The computer science had not this time. It appeared less then fifty years ago and it develops dynamically, its applications are even quicker, more dynamic. This situation creates new demands. Many researchers with serious achievements in the field of programming (to mention just two names C.A.R.Hoare and E.Dijkstra) remarked the necessity of creation of logical tools proper for computer science.

Let us remark that the amount and the importance of software grows steadily. On the market the transfer of software is bigger than this of hardware. One can gain a lot producing quickly programs of quality. On the other hand the price of software is high. It is caused by diverse factors. We did not reached yet the ability to produce the software in an industrial way. Many believes that an appropriate theory is needed. At the foundations of technology one can find various physical and mathematical theories. For the development of an industrial technology of software production a theory of programming is necessary. The price which we pay for using programs with errors in them (i.e. the bugs) is very high. Compare it with the price of a bridge that crashed because the engineer disregarded the laws of mechanics. The price of using wrong software is multiplied because of diversity and the of mass scale of applications of the computer technology in a nowadays world, and also because we are introducing the innovations in a hurry. The wish to base the production of programs on a formalized mathematical calculus, in order to eliminate (dangerous and costly) errors, should be in our opinion the minimal goal.

There is a lot of discussion on the necessity of acceleration of software production, on its automatization. We can not judge in this moment whether the work of programmers is possible to automatize. Let us remark, however, how little tools exists which help in the labor of programming. It is, *sui generis*, a paradox that there exist many programs which help in the work of designers of machines or electronic circuits, architects, physicians, composers, creators of almost all professions, and there are not tools addressed to the profession of programmer.

The quick design and testing of effective algorithms does not cover the whole problem. It is not the solitary work of an ingenious programmer which decides about the results. The work on software is more and more the collective, or social, work. Apart of the author(s) of a program, and obviously of computer, we meet the users, the people maintaining the software et al. The process of using software requires a language to communicate. This language is not a programming language, nor an ethnic language of a certain human group. The production of software leads to creation of texts. Texts have to meet certain requirements (specifications). Hence we have a problem in which at least three elements can be distinguished.

Fig.1.1

This are the arguments, their strength, correctness and quality, which decide whether programmer will convince user or commissioner that his program meets the requirements. The language in which the process of software production takes place must assure the possibility of formulating the requirements a program is to meet, the possibility of writing programs and the

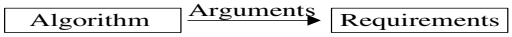


Figure 1.1: Algorithm & Specification

possibility of formulating the arguments as well as the results of analysis.

REMARK

Remark 1.1.1 *We are not going to force the people involved in the process of software use and production, to formalize all their work. We are appealing to the imagination of the reader. The work on software should be formalizable. (Very much like mathematics: the papers published by mathematicians are not formal proofs they are communiques about the possible formalization of proofs). On the other hand we would like to make a remark on the more and more distributed computer systems which help lawyers in their work. If the work of lawyers can be supported by the formalized systems (for every computer system is formal), why the programmers can not profit from the tools they are working with?*

Our beliefs, listed below, are in connection with the above considerations:
- Before passing an algorithm or a programming system to the user(s) one should make an analysis concerning the quality of the software product offered. To neglect it, is hazardous in economic aspects and sometimes it can be hazardous to a humans life.

- How to solve this dilemma? To convince the buyer of our products that the program has been constructed in accordance to the state of art and the present state of knowledge. It is obviously a program minimum. Our proposal goes beyond this minimum and offers the possibility of deeper analysis of software,(c.f.chapter 5).

- An algorithm is only an expression, it is the interpretation of signs which appear in it that decides about the meaning of the expression, (c.f.3.2.4)

- The interpretation can be precisely characterized assuming certain axioms (cf. chapter 5)

- An analysis of programs can be done on a formal base, proofs of semantical properties of programs can be given (cf. chapter 4)

What are the questions which appear during the work on software? We think that the problems encountered can be classified into one of the three groups:

(1) Is the problem solvable at all? It is an economical nonsense to spend money on squaring the circle, construction of perpetuum mobile. Frequently

we heard the promises of that type, e.g. when a company promised to sell a program which was going to answer whether any other program given as a data is correct or not. In other words: avoid inconsistent specifications.

(2) Is the program we constructed or obtained (bought, stolen...) computing the correct solution of the problem. Again an answer to a question of that kind has economic consequences. In other words: should we pay? is it worth its price?

(3) Is the given program the best solution to the problem? In other words shall we pay again for a better product?

It was pointed several times that the answer to the questions (2) and (3) can be obtained only by proving. A proof is also required for a negative answer to the question (1). It is obvious that in the case (1) an appropriate construction i.e. an appropriate experiment, is of importance. Neither theoretical considerations nor experiments can pretend to be the one and only one method of work on software. The role of experiment is caused by the fact that our programs are to work, are to be applied in someone's job. The importance of theory follows from the mass application of software and from the fact that no one experiment can decide that the program in question is error-free, and will work in a reliable manner in all possible situations. An experiment can help in detection of an error, but it can never be used as an argument that a program is correct. Coming back to our example with the construction of a bridge, we think that in a similar way one should do with programs: during the design and construction one should use all the accessible theoretical knowledge and prove that it has specified properties, and later one should experiment with it. For it is the experiment which can help us to learn the complexity of algorithm, e.g. to estimate the coefficients of a polynomial describing the time needed to compute the solution (if it is a polynomial).

1.2 Which logic?

The expressions which appear during the analysis of programs can not be limited to first-order formulas only. They require special tools to express the properties of programs.

There exist deductive systems appropriate for conducting considerations about semantical properties of programs: logics of programs. Here we shall propagate the algorithmic logic, created and developed in Poland since 1968. The reason for doing this is simple, it is better studied and has more applications than others, younger theories.

The deductive system of algorithmic logic can be used in specification of

data structures and modules of programs that correspond to them. This application of algorithmic logic seems more important than proving properties of algorithms.

Algorithmic logic, its axioms and inference rules are helpful in defining the semantics of a programming language.

We claim that for work with programs the tools offered by the first-order logic are not satisfactory. First-order logic is not able to express properties of programs. It turns out that certain properties of algorithms are equivalent to mathematical properties known to be expressible in the language of first-order logic. The list of these properties is long and one can find in it the properties basic for mathematics, e.g. "to be a natural number", "to be an archimedean ring" etc. In the sequel we present these properties and show that they are in fact algorithmic properties. Mathematical logic developed many non-classical variants, e.g. infinitary logic which admits enumerable disjunctions, logics of higher orders etc. These logics are strong enough to express all the properties not expressible in first-order logic. The majority of interesting properties of programs can be studied in these logics. For example, the property "program while g do K od has no infinite computations" can be expressed as an infinite disjunction of formulas, each formula (they increase their length!) saying the computation will have n iterations of the instruction K. Similarly, one expresses this property of programs using quantifiers that operate on finite sets (it is the weak second order logic). It seems to us that the application of such formalisms is not especially attractive to programmers. For it would imply the necessity of, first, translation of software problems into the language say, second-order logic, next, to conduct the analysis of problem, and finally, to translate the results back to the programmer's language. Even if all the steps of the described procedure were fully mechanizable it would be something unnatural, artificial for a programmer. Therefore we propose algorithmic logic. The algorithmic logic plays for computer science the role which the mathematical logic plays for mathematics.

The expressions of algorithmic logic are constructed from programs and the logical formulas. There is no need to use infinitary disjunctions nor the second-order quantifiers. The formulas of algorithmic logic have the power to express all semantical properties of importance in programmer's practice.

Algorithmic logic AL supplies the tools needed to perform deductive reasonings. This makes possible to analyze properties of programs, a priori, before a computational experiment.

The present authors think that the importance of proposed tools lies more in the area of documentation (specification), of discourse between programmer and the user or commissioner of software than in the area of proving programs. In chapter 5 we give a detailed example of such an application

of algorithmic logic. We show that in the phase of design of a program one can distinguish modules. That the goals the modules are to achieve, can be described axiomatically. That the analysis of such axiomatic theories has in many cases a positive impact for the process of programming. The statement says simply, one can build algorithmic theories of data structures. We are also pointing to the connection between the notion of interpretation of one algorithmic theory in another theory with the notion of implementation of corresponding data structures.

We have heard the critical voices that algorithmic logic is not for men, for it contains the rules with infinitely many premises? How to respond? First, let us ask the critic: is it possible to construct any deductive system which will describe the behavior of programs, and will be complete and finitary? The answer is, no. Secondly, after closer examination most of proposed finitary systems turns out to be infinitary (c.f. discussion in 8.3). Thirdly, certain researchers say we shall avoid the rules with infinitely many premises taking as axioms all formulas valid in a given data structure (or in a given class of data structures). Well, formally everything is o.k. One can play with definitions at his discretion. But let us ask what we gain? What we loose? It seems that in this trade we loose something. Do we have any method of recognizing which formulas of first-order logic are valid in a given data structure? It is the goal of mathematicians to increase our knowledge, to add new facts to the thesaurus of already known theorems. And it seems that this work will have no end at all. Stated in other words the set of all formulas of first order arithmetic valid in the standard model of natural numbers is hyperarithmetical.

We think that one should find the arguments for validity of formulas. It is not worthwhile to assume all valid formulas as axioms. One could add the remark that the set of valid formulas asserting correctness of programs w.r.t. quantifier-free formulas lies much lower in the Kleene-Mostowski hierarchy of undecidable sets.

Let us try to express our doubts in another way. Various abstract data types are studied on the base of algebraic (mostly equational, sometimes first-order logic) theories. Certain sets of equations are assumed as axioms to define the structures. Then it turns out that axioms are not enough to define precisely the class of data structures we had in mind. This fact holds for the most of data structures of interest in computer science and is well known from the studies on the classical predicate calculi. There is no finite set of first-order formulas which would constitute a precise enough axiomatization of the structure of natural numbers. Similar statements hold for the other data structures important in computer science. In this situation we should search such tools which enable to reach both goals: a definability (i.e.

axiomatization) of data structures and an ability to analyze computations in the structures. Proposal of algorithmic logic unifies work in both directions in a harmonious way. Many other proposals seem to miss the most important goal which is the analysis of programs with the possibly simplest tools used in it.

It could happen that a reader expected from us a book on mathematical logic in its classical shape. In such case we can advise quite numerous literature [44,33,41,21], see also the short introduction in chapter 2.

To the following two arguments justifying our choice:

1. (a) Proofs in classical logic are also proofs in algorithmic logic.

   (b) The language of first-order logic has too little expressive power to be used in work with programs and their semantical properties.

2. we would like to add one more remark:

   (a) The metatheory of algorithmic logic differs essentially from the metatheory of classical logic. This sentence is true both with respect to the model theory (logics of programs do not enjoy the popular properties like: the "upward" Skolem-Lövenheim theorem, Łoś' theorem on ultraproduct et.al. cf.[44]) as to the theory of proof (properties like Craig's interpolation theorem, Beth's theorem on elimination of definitions, theorem of Robinson etc cf.[33]).

We would like to stress the similarities among various logics of programs. This, what is common to all logics of programs, is more or less universal ability to express semantical properties of programs by formulas of the corresponding language and deductive tools which enable to trade the process of semantical verification of program properties for the process of proving the formulas expressing these properties. Obviously, there are also differences.

In the volume presented to the reader we deal with many questions which arise in a natural way. We begin with discussion of certain simple programming language, or more precisely from a class of languages. Each language will be conceived as a pair consisting of an alphabet and of a set of well formed expressions. The alphabet contains subsets of signs: variables, functors, predicates, signs of logical operations, signs of program operations, quantifiers and auxiliary signs. In the set of well formed expressions we distinguish three subsets: set of terms, set of formulas, set of programs. By analyzing the examples we shall see that program is an expression without a determined meaning, its meaning does not follow just from its syntactical

structure. One program can have many different meanings depending of the interpretation which is associated to the signs occurring in the program. If an interpretation is fixed we can talk of a data structure in which program computes. The remark that one program can be interpreted in various data structures is important not only to those who wish to learn about the nature of things. It has also practical aspect. It enables, namely, to use one algorithm in various contexts to achieve quite different goals. In this way we make a liaison to the advanced methodology of programming, to abstract data types, to object-oriented programming etc.

How to describe the meaning of a program? We are convinced that the majority of us connects its own intuitions with the concept of a computational process. Here we use a definition of computation determined by a program, a data structure (i.e. an appropriate algebraic system) and a valuation of variables. Valuation of variables is a formal counterpart of the notion of a memory state. Data structure is a formal counterpart of an ALU arithmetic-logical unit of computer. This notion is more general, more abstract and therefore we can analyze the behavior of programs written in modern programming languages as well as assembler programs. A computation can be finite of infinite, successful or not i.e. aborted, correct or not. These or others semantical properties can be enjoyed by computations. The semantical properties are of most interest to a user of a program. We mentioned already that reducing the analysis of algorithms to the experiments only is erroneous from the methodological point of view. Experiments, often called debugging, can help to detect certain facts. They can not be the evidence for a conclusion of the type: "all computations of a program will be correct". How to solve the dilemma? We propose to express semantical properties of programs by formulas. Then by studying the validity of formulas we will be able to analyze semantics. We shall say that a semantical property SP is expressible by a formula a, if the formula is valid if and only if when the semantical property SP takes place. In the sequel we shall construct a language in which one can find programs and formulas expressing semantical properties of programs. In a natural way we shall come to the next problem of finding a deductive system which enable to prove validity of formulas. The logical calculus we are going to build, will be determined by two sets: set of axioms and set of inference rules. The sets must be chosen in the way which guarantees the consistency. That is, no a false statement can be proved in the system. Moreover, for every valid formula should exists a proof. This will allow us to replace the job of analysis of semantical properties of programs by a job of analyzing whether a formula is true or not, which in turn is replaced by a job of proving formulas.

The constructed calculus finds diverse applications: in the analysis of

program's properties, in specification of software that is to be prepared, in eduction of programmers etc. It turned out that algorithmic formulas adequately describe initial and terminal conditions of programs. Therefore they can form an appendix to agreements on construction of software. The formulas can be also used in specifications of software systems as well. It turned out that the language is suitable in construction of theories of data structures. It is possible to develop programming systems together with their specifications. The specifications will have the form of systems of axioms that describe the modules of a designed software. Then the implementation of a system can be viewed as a problem of finding a model for a theory. On the other hand the axioms can be used as a criterion of correctness of an implementation.

The book could not cover many topics, even if they seem important. Our choice was following our private taste. We were preferring the questions important to the programmers practice. First part of the book contains general informations, basic notions of universal algebra and logic (chapter 2), presentation of algorithmic logic (chapters 3 and 4), applications of the logic (chapters 4 and 5). Also chapter 8 should be included into this part. The second part of the book is, sui generis, invitation to research. We would be happy to hear of more applications of axioms of procedures which are presented in chapter 6. In chapter 7 we present mathematical models of concurrent computations. We are not satisfied with the axiomatic systems describing these models.

Outside the scope of the book remains extremely interesting and important problem of axiomatic definition of modern programming tools like: classes, coroutines, inheritance, exception handling etc. Nevertheless, we hope that the future research will bring an answer to the question whether it is possible to build programming systems helping to produce software. We offer algorithmic logic as an element of this future construction.

Chapter 2

Data structures

The meaning of programs, their computations, depends on the data structure on which the program is evaluated. The properties of functions and of relations within the data structure determine the properties of programs. The same program realized on two different structures may behave differently (e.g. when computed in 16- or 32-bit arithmetic). On the other hand, two different programs may behave similarly due to the properties of the data structure on which the computations are performed (e.g. if an operation is commutative, and the programs differs only by the order of arguments).

In this book we present an opinion, that data structures are just algebraic structures, stressing in this way their abstract character and their algebraic and algorithmic properties. For programmers the notion of data structure is usually associated with the practical representation of data in the computer rather than with the abstract operations and the algebraic properties. In the sequel we will show that both approaches can be smoothly combined.

2.1 On algebraic structures

Let $X_1, X_2, ..., X_n$ be nonempty sets and let $X_1 \times X_2 \times ... \times X_n$ denote the n-argument cartesian product

$X_1 \times X_2 \times ... \times X_n = \{(x_1, ..., x_n) : x_i \in X_i \text{ for } 1 \leq i \leq n \}$.

Definition 2.1.1 *A subset of the cartesian product $X_1 \times X_2 \times ... \times X_n$ will be called an n-argument* relation *(two-argument relations are called binary relations).*
Let Y be a nonempty set, let f be a subset of $X_1 \times X_2 \times ... \times X_n \times Y$ such that for any $x_i \in X_i$ $1 \leq i \leq n$, $y, y' \in Y$, $(x_1, ..., x_n, y) \in f$ and $x_1, ..., x_n, y\prime) \in f$ implies $y = y'$, then f will be called an n-argument partial function*, written usually as $f : X_1 \times X_2 \times ... \times X_n \longrightarrow Y$*

When f is a partial function and $(x_1, ..., x_n, y) \in f$, then we write, following the standard notation, $y = f(x_1, ..., x_n)$.
The set $Dom(f)$ is called the domain *or the set of arguments of the function f, i.e.*
$Dom(f) = \{(x_1, ..., x_n) \in X_1 \times X_2 \times ... \times X_n : (\exists y \in Y)\ (x_1, ..., x_n, y) \in f\}$.
The set $Rg(f)$ is called the codomain *or the set of values of the function f*
$Rg(f) = \{y \in X : (\exists\ \mathbf{x} \in X_1 \times X_2 \times ... \times X_n)\ f(\mathbf{x}) = y\ \}$.
If $Dom(f) = X_1 \times X_2 \times ... \times X_n$, then we shall say that f is a total *function or simply a function. Functions and partial functions will be called operations. n is the* arity *of the function f.*

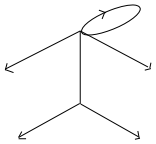
Obviously each n-argument operation is an $n+1$-argument relation. Conversely, each n-argument relation r determines certain total function f_r, called the characteristic function of the relation r, and defined as follows

$$f_r(x_1, ..., x_n) = \Big\{\{.1 \quad \text{if } (x_1, ..., x_n) \in r0 \quad \text{if } (x_1, ..., x_n) \notin r$$

Definition 2.1.2 *By an* algebraic structure *of type $< n_1, ..., n_k; m_1, ..., m_l >$ we shall mean any system of the form*
$\mathbb{A} = < A, f_1, ..., f_k; r_1, ..., r_l >$
such that
- A is a nonempty set,
- for each $1 \leq i \leq k$, f_i is an n_i-argument operation on the set A,
- for each $1 \leq j \leq l$, r_j is an mj-argument relation on the set A.
The set A is called the universe *of the algebraic structure $\mathbb{A}$. The sequence of natural numbers $< n_1, ..., n_k; m_1, ..., m_l >$ is the* type *of the system $\mathbb{A}$. An algebraic structure without relations is called an* algebra.

Example 2.1.3 *The simplest example of an algebraic structure is a* graph. *This is the system of the form $< V, E >$ where the set V is the set of vertices, and the relation E is the set of edges. Figure 2.1 shows a simple graph, whose set of vertices is {1,2,3,4,5,6}, and whose set of edges is {(1,2), (1,3), (1,4), (1,1), (4,5), (4,6)}.*

Example 2.1.4 *The two-element* Boolean algebra B_0 *is the algebraic structure of the form*
$< B_0, \cup, \cap, -, 1, 0 >$ *of type $< 2, 2, 1, 0, 0 >$ such that $B_0 = \{1, 0\}$ and the operations $\cup$, $\cap$,- are defined for arbitrary $a, b \in B_0$ as follows*
$a \cup b = 1$ *iff* $a = 1$ *or* $b = 1$
$a \cap b = 1$ *iff* $a = 1$ *and* $b = 1$



$-\ a = 1$ *iff* $a = 0$.
Elements of the set B_0 *are usually interpreted as the values truth (1) and false (0). The operations* $\cup$, $\cap$, $-$ *are known, respectively, as disjunction, conjunction and negation.*

Example 2.1.5 *Let suc denote the successor function on the set of natural numbers* N. *Let* 0 *be a constant, i.e. a zero-argument operation on* N *and* $=$ *the equality relation on* N.*The system* $N =< N,\ suc, 0; = >$ *is the well known algebraic structure of type* $< 1, 0; 2 >$. *We call this structure the* standard structure of arithmetic of natural numbers.

Definition 2.1.6 *Two algebraic structures* $\mathbb{A}$ *and* $\mathbb{B}$ *will be called* similar *iff their types are the same, i.e. structures*
$\mathbb{A} =< A, f_1, ..., f_k; r_1, ..., r_l >$ *and* $\mathbb{B} =< B, f'_1, ..., f'_s; r'_1, ..., r'_t >$
are similar if and only if
(a) $k = s, l = t$,
(b) f'_i *is an* n_i*-argument operation on* B *iff* f_i *is an* n_i*-argument operation on* A, *for each* $1 \leq i \leq k$, *and*
(c) r'_j *is an* m_j*-argument relation in* B *iff* r_j *is an* m_j*-argument relation on* A , *for each* $1 \leq j \leq l$.
If algebraic structures $\mathbb{A}$ *and* $\mathbb{B}$, *defined as above, are similar, then* f_i *and* f'_i, *for* $1 \leq i \leq k$, *are corresponding operations, and* r_j *and* r'_j, *for* $1 \leq j \leq l$, *are corresponding relations in* $\mathbb{A}$ *and in* $\mathbb{B}$, *respectively .*

Example 2.1.7 *Let* $\mathbb{A} =< A^*, \circ; \prec >$ *be an algebraic structure such that* A^* *is the set of all words over the alphabet* A *including the empty word* ε
$A^* = \bigcup_{n \in N} A_n$
$\circ$ *is the two-argument operation of concatenation (i.e. if* $w_1, w_2 \in A^*$, *then the word* $w_1 w_2$, *which is obtained by writing the word* w_2 *after* w_1, *is the result of applying operation* $\circ$ *to the words* w_1 *and* w_2*),*
$\prec$ *is the binary relation such that* $w_1 \prec w_2$ *iff* $w_2 = w_1 w_3$ *for some* $w_3 \in A^*$.
The system $A =< A^*, \circ; \prec >$ *is similar to the system* $< R, +; \leq >$, *where*

*R denotes the set of real numbers, + is addition on R, and $\leq$ is the ordering relation on R, since both systems are of type $< 2;2 >$. The system $A =< A*, ^\circ; \prec >$ is similar neither to the system $< R, +, *; \leq >$ nor to the system $< R, ^{-1}; =>$, where * denotes multiplication and $^{-1}$ is the one-argument reciprocal operation.*

Let h be a function such that $Dom(h) = A$ and $Rg(h) \subseteq B$, then we shall say that h is a mapping of the set A into the set B, and denote it by $h : A \longrightarrow B$. If for any elements $a, b \in A$, a$\neq$b implies $h(a) \neq h(b)$, then h is called a *one-to-one* function. If each element of the set B is a value of h, i.e. $Rg(h) = B$, then h is a mapping *onto* the set B.

Definition 2.1.8 *Let $\mathbb{A}$ and $\mathbb{B}$ be similar algebraic structures*
$\mathbb{A} =< A, f1,..., fk; r1,...,rl >$, $\mathbb{B} =< B, g1,...,gk; s1,...,sl >$.
Any function $h : A \longrightarrow B$ such that the following three properties hold is called a homomorphism *from $\mathbb{A}$ into $\mathbb{B}$:*
(1) for each n_i-argument operation f_i on A and for all $a_1, ..., a_{n_i} \in A$,
if $(a_1, ..., a_{n_i}) \in Dom(f_i)$, then $(h(a_1), ..., h(a_{n_i})) \in Dom(g_i)$,
(2) if $(a_1, ..., a_{n_i}) \in Dom(f_i)$, then $h(f_i(a_1, ..., a_{n_i})) = g_i(h(a_1), ..., h(a_{n_i}))$,
(3) for each m_j-argument relation r_j on A and for all $a_1, ..., a_{m_j} \in A$,
if $(a_1, ..., a_{m_j}) \in r_j$, then $(h(a_1), ..., h(a_{m_j})) \in s_j$.

Example 2.1.9 *Let us consider the algebraic structure $< A*, ^\circ; \prec >$ (cf. example 2.1.4) and the system $< N, +; \leq >$ with the natural interpretation of + and $\leq$. The function $h : A* \longrightarrow N$ such that such that $h(\varepsilon) = 0$, where ε denotes the empty word,*
$h(x) = 1$ for $x \in A$
$h(w^\circ x) = h(w) + h(x)$ for any $w \in A^*$ and $x \in A$
is a homomorphism which maps A^* into N.
Conditions (1) and (2) of the definition 2.1.4 are satisfied in the obvious way. For the proof of property (3) let us take any $w_1, w_2 \in A^*$. We then have
$w_1 \prec w_2 \Longrightarrow (\exists w_3) w_2 = w_1{}^\circ w_3 \Longrightarrow (\exists w_3) h(w_2) = h(w_1) + h(w_3) \Longrightarrow h(w_1) \leq h(w_2)$
which ends the proof.

Definition 2.1.10 *If h is a homomorphism from $\mathbb{A}$ onto $\mathbb{B}$ and h is one-to-one function such that*
(1) for any elements $a_1, ..., a_{n_i}$ of the structure $\mathbb{A}$ and for each n_i-argument function f_i in A,
$(a_1, ..., a_{n_i}) \in Dom(f_i)$ iff $(h(a_1), ..., h(a_{n_i})) \in Dom(g_i)$ and
(2) for any elements $a_1, ..., a_{m_j}$ of the structure $\mathbb{A}$ and for each m_j-argument

relation r_j *in* A,
$(a_1, ..., a_{m_j}) \in r_j$ *iff* $(h(a_1), ..., h(a_{m_j})) \in s_j$,
then h *is called* isomorphism.
If there exists an isomorphism from $\mathbb{A}$ *onto the structure* $\mathbb{B}$, *then* $\mathbb{A}$ *and* $\mathbb{B}$ *are called* isomorphic structures, *abbreviated to* $\mathbb{A} \approxeq \mathbb{B}$.

Example 2.1.11 *Let* $+_m$ *be addition modulo* m *in the subset of the set of natural numbers* $\mathbb{Z}_m = \{0, 1, ..., m-1\}$. *Consider the algebraic structure of remainders modulo* m,
$\mathbb{Z}_m =< \{0, 1, ..., m-1\}, +_m >$,
and also the algebraic structure $\mathbb{A}$ *of 0-1 sequences of length* $n+1$ *with the two-argument operation* $+$, $< \{0,1\}^{n+1}, + >$, *where* $+$ *is defined as follows : given sequences* $a = \{a_0, ..., a_n\}$ *and* $b = \{b_0, ..., b_n\}$, *the result of the operation* $+$ *on arguments* a *and* b *is the sequence* $\{c_0, ..., c_n\}$ *such that for* $j \leq n$ $c_j = (a_j + b_j + p_{j-1}) \ mod\ 2$, *where the sequence of remainders* (p_j) $1 \leq j \leq n$ *is defined by* $p_{-1} = 0$ *and for* $j \geq 0$

$$p_j = \begin{cases} 1 & \text{if } (a_j + b_j + p_j - 1) \geq 2 \\ 0 & \text{otherwise} \end{cases}$$

Let $m = 2^{n+1}$ *and let* h *be the function which maps the set* $\{0,1\}^{\ n+1}$ *into the set* $\mathbb{Z}_m$ *and such that* $h(f) = \sum\limits_{0 \leq i \leq n} 2^i * f(i)$ *for* $f \in \{0,1\}^{n+1}$. *Obviously, for any* $f \in \{0,1\}^{n+1}$ *the following inequality holds* $\sum\limits_{0 \leq i \leq n} 2^i * f(i) < 2^{n+1}$ *i.e.* $h(f) \in \mathbb{Z}_m$. *Thus* h *is a mapping into the set* $\mathbb{Z}_m$. *Moreover, for every number* $k \in \mathbb{Z}_m$ *there exists a uniquely defined sequence* $f \in \mathbb{Z}_m$ *such that* $h(f) = k$. *In fact, it is enough to take* $f(i) = (k \ div\ 2^i) \ mod\ 2$ *where div denotes the operation of integer division.* h *is a one-to-one function, since, for different sequences* f, g, *if* i *is the greatest natural number such that* $f(i) \neq g(i)$, *then for* $f(i) = 1$ *and* $g(i) = 0$, *we have* $h(f) - h(g) = 2^i + \sum\limits_{j<i} 2^j f(j) - \sum\limits_{j<i} 2^j g(j) > 0$.

zle nie powinnam uzywac odejmowania!!!!!!!!!!!!!!!!!!!!!!
Moreover, if $f + g = (c_0, ..., c_n)$, then

$$h(f+g) = \sum_{i \leq n} 2^i * c_i = \sum_{i \leq n} 2^i (a_i + b_i + p_{i-1}) mod 2 =$$

$$\sum_{i \leq n} 2^i (a_i + b_i + p_{i-1} - 2(a_i + b_i + p_{i-1})) div 2 =$$

$$\sum_{i\leq n} 2^i a_i + \sum_{i\leq n} 2^i b_i + \sum_{i\leq n} 2^i * (p_{i-1} - 2 * (a_i + b_i + p_{i-1})div2)$$

Notice that for each $0 \leq i \leq n$, $p_i - (a_i + b_i + p_i - 1)$ div $2 = 0$ and

$$(a_n + b_n + p_n - 1)div2 = \begin{cases} 1 & \text{if } (\sum_{i\leq n} 2^i a_i + \sum_{i\leq n} 2^i b_i) \geq m \\ 0 & \text{otherwise} \end{cases}$$

Finally,

$$h(f+g) = h(f) + h(g) - 2^{n+1}(a_n + b_n + p_{n-1})div2 = h(f) + \ h(g).$$

Thus h is an isomorphism which maps the system $\mathbb{A}$ onto $\mathbb{Z}_m$

Definition 2.1.12 *By a congruence on an algebraic structure $\mathbb{A}$ we mean an equivalence relation $\sim$ on A such that*
(1)for each n_i-argument operation f_i in A and for all elements $a_1, ..., a_{n_i}, a'_1, ..., a'_{n_i} \in A$, if $a_j \sim a'_j$ for $1 \leq j \leq n_i$, then $(a_1, ..., a_{n_i}) \in Dom(f_i)$ iff $(a'_1 ..., a'_{n_i}) \in Dom(f_i)$ and for $(a_1, ..., a_{n_i}) \in Dom(f_i)$ $f_i(a_1, ..., a_{n_i}) \sim f_i(a'_1, ..., a'_{n_i})$,
(2) for each m_j-argument relation r_j and for all elements $a_1, ..., a_{m_j}, a'_1, ..., a'_{m_j} \in A$, if $a_i \sim a'_i$ for $1 \leq i \leq m_j$, then $(a_1, ..., a_{m_j}) \in r_j$ iff $(a'_1, ..., a'_{m_j}) \in r'_j$.

Example 2.1.13 *Consider the algebraic structure $< N, +, *, 0 >$ and for some fixed number k a binary relation $\approx_k$ on N such that*
$m \approx_k n$ iff $m \bmod k = n \bmod k$, for all $n, m \in N$.
The relation $\approx_k$ is reflexive, symmetric, transitive and therefore is an equivalence relation on the set N. The very simple verification of this fact is left to the reader. The relation $\approx_k$ is a congruence on the system $< N, +, *, 0 >$. Let us briefly check this. Suppose that $m_1 \approx_k n_1$ and $m_2 \approx_k n_2$, then there are natural numbers $i_1, i_2, j_1, j_2, r_1, r_2$, such that
$m_1 = i_1 * k + r_1 m_2 = i_2 * k + r_2$,
$n_1 = j_1 * k + r_1 n_2 = j_2 * k + r_2$.
Thus $m_1 + m_2 = (i_1 + i_2) * k + r_1 + r_2$,
$n_1 + n_2 = (j_1 + j_2) * k + r_1 + r_2$.
The last two equations imply that $(m_1 + m_2) \bmod k = (n_1 + n_2) \bmod k$.
*Analogously, for multiplication *,*
$m_1 * m_2 = (i_1 * i_2) * k + (r_1 * i_2 + r_2 * i_1) * k + r_1 * r_2$,
$n_1 * n_2 = (j_1 * j_2) * k + (r_1 * j_2 + r_2 * j_1) * k + r_1 * r_2$.
Dividing $m_1 * m_2$ by k we obtain the same remainder as dividing $r_1 * r_2$ by k, which is equal, according to the above remarks, to the remainder obtained from dividing $n_1 * n_2$ by k. Hence $(m_1 * m_2) \bmod k = (n_1 * n_2) \bmod k$.

Example 2.1.14 *Consider the system* $< A*, ^{\circ}; \prec >$ *and the homomorphism* $h : A \longrightarrow N$ *defined as in example 2.1.5. Let* $\approx$*be the equivalence relation defined as follows: for any* $u, w \in A^*$, $,u \approx w$ *iff* $h(u) = h(w)$ *.For* $u \approx u'$ *and* $w \approx w'$ *we have* $h(u^{\circ}w) = h(u) + h(w) = h(u') + h(w') = h(u'^{\circ}w')$ *, hence* $u^{\circ}w \approx u'^{\circ}w'$*. Nevertheless,* $\approx$ *is not a congruence relation when* A *contains more then one element. Indeed, if* x, y *are different elements of* A*, then although* $x \approx x, xx \approx yy$ *and* $x \prec xx$*, it is not the case that* $x \prec yy$*.*

Definition 2.1.15 *Let* $\mathbb{A}$ *be an algebraic structure*
$\mathbb{A} =< A, f_1, ..., f_k; r_1, ..., r_l >$
of type $< n_1, ..., n_k; m_1, ..., m_l >$ *and let* $\approx$ *be a congruence on* $\mathbb{A}$*. The system*
$\mathbb{A}/\approx =< A/\sim, f'_1, ..., f'_k; r'_1, ..., r'_l >$
is called a quotient *structure, if*
(1) $A/\approx$ *is the set of equivalence classes of the relation* $\approx$*, i.e.*
$A/\approx = \{[a] : a \in A\}$*, where* $[a] = \{b \in A : a \approx b\}$*,*
(2) f'_i *is* n_i*-argument operation on* $A/\approx$*such that for all* $a_1, ..., a_{n_i} \in A$*,*
$([a_1], ..., [a_{n_i}]) \in Dom(f'_i)$ *iff* $(a_1, ..., a_{n_i}) \in Dom(f_i)$
and for all $([a_1], ..., [a_{n_i}]) \in Dom(f'_i)$*,*
$f'_i([a_1], ..., [a_{ni}]) = [f_i(a_1, ..., a_{n_i})]$
(3) r_j *is* m_j*-argument relation on* $A/\approx$ *such that for all* $a_1, ..., a_{m_j} \in A$*,*
$([a_1], ..., [a_{m_j}]) \in r'_j$ *iff* $(a_1, ..., a_{m_j}) \in r_j$*.*

Example 2.1.16 *Let us consider a congruence* $\sim_m$ *from the example 2.1.7. The quotient structure* $< N/\sim_m, \otimes >$ *is isomorphic to the structure* $\mathbb{Z}_m$ *defined in the example 2.1.6. The only operation* $\otimes$ *of the system is defined according to the following definition: for all* $a, b \in N$
$[a] \otimes [b] = [a + b]$*.*
Putting $h([a]) = a \bmod m$*, we define a one-to-one function* h *from* $N/\approx m$ *onto* $\mathbb{Z}_m$*, which is an isomorphism between the two systems.*

In the sequel, when the names of relations and functions are not important we shall write, for short $\mathbb{A} =< A, \Omega_F; \Omega_R >$, to denote algebraic structure with a sequence of functions WF and a sequence of relations Ω_R in $\mathbb{A}$.

Most of applications deal with algebraic structures which are many-sorted structures. A structure is many-sorted if its universe is the union of disjoint subsets called sorts. Different arguments of a function or of a relation in such a system may belong to different sorts. As an example, consider the vector-space, in which we have two kinds of sets: Vectors and Scalars (components of vectors). The operation of scalar multiplication is then defined on the cartesian product $Vectors \times Scalars$

When specifying functions in a many-sorted structure with elements of different sorts, we must declare not only the arity of an operation but also

the sorts of its arguments. This will be called the type of the operation (or relation). In our example the type of scalar multiplication is $(Vectors \times Scalars \longrightarrow Vectors)$.

Definition 2.1.17 *Let A be a set such that, for some number t,*
$A = A_1 \cup ... \cup A_t$ and $A_i \cap A_j = \emptyset$ for $i \neq j$, $1 \leq i, j \leq t$.
Let type (Ω_F) denote the sequence of types of the functions in Ω_F and let type(Ω_R) denote the types of the relations in Ω_R .
A system $A =< A, \Omega_F; \Omega_R >$ is called a many-sorted algebraic structure of type $< t, type(\Omega_F), type(\Omega_R) >$, or a t-sorted structure, iff
(1) for each operation $f \in \Omega_F$ of type $(i_1 \times ... \times i_n \longrightarrow i_{n+1})$
$Dom(f) \subseteq A_{i_1} \times ... \times A_{i_n}$ and $Rg(f) \subseteq A_{i_{n+1}}$,
(2) for each relation $r \in \Omega_R$ of type $(j_1 \times ... \times j_m)$, $r \subseteq (A_{j_1} \times ... \times A_{j_m})$.

Examples of many-sorted structures will be presented in the next section. For the moment, we note simply that every many-sorted structure is an algebraic structure in the sence of definition 2.1.2. If f is an operation of type $(i1 \times ... \times in \longrightarrow i(n+1))$ in some many-sorted system,then writting for all $x_1, ..., x_n$

$$f(x_1, ..., x_n) = \begin{cases} .f(x_1, ..., x_n) & \text{for} \quad (x_1, ..., x_n) \in Dom(f) \\ \text{undefined} \end{cases}$$

defines an n-argument partial function f.

Let $\mathbb{A} =< A, \Omega_F; \Omega_R >$ and $\mathbb{B} =< B, \Omega'_F; \Omega'_R >$ be two many-sorted structures with sorts $A_1, ..., A_t$ and $B_1, ..., B_{t'}$, respectively.

Definition 2.1.18 *We shall call two many-sorted structures $\mathbb{A}$, $\mathbb{B}$ similar if*
(1) $\mathbb{A}$, $\mathbb{B}$ are similar algebraic structure in the sense of definition 2.1.3,
(2) $t = t'$ and
(3) for each function $f \in \Omega_F$ (relation $r \in \Omega_R$) the corresponding function in $\mathbb{B}$ (corresponding relation in $\mathbb{B}$) have the same type.

The notions of homomorphism and isomorphism of many-sorted systems are almost identical to the corresponding notions for algebraic systems. The minor changes required are a consequence of the partition of the universe into different sorts.

Definition 2.1.19 *A mapping h which maps a t-sorted system $\mathbb{A}$ into (onto) a similar system $\mathbb{B}$ is called a homomorphism (isomorphism), if h is a homomorphism (isomorphism) from the algebraic structure $\mathbb{A}$ into (onto) the algebraic structure $\mathbb{B}$ (cf. Definition 2.1.4) and, moreover, for each $i \leq t$, $h(A_i) \subseteq B_i$.*

Lemma 2.1.20 *For any three similar many-sorted systems $\mathbb{A}$, $\mathbb{B}$, $\mathbb{C}$, if h_1 is a homomorphism from $\mathbb{A}$ into $\mathbb{B}$ and h_2 a homomorphism from $\mathbb{B}$ into $\mathbb{C}$, then the function h such that $h(a) = h_2(h_1(a))$ for all $a \in A$ (the composition of functions h_1 and h_2) is a homomorphism from the system $\mathbb{A}$ into the system $\mathbb{C}$.*

The important notion of congruence is defined in a slightly more complicated way. We have to ensure that elements of different sorts are not in the same equivalence class.

Definition 2.1.21 *By a* congruence *in the many-sorted structure $\mathbb{A}$ we shall mean an equivalence relation $\approx$in A which satisfies the conditions of definition 2.1.6 and such that, for all $a \in \mathbb{A}$, if $a \in A_i$, then $[a] \subseteq A_i$.*

Let $\mathbb{A}$ be a many-sorted structure. Let us assume that its sorts are $A_1, \ldots, A_t$. Let $\approx$ be a congruence in $\mathbb{A}$. Let B_i denote the set $A_i/\approx$. From the definition of congruence we have $B_i \cap B_j = \emptyset$ for any j$\neq i$. Let f be an arbitrarly chosen n-argument operation in $\mathbb{A}$ of type $(i_1 \times \ldots i_n \longrightarrow i_{n+1})$. We shall define the corresponding function f^* in $B = B_1 \cup \ldots \cup B_t$ as follows $Dom(f^*) = \{([a_1], ..., [a_n]) : (a_1, ..., a_n) \in Dom(f)\}$ and for $([a_1], ..., [a_n]) \in Dom(f^*)$, $f^*([a_1], ..., [a_n]) = [f(a_1, ..., a_n)]$.

For each $i \leq t$ and for each $a \in A_i$, we have $[a] \in B_i$, since $[a] \subseteq A_i$ Hence, $f^* \subseteq B_{i_1} \times \ldots \times B_{i_n} \times B_{i_{n+1}}$ and, in consequence, the type of the function f^* is the same as the type of the function f.

Similarly, for each relation r of type $(j_1 \times \ldots \times j_n)$ in $\mathbb{A}$ we define the relation r^* as follows: for any $(a_1, ..., a_n) \in A_{j_1} \times \ldots \times A_{j_n}$,

$([a_1], ..., [a_n]) \in r^*$ iff $(a_1, ..., a_n) \in r$.

It follows, that $r^* \subseteq B_{j_1} \times \ldots \times B_{j_n}$, and that r^* has the same type as r.

Let us denote by Ω^*_F the set of all functions f^* corresponding to functions $f \in \Omega_F$ and by Ω^*_R the set of all relations r* corresponding to relations r$\in \Omega_R$.

Definition 2.1.22 *The system $\mathbb{A}/\sim =< B, \Omega^*_F; \Omega^*_R >$, where $B = B_1 \cup \ldots \cup B_t$, is called a many-sorted quotient structure.*

It is obvious that, by the definition, systems $\mathbb{A}/\sim$and $\mathbb{A}$ are similar. Moreover, the following useful property holds.

Lemma 2.1.23 *Let $\mathbb{A}$ and $\mathbb{A}/\sim$ be many-sorted algebraic structures defined as above.The function*
$h : A \longrightarrow A/\sim$ such that $h(a) = [a]$ for all $a \in A$,
is a homomorphism from $\mathbb{A}$ onto $\mathbb{A}/\sim$.

The mapping h is a total function from A onto $A/\sim$ (It is not a one-to-one function, however, since for all $b \in [a]$, we have $h(b) = [a]$). Consider any operation f of the system $\mathbb{A}$.

Let $(i_1 \times ... \times i_n \Longrightarrow i_{n+1})$ be the type of f and let $(a_1, ..., a_n) \in Dom(f)$. Then, by the definition of a quotient structure, we have $([a_1], ..., [a_n]) \in Dom(f^*)$ and

$$h(f(a_1, ..., a_n)) = f^*(h(a_1),\ ...\ , h(a_n)) = f^*([a_1], ..., [a_n]) = [f(a_1, ..., a_n)]$$

Consider finally, any relation r of type $(i_1 \times ... \times i_n)$ in $\mathbb{A}$. Suppose that $(a_1, ..., a_n) \in r$. Then $([a_1], ..., [a_n]) \in r^*$, by the definition of a quotient system, and therefore $(h(a_1),\ ...\ , h(a_n)) \in r^*$. It follows that h is a homomorphism. We shall call it the natural homomorphism induced by the congruence $\sim$.

Remark 2.1.24 *Each many-sorted system $\mathbb{A} =< A, \Omega_F; \Omega_R >$ with t sorts $A_1, ..., A_t$ can be treated as a one-sorted structure $\mathbb{A}'$,*
$\mathbb{A}' =< A', \Omega_F; \Omega_R \cup \{sort_i : i \leq t\} >$,
in which the signature is extended by t one-argument relations $sort_i$ defined by the logical equivalence
$sort_i(a) \Longleftrightarrow\ a \in A_i$.
Relations $sort_i$ for $i \leq t$ define types of the structure $\mathbb{A}$. If f is a function of type $A_{i_1} \times ... \times A_{i_n} \longrightarrow A_{i_{n+1}}$, then the necessary condition for the sequence $a_1,\ ...\ , a_n$ belongs to the domain of function f is
$(sort_{i_1}(a_1) \wedge ... \wedge sort_{i_n}(a_n))$.
A similar condition can be formulated for relations.
Obviously, if $\sim$is a congruence in system $\mathbb{A}$, then $\sim$is a congruence in $\mathbb{A}'$.

2.2 Algebraic systems for the representation of finite sets

Definition 2.2.1 *By the standard dictionary data structure, we shall understand a two-sorted algebraic structure $D =< E \cup D, insert, delete; member, empty >$ such that*
(a) $E \cap D = \emptyset$ and $D = Fin(E)$, i.e. D is the set of all finite subsets of E,
insert and delete are functions of type $(E \times D \longrightarrow D)$,
member is a binary relation of type $(E \times D)$,
empty is one-argument relation of type (E) and
$Dom(delete) = E \times (D \backslash \{\emptyset\})$, $Dom(insert) = E \times D$
(b) for arbitrary $e \in E$ and $d \in D$,

$insert(e,d) = d \cup \{e\}$,
$delete(e,d) = d \backslash \{e\}$, *when* $d \neq \emptyset$
$d \in empty$ *iff* $d = \emptyset$,
$(e,d) \in member$ *iff* $e \in d$.

The structure dictionary is a data structure of importance and is often used in informatics. We can apply it when dealing with finite sets and when the operations to be used are the insertion of a new element into a set or the deletion of an element from a set.

Definition 2.2.2 *By the standard stack data structure we shall understand a two-sorted algebraic structure* $S =< E \cup S, push, pop, top; empty, = E >$ *such that*
(1) E *is an arbitrary non-empty set,* S *is the set of all finite sequences with elements in* E*, and*
push is an operation of type $(E \times S \longrightarrow S)$,
pop is an operation of type $(S \longrightarrow S)$,
top is a one-argument operation of type $(S \longrightarrow E)$,
empty is a one-argument relation of type (S),
$=_E$ *is a two-argument relation of type* $(E \times E)$ *and*
(2) for arbitrary $e \in E$ *and* $s \in S$*, if* $s = (e1, ..., en)$*, then*
$pop(s) = (e2, ..., en)$,
$push(e,s) = (e, e1, ..., en)$,
$top(s) = e1$,
If s *is an empty sequence then the values of* $pop(s)$ *and* $top(s)$ *are not defined.*
$=_E = \{(e,e) : e \in E\}$
$s \notin empty$ *iff* s *is a non-empty sequence*
Elements of the set S *will be called stacks.*

Stacks play a fundamental role in the construction of programming languages compilers. This structure is very important during the syntactical analysis of the source text of a program, as well as during execution of a program. A stack whose elements are activation records, is the fundamental tool for the realization of recursive procedures.

Another widely used data structure, which also deals with finite sequences is the structure of queues.

Definition 2.2.3 *By the standard data structure of queues we shall mean a two-sorted algebraic structure*
$< E \cup Q, put, out, first; em, = E, = Q >$,
such that:

E is a non-empty set of elements (of queues),
Q is the set of all finite sequences over E (including the empty set $\emptyset$),
and the operations and relations of the system are defined as follows
put is a two-argument operation of type $(Q \times E \longrightarrow Q)$
out is a one-argument operation of type $(Q \longrightarrow Q)$
$first$ is a one-argument operation of type $(Q \longrightarrow E)$
em is a one-argument relation of type (Q)
$=_E$ is the identity relation of type $(E \times E)$
$=_Q$ is the identity relation of type $(Q \times Q)$
and for any $q = (e_1, ..., e_n) \in Q$ and $e \in E$ we have
$put(q, e) = (e_1, ..., e_n, e)$ $\quad Dom(put) = Q \times E$
$out(q) = (e2, ..., en)$ $\quad Dom(out) = Q\backslash\{\emptyset\}$
$first(q) = e1$ $\quad Dom(first) = Q\backslash\{\emptyset\}$
$q' \in em$ iff q' is the empty sequence $\emptyset$

Queues are data structures often used in simulation programs where the performance of real systems is modelled and analyzed. In operating systems a queue of processes is created to solve access conflicts to the same port, e.g. to a line printer.

Below we present two kinds of tree data structures which can be used for example, in the evaluation of the value of an expression, or for data management.

Definition 2.2.4 *Let At be a non-empty set, elements of which are called* atoms. *Let $Tree$ be the smallest set of expressions which contains,*
for each $a \in At$, an expression (a),
and contains some special element nil,
and such that, for any two elements $t1$, $t2 \in Tree$ such that $t1, t2 \neq nil$, the expression $(t1 \circ t2)$ is also an element of the set $Tree$.
Elements of the set $Tree$ will be called binary trees.

Definition 2.2.5 *By the standard binary tree data structure we shall mean a two-sorted algebraic structure*
$< At \cup Tree,\ cons, left, right; atom, empty >$,
such that
cons is an operation of type $(Tree \times Tree \longrightarrow Tree)$ $left$ and right are operations of type $(Tree \longrightarrow Tree)$
empty and atom are relations of type $(Tree)$, i.e. subsets of the set $Tree$,
and, for $t \in Tree$ and $t1, t2 \in Tree\backslash\{nil\}$,
$cons(t1, t2) = (t1 \circ t2)$, $\quad Dom(cons) = (Tree\backslash\{nil\}) \times (Tree\backslash\{nil\})$
$left((t1 \circ t2)) = t1$, $\quad Dom(left) = Tree\backslash\{nil\}$

$right((t1°t2)) = t2, \qquad Dom(right) = Tree\backslash\{nil\}$
$right(t) = left(t) = nil, \quad$ *if* $atom(t)$
$t \in atom$ *iff* $t = (a) \quad$ *for some* $a \in At$
$t \in empty$ *iff* $t = nil$

Below we present another type of binary tree which finds wide application in sorting.

Let Et be a set linearly ordered by some relation $\prec$. Elements of the set Et will be called labels. Let Tr be the smallest set which contains the expression ( ) and such that, if $e \in Et$ and $t1, t2 \in Tr$, then $(t1et2) \in Tr$. Elements of the set Tr are called labeled binary trees (trees with information at their nodes). If a tree t has the form $(t1et2)$, then $t1$ is called its left subtree and $t2$ its right subtree. Element e is called the label of t.

Definition 2.2.6 *A tree $t \in Tr$ is called a binary search tree, if t is of the form ( ), or of the form $(t1et2)$ and the following conditions are valid*
(1) for any label e' which occurs in $t1$, $e' \prec e$,
(2) for arbitrary label e' which occur in $t2$, $e < e'$,
(3) $t1$ and $t2$ are binary search trees.

Let BST be the set of all binary search trees over the set Et. Each element t of this set is a finite binary tree. Each vertex of the tree is labeled by an element of the set Et. For every subtree t' of the tree t, the label associated with the root of t' is greater than any label associated with a node in its left subtree. Similarly for right subtrees, any label associated with a node of the right subtree is greater that the label of the root of the tree.

Example 2.2.7 *Figure 2.2 represents a binary search tree labeled by natural numbers. The binary tree which is illustrated in Figure 2.3 is not a binary search tree.*

Fig.2.2
Fig.2.3

Definition 2.2.8 *By the standard binary search tree data structure (or, for short, BST structure) we shall mean the two-sorted algebraic structure*
$< Et \cup BST,\ val, left, right, new, upl, upr; isnone, ú, =>,$
where
val is a one-argument operation of type $(BST \longrightarrow Et)$
$val(t_1\ e\ t_2) = e \qquad Dom(val) = BST \setminus \{(\ )\}$
$left, right$ *are operations of type* $(BST \longrightarrow BST)$
$left(t_1\ e\ t_2) = t_1,$

$right(t_1\ e\ t_2) = t_2 \qquad Dom(left) = Dom(right) = BST \backslash \{(\ )\}$,
new is one-argument operation of type $(Et \longrightarrow BST$
$new(e) = ((\ )\ e\ (\ ))$ *for any* $e \in E$,
upl and upr are two-argument operations of type $(BST \times BST \longrightarrow BST)$

$$upl(t, t_1 e t_2) = \begin{cases} (tet_2) & iff(tet_2) \in BST \\ undefined & otherwise \end{cases},$$

$$upr(t, t_1 e t_2) = \begin{cases} (t_1 et) & iff(t_1 et) \in BST \\ undefined & otherwise \end{cases},$$

isnone is a one-argument relation, which is satisfied by the element $(\ )$ *only,*
$\prec$ *is a linear ordering in the set* Et, *and*
$=$ *is the identity relation in* Et.

2.3 Formal language & semantics

The study of properties of concrete data structures requires a language able to express properties of the operations and relations of the structure. In mathematics we use a (not necessarily formalized) language of first-order logic. This language will also constitute the base for our work, cf. [44,33,45]. A short presentation of the formal definition of this language and its semantics is given below.

Definition 2.3.1 *By the alphabet of a first-order language we shall understand a set* $\mathcal{A}$ *which is the union of disjoint sets:*
V *- of individual variables,*
V_o *- of propositional variables,*
P *- of symbols of relations (i.e. predicates),* $P = (\rho_j)_{j \in J}$,
Ψ *- of symbols of functions (i.e. functors),* $\Psi = (\varphi_i)_{i \in I}$,
$\{\wedge, \vee, \neg, \Longrightarrow\}$ *- of logic symbols, called conjunction, disjunction, negation, implication,*
$\{\forall, \exists\}$ *- of symbols of quantifiers, called universal and existential quantifiers respectively,*
$\{\ )\ ,(\ \}$ *- of auxiliary symbols, such as parentheses and comma.*

In what follows we shall assume that the set of individual variables consists of the non-empty disjoint sets V_i, $1 \leq i \leq tt$ for some natural number tt. If $x \in V_i$, then we shall say that x is a variable of type i. Each predicate $\rho \in P$ has a defined type $t(\rho)$ of the form $(i1 \times ... \times i_n \rightarrow 0)$, where n is the

number of arguments (arity of predicate), and $i_1, ..., i_n$ are the types of the arguments of the predicate ρ, $1 \leq i1, ..., in \leq tt$. Each functor $\varphi \in \Psi$ has a defined type $t(\varphi)$ of the form $(i_1 \times ... \times i_n \rightarrow i_{n+1})$, where n is its arity, and $i_1, ..., i_n$ are the types of its arguments, $1 \leq i_1, ..., i_n, i_{n+1} \leq tt$. We shall assume, moreover, that the alphabet is (at most) a countable set.

Definition 2.3.2 *The triple* $< tt, \{t(\varphi)\}_{\varphi \in \Psi}, \{t(\rho)\}_{\rho \in P} >$ *will be called the type of the language.*

The set of all proper expressions of a first order language consists of the set of terms and the set of formulas. Using terms, we can define new functions. Similarly, formulas can be considered as definitions of new predicates.

Definition 2.3.3 *The set of terms T over the alphabet $Alph$ is the smallest set which contains the set of individual variables V and such that, if φ is a functor of type $(i_1 \times ... \times i_n \rightarrow i_{n+1})$ and $\tau_1, ..., \tau_n$ are terms of types $i_1, ..., i_n$ respectively, then $\varphi(\tau_1, ..., \tau_n)$, is a term of type i_{n+1}.*

Definition 2.3.4 *The set of formulas F over the alphabet $\mathcal{A}$ is the smallest set of expressions which contains all propositional variables and all expressions of the form $\rho(\tau_1, ..., \tau_m)$, (called elementary formulas) where ρ is a predicate of type $(i_1 \times ... \times i_m \rightarrow 0)$, and $\tau_1, ..., \tau_m$ are arbitrary terms of types i_1,...,i_m, respectively, and such that*

- if $\alpha, \beta \in F$ (i.e. are formulas), then the expressions of the form $(\alpha \wedge \beta)$, $(\alpha \vee \beta)$, $\neg\alpha$, $(\alpha \Longrightarrow \beta)$ are also formulas,
- if α is a formula and x is an individual variable, then the expressions of the form $(\forall x)\alpha$, $(\exists x)\alpha$ are also formulas.

Formulas which do not contain any quantifier symbol will be known as open formulas.

Remark 2.3.5 *Throughout this book we use τ (with indices if necessary) to denote terms, α, β or δ to denote formulas and γ to denote open formulas.*

Example 2.3.6 *If $x, y, z \in V$ and $+, *$ are two-argument functors and $>$ is a two-argument predicate in a one-sorted language L, then*

$*(+(x, y), z) + (*(x, z), *(y, z))$
are terms and
$(\forall y)(\exists x) > (*(+(x, y), z),$
$+(*(x, z), *(y, z)))$
$\neg(> (*(x, x), *(y, y)) \Longrightarrow\ > (x, y))$

are formulas in this language.

In the usual notation for two-argument relations and functions, the above expressions have the following, much more readable, form

$(x + y) * z(x * z + y * z)$

$(\forall y)(\exists x)(x + y) * z \ > \ (x * z + y * z)$

$\neg(x * x > y * y \Longrightarrow x > y)$

From now on, we shall use this more readable notation to represent two-argument functions or relations.

For any formula of the form $(\exists x)\alpha$ (or $(\forall x)\alpha$), α is known as the scope of existential (or universal) quantifier. Any occurrence of the variable x in the expression α is said to be bounded by the existential (universal) quantifier. If an individual variable y occurs in the formula α and is not bounded by any quantifier, then we shall say that y is a free variable in α. Writing $\alpha(x_1, ..., x_n)$ we stress that variables $x_1, ..., x_n$ are free in the formula α. The set of all free variables that occur in a formula α will usually be denoted by $FV(\alpha)$. The set of all variables that occur in a term τ or in a formula α will be denoted by $V(\tau)$ and $V(\alpha)$, respectively.

Example 2.3.7 *Consider the first-order language over the alphabet described in example 2.3.1 and the formula α of the form $(\beta \vee \delta)$, where*
$\beta = (\forall x)(\exists y)((x + y) * z > y)$, $\delta = (x \ > \ y(x + y))$.
The variable z is free in β and variables x, y are bounded in β. All variables that occur in the formula δ are free variables. In the formula α there are three free variables x, y, z and two bounded variables x, y. Notice, that the same variable may occur in one formula as free and as bounded:

$$\underbrace{((\forall x)(\exists y)((x + y) * z \ > \ y)}_{\text{bounded occurrences}} \vee \underbrace{(x \ > \ y * (x + y)))}_{\text{free occurrences}}$$

We close this formal definition of first-order languages with the following definition.

Definition 2.3.8 *By the first-order language over an alphabet $\mathcal{A}$ we shall mean a system $\mathcal{L} =< \mathcal{A}, T; F >$, where T is the set of all terms and F is the set of all formulas over the alphabet $\mathcal{A}$.*

Remark 2.3.9 *Different alphabets determine different sets of terms and formulas, and hence, different first-order languages. The above definition, thus, describes the class of first-order languages.*

We use the following denotations: **true** for the formula (p $\vee$ $\neg$p) and **false** for the formula $(p \wedge \neg p)$, where p is a propositional variable of the

language under consideration. If the predicate = occurs in the language $\mathcal{L}$, then we stress this fact by writing $\mathcal{L}_{=}$.

Terms and formulas are only formal expressions. Meaning will be given to them when we interpret all their symbols. The interpretation of all proper expressions of the language is called its semantics.

Let $\mathcal{L}$ be a first-order language of type $< tt, \{t(\varphi)\}_{\varphi\in\Psi}, \{t(\rho)\}_{\rho\in P} >$, and let A be a non-empty set which is set-theoretic union of disjoint sets A_i for $1 \leq i \leq tt$. For each functor $\varphi \in \Psi$ of type $t(\varphi) = (i_1 \times ... \times i_n \rightarrow i_{n+1})$ let φ_A denote n-argument partial function in A,

$$\varphi_A : A_{i_1} \times ... \times A_{i_n} \rightarrow A_{i_{(n+1)}}$$

and for each predicate ρ of type $t(\rho) = (i_1 \times ... \times i_m)$, let ρ_A denote an arbitrary m-argument relation on A,

$$\rho_A \subseteq \ A_{i_1} \times ... \times A_{i_M}$$

In this way we have fixed an algebraic structure A,

$A = \ < A, (\varphi_A)_{\varphi\in\Psi}; (\rho_A)_{\rho\in P} >$,

of the same type as the language $\mathcal{L}$. The function φ_A and the relation ρ_A will be known as the interpretation of the functor φ and of the predicate ρ in the structure A. When considering a first-order language $\mathcal{L}_{=}$ we will usually assume that the identity relation on A is the interpretation of the predicate =. We call such an algebraic system A a data structure of the language $\mathcal{L}$.

Example 2.3.10 *Let φ, ψ be binary functors in the language $\mathcal{L}$, and x, y, z individual variables. Let A be a data structure for $\mathcal{L}$, with the set of real numbers as universe and with operations + and $*$ as interpretations of the symbols φ, ψ. With this, the term $\psi(\varphi(x,y),z)$ can be understood as a three-argument function $\varphi_A(\psi_A(x, y,), z)$ defined on the set of real numbers, which to any triple of real numbers a_1, a_2, a_3 associates the real number which is the value of arithmetic expression $(a_1 + a_2) * a_3$.*

Definition 2.3.11 *By a* valuation *in the data structure A for the language L we mean any function*
v: V∪Vo $\rightarrow$ A∪Bo
such that v(x)∈A_i for x∈V_i, 1≤i≤tt and v(q)∈B_0 for q∈V_0, i.e. valuation is a function which assigns to each individual variable an element of the same type, and to each propositional variable an element of the Boolean algebra B_0.

Let us now generalize the observations from example 2.3.10. Any term τ with variables $x_1, ..., x_n$, determines for a given data structure an n-argument partial function which depends only on the values of variables $x_1, ..., x_n$.

Let us denote this function by τ_A. For simplicity, we shall consider τ_A as a function defined on the set $W(A)$ of all valuations in the structure A. For every valuation v, τ_A eventually determines an element $\tau_A(v)$ of A defined recursively as follows:

$x_A(v) = v(x)$ for $x \in V \cup V_0$

$$\varphi(\tau_1, ..., \tau_n)_A(v) = \begin{cases} \varphi_A(\tau_{1A}(v), ..., \tau_{nA}(v)) \text{ if } \tau_{1A}(v), ..., \tau_{nA}(v) \\ \qquad\qquad\qquad\qquad\qquad\qquad \text{and } \varphi_A(\tau_{1A}(v), .., \tau_{nA}(v) \text{ are defined} \\ \text{undefined otherwise} \end{cases}$$

Let us review

- the value of a simple term of the form x is given by its valuation in the structure A,

- the value of a term $\tau = \varphi(\tau_1, ..., \tau_n)$ is the element $\varphi_A(a_1, ..., a_n)$, if the functions τ_{1A},...,τ_{nA} have defined values in the valuation v, equal respectively to $a_1, ..., a_n$, and $(a_1, ..., a_n)$ belongs to the domain of the partial function φ_A.

Similarly we can determine the meaning of formulas. The formal definition is preceded by an example.

Example 2.3.12 *Let $\mathcal{L}$ and A be the language and data structure defined in example 2.3.10. Assume moreover, that we have a two-argument predicate ρ interpreted in A as $<$. Consider the formula*
$(\rho(\psi(x, x), \psi(y, y)) \implies \rho(x, y))$.
Interpreting it in the structure A, we obtain the expression $(x * x < y * y \implies x < y)$, which can be understood as a definition
- of a binary relation which holds for real numbers a, b iff $a < b$ or $|a| \geq |b|$,
or
- of a two-argument function $f(x, y)$, which takes on the value $\mathbf{1} \in B_0$ iff $x < y$ or $|x| \geq |y|$.

Each formula α determines a total function α_A on a data structure A, which to every valuation v assigns an element of the Boolean algebra Bo. For a given argument v the value of the formula a, at the valuation v in the data structure A, will be denoted by $\alpha_A(v)$ and is defined with respect to the structure of a as follows:

$q_A(v) = v(q)$ for $q \in V_0$,

$$\rho(\tau_1, ..., \tau_n)_A(v) = \begin{cases} \rho_A(\tau_{1A}(v), ..., \tau_{nA}(v)) & \text{if the values } \tau_{1A}(v), ..., \tau_{nA}(v) \text{ are defined} \\ \mathbf{0} \text{ otherwise} \end{cases}$$

$(\alpha \wedge \beta)_A(v) = \alpha_A(v) \cap \beta_A(v)$

$(\alpha \vee \beta)_A(v) = \alpha_A(v) \cup \beta_A(v)$
$(\alpha \Longrightarrow \beta)_A(v) = \alpha_A(v) \rightarrow \beta_A(v)$
$(\neg\alpha)_A(v) = -\alpha_A(v)$
$((\forall x)\alpha)_A(v) = \mathbf{1}$ iff for all $a \in A$, $\alpha_A(v^x_a) = 1$
$((\exists x)\alpha)_A(v) = \mathbf{1}$ iff there exists an $a \in A$ such that $\alpha_A(v^x_a) = 1$,
where $v^x_a(z) = v(z)$ for $z \neq x$ and $v^x_a(x) = a$.

(in other words, the valuation v^x_a is identical to the valuation v except that x takes on the value a)

The universal quantifier is interpreted as the greatest lower bound in the boolean algebra B_0,

$((\forall x)a)_A(v) = inf\{\alpha_A(v^x_a) : a \in A\}$,

and the existential quantifier as the least upper bound in B_0,

$((\exists x)a)_A(v) = sup\{\alpha_A(v^x_a) : a \in A\}$.

Definition 2.3.13 *We say that a formula α is* satisfied *by the valuation v in the structure A if and only if $\alpha_A(v) = \mathbf{1}$, and we write this symbolically $A, v \models \alpha$.*
If a formula α is satisfied in the structure A by every valuation, then we say that α is valid *in A, and we write $A \models \alpha$. If the formula α is valid in every structure, then we say that α is a* tautology, *which we write as $\models \alpha$.*

Remark 2.3.14 *(1) Every formula is a finite expression. Thus its value depends only on the finite number of values of variables which occur in the formula.*
(2) If we consider a first-order language $\mathcal{L}_=$ (i.e. the binary predicate $=$ belongs to our language) then for any valuation v in any data structure for $\mathcal{L}_=$, the following holds
$A, v \models \tau = \tau$ iff $v \in Dom(\tau)$.
In other words, $\tau = \tau$ is valid in A only for those valuations for which the value of the term τ in the structure A is defined.

Example 2.3.15 *(1) The formula $(\forall y)x \leq y$ is satisfied in the structure of natural numbers with the usual interpretation of predicate $\leq$ by the valuation v in which $v(x) = 0$. It is not valid in this structure, since nor all valuations satisfy it.*
(2) The formula $(x \leq y \vee y \leq x)$ is valid in every data structure in which the symbol $\leq$ is interpreted as a linear ordering.
(3) The formula $(x \leq y \vee \neg x \leq y)$ is a tautology since it is valid independently of the values of the variables and of the interpretation of the relation

symbol $\leq$.

Lemma 2.3.16 *The following formulas in the first-order language* $\mathcal{L}$ *are valid in any data structure for* $\mathcal{L}$*. They are called laws of classical logic.*

$(\alpha \vee \neg\alpha)$	*law of the exclusive middle*
$(\neg(\neg\alpha) \Leftrightarrow \alpha)$	*law of double negation*
$((\alpha \Longrightarrow \beta) \Longrightarrow (\neg\beta \Longrightarrow \neg\alpha))$	*transposition law*
$((\forall x)\neg\alpha \Leftrightarrow \neg(\exists x)\alpha)$	*de Morgan's law*

Definition 2.3.17 *Let* Z *be any set of formulas in the language* $\mathcal{L}$*. We say that a data structure* A *is a* model *of the set* Z*, written symbolically as* $A \models Z$*, if every formula* α *from* Z *is valid in* A*,* $A \models \alpha$*.*

Example 2.3.18 *Consider the first-order language* $\mathcal{L}$*, in which there are individual variables of two sorts* E *and* S*, and there are functors: top, pop and push of types*$(S \to E)$*,* $(S \to S)$*,* $(E \times S \to S)$ *and predicates: empty,* $=_S$ *of types* (S) *and* $(S \times S)$*, respectively.*
The standard stack structure defined in section 2.2 is a model of the following set of formulas
$(\forall e)(\forall s)\neg empty(push(e, s))$
$(\forall e)(\forall s)e = top(push(e, s))$
$(\forall e)(\forall s)s = pop(push(e, s))$
$(\forall e)(\forall s)(\neg empty(s) \Longrightarrow s = push(top(s), pop(s)))$.
Verification is easy.

Example 2.3.19 *Let* $\alpha(x)$ *denote a formula in the language* $\mathcal{L}_=$ *with one free variable* x*. Let* $\alpha(x/\tau)$ *be the formula obtained from* α *by simultaneous replacement of all occurrences of* x *by* τ*. We shall prove that each data structure for* $\mathcal{L}_=$ *is a model for all formulas of the form*

$$((\forall x)\alpha(x) \Longrightarrow (\tau = \tau \Longrightarrow \alpha(x/\tau)),$$

where τ *is any term of the same type as the variable* x*.*
Let us fix a data structure A *for* $\mathcal{L}_=$*. Let* v *be an arbitrary fixed valuation in* A*, and* τ *be any term of the same type as* x*. Assume that*$A, v \models (\forall x)\alpha(x)$ *and* $A, v \models t = t$*.*
The second condition implies that the value of term τ *is defined at the valuation* v*. By the definition of the semantics of the language, and by the first condition we have* $A, v_a x \models \alpha(x)$ *for* $\alpha = \tau_A(v)$*. Thus* $A, v \models \alpha(x/t)$*.*

2.4 Expressibility problems in the first-order language

Definition 2.4.1 ***Definition 2.4.2*** *We shall say that a property p is* expressible *in the language $\mathcal{L}$ iff there exists a formula a in the language $\mathcal{L}$ such that, for every data structure A ,*

$$A \text{ has the property } p \text{ iff } A \models \alpha.$$

Example 2.4.3 *The property "to be an Abelian group" is expressible in the language $\mathcal{L}_=$, in which φ is a two-argument functor and 0 is a constant. Let α denote the conjunction of the following formulas:*
$(\forall x)(\exists y)\ \varphi(x,y) = 0$
$(\forall x,y)\ \varphi(x,y) = \varphi(y,x)$
$(\forall x,y)\ \varphi(x,\varphi(y,z)) = \varphi(\varphi(x,y),z)$
$(\forall x)\ \varphi(x,0) = x$
and let β denote the conjunction of the formulas:
$(\forall x)(x = x)$
$(\forall x,y)(x = y \implies y = x)$
$(\forall x,y,z)((x = y \wedge y = z) \implies x = z)$
$(\forall x,y)(x = y \implies \varphi(x) = \varphi(y))$
Then a structure A is an Abelian group iff $A \models (\alpha \wedge \beta)$.

Example 2.4.4 *Let L be an arbitrary first-order language in which R is the set of predicates and F the set of functors. Let eq∈R. Consider the following set of formulas Axeq in the language $\mathcal{L}$:*
$(\forall x)eq(x,x)$
$(\forall x,y)(eq(x,y) \implies eq(y,x))$
$(\forall x,y,z)((eq(x,y) \wedge eq(y,z)) \implies eq(x,z))$
$(\forall x,y)((eq(\varphi(\mathbf{x}),\varphi(\mathbf{x})) \wedge eq(\varphi(\mathbf{y}),\varphi(\mathbf{y})) \wedge eq(x_1,y_1) \wedge ... \wedge eq(x_n,y_n)) \implies eq(\varphi(\mathbf{x}),\varphi(\mathbf{y})))$
for each functor φ, where n is the number of arguments of φ.
$(\forall x,y)((eq(x1,y1) \wedge ... \wedge eq(xm,ym)) \implies r(x) \Leftrightarrow r(y))$
for each predicate r∈R, where m is the number of arguments of r. Here $\boldsymbol{x}$ denotes the vector (x_1,...,x_n), and $\boldsymbol{y}$ the vector (y_1,...,y_n).In every data structure A for the language $\mathcal{L}$ the following property holds:
$A \models$ Axeq iff eqA is a congruence in A.
The first three conditions guarantee that eqA is an equivalence relation in A, the remaining ones express the extensionality property of the relation eqA. Henceforth the set of formulas Axeq will be known as the axioms of equality.

Example 2.4.5 *(a) Let α denote the following formula in a certain language $\mathcal{L}_=$ with equality*
$(\exists x_1)..(\exists x_n)(\forall x)(x = x_1 \vee ... \vee x = x_n)$.
The formula α has the following property: for every data structure A, $A \models \alpha$ iff the universe of the data structure A contains at most n elements.
Indeed, consider a data structure A, such that its universe has cardinality greater than n. Let v be an arbitrary valuation in A and let a be an element of A, such that $v(x_i) \neq a$ for $1 \leq i \leq n$. Then
$A, v_a^x \models (x \neq x_1 \wedge ... \wedge x \neq x_n)$
and consequently
non $A, v \models (\forall x)(x = x_1 \vee ... \vee x = x_n)$.
The valuation v was an arbitrary valuation, hence
non $A, v \models (\exists x_1)...(\exists x_n)(\forall x)(x = x_1 \vee ... \vee x = x_n)$,i.e. non $A \models \alpha$. Conversely, if non $A \models \alpha$, then (since α is a closed formula) for every valuation v, non $A, v \models \alpha$. Therefore non $A, v \models (\forall x)(x = x_1 \vee ... \vee x = x_n)$ for every v.In accordance with the definition of quantifiers for every valuation v there exists an element a such that non $A, v_a^x \models (x = x_1 \vee ... \vee x = x_n)$. Finally, for every sequence of values $v(x_1), ..., v(x_n)$ there exists an element a such that $a \neq v(x_i)$, for $i \leq n$, i.e. $card(A) > n$.
(b) Consider the formula β of the form:
$(\exists x_1)...(\exists x_n)(x_1 \neq x_2 \wedge x_1 \neq x_3 \wedge ... \wedge x_1 \neq x_n \wedge x_2 \neq x_3 \wedge x_2 \neq x_n ... \wedge x_{n-1} \neq x_n)$.
Formula β is valid in those and only those data structures in which the universe has at least n elements. Hence the formula β expresses the property "the universe of the system has at least n elements". The proof of this fact is left to the reader.

Putting together the two formulas we can define the class of data structures that have exactly n elements since the conjunction $(\alpha \wedge \beta)$ expresses this property.

Theorem 2.4.6 *Let A and B be data structures for the language $\mathcal{L}$. Let h be an isomorphism mapping structure A onto structure B, $h : A \to B$. Consider a term τ and a valuation v in A, then τ_A is a mapping defined for the valuation v iff B is defined for the valuation $h(v)$, and when $\tau_A(v)$, $\tau_B(h(v))$ are defined, we have*
$h(\tau_A(v)) = \tau_B(h(v))$ *(2.1)*
Moreover, for every formula α and for every valuation v
$A, v \models \alpha \Leftrightarrow B, (h(v)) \models \alpha$ *(2.2)*

Proof. The proof of the theorem proceeds by induction with respect to the structure of terms and formulas.

If τ is an individual variable x, then
$h(\tau_A(v)) = h(v(x)) = (h(v))(x) = x_B(h(v)) = \tau_B(h(v))$.
If τ is of the form $\varphi(\tau_1, ..., \tau_n)$ and if the property (2.1) is satisfied for terms $\tau_1, ..., \tau_n$, then from the definition of an isomorphism and from the inductive hypothesis
$h(\varphi(\tau_1, ..., \tau_n)_A(v)) = h(\varphi_A(\tau_{1A}(v), ..., \tau_{nA}(v))) = \varphi_B(h(\tau_{1A}(v)), ..., h(\tau_{nA}(v))) =$
$\varphi_B(\tau_{1B}(h(v)), ..., \tau_{nB}(h(v))) = \varphi(\tau_1, ..., \tau_n)_B(h(v))$.
Thus, by induction, property (2.1) holds for all terms of the language $\mathcal{L}$.
Consider now the formulas of $\mathcal{L}$. Let α be a formula of the form $\rho(\tau_1, ..., \tau_m)$.
$A, v \models \alpha$ iff $A, v \models \rho(\tau_1, ..., \tau_m)$ iff τ_{iA} is defined for all $i \leq m$ and $(\tau_{1A}(v), ..., \tau_{mA}(v)) \in \rho_A$.
by the definition of an isomorphism and from the above remarks,
$A, v \models \alpha$ iff $(h(\tau_{1A}(v)), ..., h(\tau_{mA}(v))) \in \rho_B$ iff
$(\tau_{1B}(h(v)), ..., \tau_{mB}(h(v))) \in \rho_B$ iff $B, h(v) \models \rho(\tau_1, ..., \tau_m)$.
Assume that property (2.2) is valid for formulas β, β_1, β_2 and consider a formula of the form $(\beta_1 \wedge \beta_2)$. We then have
$A, v \models (\beta_1 \wedge \beta_2)$ iff $A, v \models \beta_1$ and $A, v \models \beta_2$ iff
$B, h(v) \models \beta_1$ and $B, h(v) \models \beta_2$ iff $B, h(v) \models (\beta_1 \wedge \beta_2)$.
Let α be a formula of the form $(\exists x)\beta(x)$. If a is a variable of the same type as x, then $b = h(a)$ has the same type as x and, moreover,
$A, v \models (\exists x)\beta(x)$ iff for certain a, $A, v^x_a \models \beta(x)$ iff
for certain $a \in A$, $B, h(v^x_a) \models \beta(x)$ iff
for certain $a \in A$, $B, h(v)^x_{h(a)} \models \beta(x)$ iff
for certain $b \in B$, $B, h(v)^x_b \models \beta(x)$ iff $B, h(v) \models (\exists x)\beta(x)$.
The proof of property (2.2) in the remaining cases has a similar form. ■

By the above theorem, if $A \models \alpha$ and if B is a structure isomorphic to A, then $B \models \alpha$. In other words, if the structure A is a model for a certain set of formulas Z, then every structure isomorphic to A is also a model of Z. Hence it is impossible to uniquely characterize a structure. We can at most define a class of isomorphic data structures. We shall see later that even this goal is not always attainable.

Example 2.4.7 *Consider the structure of the natural numbers*
$N =< N, suc, 0; =>$
The property "to be a natural number" or "to be the structure of the natural numbers" cannot be expressed in the language of first-order logic with equality. Obviously we can characterize the constant 0 *and the successor operation by means of the following formula* α*:*
$(\forall x)(suc(x) = suc(x)) \wedge (\neg suc(x) = 0) \wedge (\forall x, y)(suc(x) = suc(y) \implies x = y)$.
However, from the fact that $N \models \alpha$ *it does not follow that the universe of a model* N *consists of the natural numbers nor that* N *is isomorphic to the*

structure of the natural numbers. The validity of the formula α in a structure N guarantees the following properties of the structure N:
(1) the interpretation of the functor suc is a total function,
(2) the function does not assume the value corresponding to the constant 0,
(3) the function is a one-to-one function.
It turns out that even if we add the so called induction scheme i.e. all formulas of the form
$\{\gamma(x/0) \wedge [\forall x)(\gamma(x) \implies \gamma(x/suc(x))]\} \implies (\forall x)\gamma(x)$
as additional axioms then there still exist models which we would not accept as the standard concept of natural numbers. The axioms given above, the so-called Peano axioms, are not enough to discriminate non-standard models. It turns out that there exist non-enumerable models for the Peano axioms (cf. Grzegorczyk [21]).

Unfortunately, the majority of properties that are of interest to programmers cannot be expressed in the language of first-order logic. Among such properties, we find the following:

(1) to be a finite set,
(2) to be a well founded ordering relation,
(3) to be a natural number (in its standard meaning),
(4) to be a (finite) stack,
(5) to be a (finite) queue,
(6) to be a (finite) binary tree,
and many others.

Pozytywna informacja jaka plynie z twierdzenia 2.4.1 pozwala udowodnic istnienie komputera. Obliczenia prowadzone w systemie binarnym dadza wartosci ktore po przetlumaczeniu na system dziesietny sa rowne wartosciom jakie otrzymalibysmy wykonuujac obliczenia w systemie dziesietnym. Mowimy ze diagram

TU DIAGRAM

jest przemienny. Majac swobode wyboru systemu: binarny czy dziesietny, wybieramy tanszy w realizacji, ... w eksploatacji.

Nie sa wyrazalne tez wlasnosci programow takie jak
- program P zakonczy obliczenia
- program P zakonczy obliczenia i wyniki spelnia warunek a,
W nastepnym rozdziale zaproponujemy wyjscie z tej sytuacji.

Chapter 3

Deterministic Iterative Programs

In this chapter we discuss the simplest class of programs which is rich enough to define every computable function. We call it the class of iterative programs or the class of deterministic while-programs and denote by it P. For this class we shall present some logic-oriented methods which allow us to analyze programs and data structures.

Our considerations are based on a language of first-order logic FOL with equality. This language is, however, too weak to describe more complicated properties of programs, e.g. the holting property is not expressible in the language. Consequently, we shall extend the language by allowing expressions which contain programs as well as classical terms and formulas. These expressions, called algorithmic formulas, allow us to express all important properties of programs as well as properties of data structures which are not expressible in a language of first-order logic.

3.1 Programs

From the very beginning the development of computers was accompanied by the rapid development of different programming languages concepts. Some of them are of universal character, whereas others serve as tools for highly specialized problems. They often differ, not only from a formal point of view, but also, more essentially, by the tools (instructions) offered to programmers. Moreover, this process is continuing and will continue. Nonethelees, new programming languages are appearing with new programming constructs whose aim is to make the process of program-construction easier, e.g. processes, coroutines, signals etc. Within this large population of programming lan-

guages we can distinguish several common properties. For example, in the majority of languages, a program is understood as a finite sequence of instructions. Combining simple instructions by programming constructs we obtain more complicated instructions etc.

In this section we present the class of deterministic iterative programs which abstracts from some details, such as variable declarations, since they have no influence on the process of verification.

Let $\mathcal{L}$ be a fixed FOL-language. Terms (cf. def.2.3.3) and formulas (cf. def. 2.3.4) of this language will serve as arithmetic and boolean expressions in the programs defined below.

Definition 3.1.1 *By an* assignment instruction*, we mean any expression of the form*

$x := \tau$ *or* $q := \gamma$,

where x *is an individual variable,* τ *is a term of the same type as the variable* x*,* q *is a propositional variable, and* γ *is an open formula.*

Example 3.1.2 *If* x*,* y *are individual variables of some one-sorted language* $\mathcal{L}$ *and* $+$*,* $-$ *are symbols representing two-argument operations and* $\leq$ *,* $\geq$ *are symbols representing two-argument relations then*

$$z := (x + y) - z \text{ and } q := (x \leq y \implies x \geq (y - z))$$

are assignments instructions.

Assignments instructions will be called *atomic* or *elementary* programs.

Definition 3.1.3 *The set* P *of all deterministic iterative programs is the smallest set of expressions closed with respect to the following formation rules*

(1) each assignment instruction in the language $\mathcal{L}$ *is a program of the class* P*,*

(2) if γ *is an open formula and* K*,* $M \in P$*, then the expression* ***if*** γ ***then*** K ***else*** M ***fi*** *is a program of the the class* P*,*

(3) if γ *is an open formula and* $M \in P$*, then the expression* ***while*** γ ***do*** M ***od*** *is a program of class* P*,*

(4) if $K, M \in P$*, then the expression* ***begin*** K; M ***end*** *is a program of the class* P*.*

It is natural to illustrate programs as flow-diagrams. Each deterministic program can be represented as a graph with one input and one output vertex. Each edge is labeled by a test (i.e. open formula) or assignment instruction. The simplest diagram is illustrated on Fig. 3.1.

Fig. 3.1

If the diagrams of K and M are given (cf. Fig. 3.2), then identifying the output edge of the first diagram with the input edge of the second diagram we obtain the diagram of the composed program begin K; M end (cf. Fig.3.3).

Fig. 3.2

|

Fig. 3.3

Analogously, if diagrams of programs K and M are as in figure 3.2, then the graphs given in Fig.3.4 and Fig.3.5 are illustrations of the programs

"if g then K else M fi" and "while g do K of ",

respectively.

Fig. 3.4

Fig. 3.5

To end the section, we can observe that the set of programs P creates an algebra with the two-argument operations of composition $\circ$ and branching $\veebar_\gamma$ and the one-argument operation $*_\gamma$, for $\gamma \in F_o$, defined as follows

$K \circ M \stackrel{df}{=}$ **begin** $K; M$ **end**

$K \veebar_\gamma M \stackrel{df}{=}$ **if** γ **then** K **else** M **fi**

$*_\gamma M \stackrel{df}{=}$ **while** γ **do** M **od**

The set of all assignments instructions is the set of generators of this algebra. By this we mean that every program from the set P can be constructed from assignments by means of the program operations $\circ$, $\veebar_\gamma$, $*_\gamma$.

The algebraic character of the set P stresses the modular character of the class of programs.

Henceforth we will use the following notations and abbreviations:

- instead of i iterations of the program K

begin K;K;...;K end

SYMBOL 59 \f "Fences"

i times

we shall write K^i,

- for an arbitrary variable x, instead of the instruction

if g then K else x :=x fi

we shall write for short if g then K fi.

3.2 Semantics

In order to define the semantics of a programming language one should first fix an interpretation of the language which is used to define programs. This means that the relational system or data structure corresponding to the first order language FOL must be given. The interpretation of terms and formulas which occur in programs is then defined as in section 2.3. The second important part of the semantics is the interpretation of programming constructs. Using the algebraic structure of the class P we can define the interpretation of the more complicated instructions making up from the interpretation of simpler parts.

Although there is no common concensus on what the basic programming language constructs are and what are their meanings, there is a common conception which consists in associating with a program a binary relation which describes the connections between the input and output states of the computer memory. We shall call such a relation the input-output relation. Since, in our approach, the memory state is just a valuation of variables (cf. def.2.3.6), this relation is defined on the set of all valuations. The state of the computer memory before the execution of a program is called the initial valuation and the state of the computer memory after execution of a program - the output valuation.

We shall define the input-output relation with respect to the structure of the program. The method presented here is called operational semantics of programming languages.

Let M be a program and let $\mathbb{A}$ be a data structure of the language L. If valuations v, v' are in the input-output relation determined by the program M in data structure $\mathbb{A}$, then we write this as
$v \xrightarrow{M_{\mathbb{A}}} v'$
or without the index $\mathbb{A}$ when the structure is fixed.

The inductive definition of the input-output relation is given below:
$v \xrightarrow{(x:=w)_{\mathbb{A}}} v_1$ iff $v \in Dom(w_{\mathbb{A}}), v_1(x) = \omega_{\mathbb{A}}(v)$ and $v_1(z) = v(z)$ for $z \neq x$
$v \xrightarrow{beginK;Mend_{\mathbb{A}}} v_1$ iff $v \xrightarrow{K_{\mathbb{A}}} v'$ and $v' \xrightarrow{M_{\mathbb{A}}} v_1$
$v \xrightarrow{if\ \gamma\ then\ K\ else\ M\ fi_{\mathbb{A}}} v_1 \text{iff} \mathbb{A}, v \models \gamma \text{and} v \xrightarrow{K_{\mathbb{A}}} v_1 \text{or} \mathbb{A}, v \models \neg\gamma \text{and} v \xrightarrow{M_{\mathbb{A}}} v_1$
$v \xrightarrow{while\ \gamma\ do\ M\ od_{\mathbb{A}}} v_1$ iff either $\mathbb{A}, v \models \neg\gamma$ and $v_1 = v$ or $\mathbb{A}, v \models \gamma$ and $(\exists v')v \xrightarrow{M_{\mathbb{A}}} v'$ and $v' \xrightarrow{while\ \gamma\ do\ M\ od_{\mathbb{A}}} v_1$
for all valuations v, v' in the data structure $\mathbb{A}$.

The way in which we have determined the meaning of the programs enables us to observe the process by which the initial data are transformed into the result. We shall call this process the computation of a program. In

the sequel, we define formally the notion of computation and two auxiliary notions: configuration and the direct succesorship relation.

DEFINITION 3.2.1

Definition 3.2.1 *By a* configuration *in a fixed data structure* $\mathbb{A}$ *we mean an ordered pair* $< v, K_1; ...; K_n >$, *where* v *is a valuation in* $\mathbb{A}$, *and* $K_1, ..., K_n$ *is a finite list of programs to be executed consecutively. To abbreviate the notation we will often omit the name of the structure.* □

On the set of all configurations in a fixed data structure A, we define the direct succesorship relation which will later allow us to define the notion of computation.

Definition 3.2.2 *By a direct successorship relation we mean a binary relation denoted by* $\underset{\mathbb{A}}{\longrightarrow}$ *(we shall omit the sign* $\mathbb{A}$ *if this does not lead to confusion) defined on the set of all configurations in* $\mathbb{A}$ *in the following way:*
$<v,\ z:=\omega;K_1;...;K_n > \underset{\mathbb{A}}{\longrightarrow} <v',\ K_1;...;K_n >$ *if the value of the expression* ω *is defined and* $v'(z)=\omega_{\mathbb{A}}(v)$, $v'(y)=v(y)$ *for* $y \neq z$.
$<v,$*if* γ *then* K *else* M *fi*$;K_1;...;K_n > \underset{\mathbb{A}}{\longrightarrow} <v,\ K;K_1;...;K_n >$ *when* $A,v \models \gamma$
$<v,$*if* γ *then* K *else* M *fi*$;K_1;...;K_n > \underset{\mathbb{A}}{\longrightarrow} <v,\ M;K_1;...;K_n >$ *when* $A,v \models \neg\gamma$
$<v,$*begin* $K;M$ *end*$;K_1;...;K_n > \underset{\mathbb{A}}{\longrightarrow} <v,\ K;M;K_1;...;K_n >$
$<v,$*while* γ *do* M *od*$;K_1;...;K_n > \underset{\mathbb{A}}{\longrightarrow} <v,\ K_1;...;K_n >$ *when* $A,v \models \neg\gamma$
$<v,$*while* γ *do* M *od*$;K_1;...;K_n > \underset{\mathbb{A}}{\longrightarrow} <v,M;$*while* γ *do_M_od*$;K_1;...;K_n >$ *when* $A,v \models \gamma$.

Observe the following

Remark 3.2.3 *For a given configuration* $c=<v,K_1;...;K_n>$*there exist at most one configuration* c' *such that* $c \underset{\mathbb{A}}{\longrightarrow} c'$.

Definition 3.2.4 *By the computation of a program* M *in a data structure* $\mathbb{A}$ *at the initial valuation* v, *we understand a sequence of configurations* $\mathbf{Q} = \{Q_0, Q_1, ..., Q_i...\}$ *such that*
(1) $Q_0 =< v, M >$
(2) if there exists a configuration Q' *such that* $Q_i \underset{\mathbb{A}}{\longrightarrow} Q'$, *then the* $i+1$*-th element of* $\mathbf{Q}$ *is defined and* $Q_{i+1} = Q'$,
(3) otherwise Q_i *is the final element of the computation.*

Example 3.2.5 *Consider the following program K in the language L*
begin
y:=0;
while y≠z do M od
end,
where
M=begin y:=y+1; x:=1+(1/(1+x)) end.
Let <R,0,1,+,/> be a data structure for the one-sorted language L such that R is the set of real numbers, and 0, 1, +, / have their usual meanings.
1) For initial data v such that v(x), v(z)ÎN the program K has a finite computation. If v(x)=1, v(z)=2, then the computation of K is as follows (the initial value of the variable y has no effect on the computation):

$$\left\langle \frac{x \quad y \quad z}{1 \quad - \quad 2}, K \right\rangle$$

$$\left\langle \frac{x \quad y \quad z}{1 \quad - \quad 2}, y : 0;\ \mathit{while}\ y \neq z\ \mathit{do}\ M\ \mathit{od} \right\rangle$$

$$\left\langle \frac{x \quad y \quad z}{1 \quad 0 \quad 2}, \mathit{while}\ y \neq z\ \mathit{do}\ M\ \mathit{od} \right\rangle$$

$$\left\langle \frac{x \quad y \quad z}{1 \quad 0 \quad 2}, M;\ \mathit{while}\ y \neq z\ \mathit{do}\ M\ \mathit{od} \right\rangle$$

$$\left\langle \frac{x \quad y \quad z}{1 \quad 0 \quad 2}, y := y + 1;\ x := 1 + (1/(1 + x));\ \mathit{while}\ y \neq z\ \mathit{do}\ M\ \mathit{od} \right\rangle$$

$$\left\langle \frac{x \quad y \quad z}{1 \quad 1 \quad 2}, x := 1 + (1/(1 + x));\ \mathit{while}\ y \neq z\ \mathit{do}\ M\ \mathit{od} \right\rangle$$

$$\left\langle \frac{x \quad y \quad z}{\frac{3}{2} \quad 1 \quad 2}, \mathit{while}\ y \neq z\ \mathit{do}\ M\ \mathit{od} \right\rangle$$

$$\left\langle \frac{x \quad y \quad z}{\frac{3}{2} \quad 1 \quad 2}, M;\ \mathit{while}\ y \neq z\ \mathit{do}\ M\ \mathit{od} \right\rangle$$

$$\left\langle \frac{x \quad y \quad z}{\frac{3}{2} \quad 1 \quad 2}, y := y + 1;\ x := 1 + (1/(1 + x));\ \mathit{while}\ y \neq z\ \mathit{do}\ M\ \mathit{od} \right\rangle$$

$$\left\langle \frac{x \quad y \quad z}{\frac{3}{2} \quad 2 \quad 2}, x := 1 + (1/(1 + x));\ \mathit{while}\ y \neq z\ \mathit{do}\ M\ \mathit{od} \right\rangle$$

$$\left\langle \frac{x \quad y \quad z}{\frac{3}{2} \quad 2 \quad 2}, \mathit{while}\ y \neq z\ \mathit{do}\ M\ \mathit{od} \right\rangle$$

$$\left\langle \frac{x \quad y \quad z}{\frac{3}{2} \quad 2 \quad 2}, \emptyset \right\rangle$$

2) Consider an initial valuation v such that $v(x) \in N$, $v(z) < 0$, *e.g.* $v(x) = 0$, $v(z) = -2$. *In this case the program K has an infinite computation. The reason is that at each step of the computation the value of y*

increases while the value of z remaines unchanged. Thus the condition $y = z$ will never be satisfied. This means that the loop "while ..." will be repeated infinitely many times.
3) Let v be a valuation in which $v(x) = -1$, $v(z) = 2$. For this initial valuation program K has a finite but unsuccessful computation (a failing computation):

$$\left\langle \begin{array}{ccc} x & y & z \\ -1 & - & 2 \end{array}, K \right\rangle$$

$$\left\langle \begin{array}{ccc} x & y & z \\ -1 & - & 2 \end{array}, y : 0;\ while\ y \neq z\ do\ M\ od \right\rangle$$

$$\left\langle \begin{array}{ccc} x & y & z \\ -1 & 0 & 2 \end{array}, while\ y \neq z\ do\ M\ od \right\rangle$$

$$\left\langle \begin{array}{ccc} x & y & z \\ -1 & 0 & 2 \end{array}, M;\ while\ y \neq z\ do\ M\ od \right\rangle$$

$$\left\langle \begin{array}{ccc} x & y & z \\ -1 & 0 & 2 \end{array}, y := y + 1; x := 1 + (1/(1 + x));\ while\ y \neq z\ do\ M\ od \right\rangle$$

$$\left\langle \begin{array}{ccc} x & y & z \\ -1 & 1 & 2 \end{array}, x := 1 + (1/(1 + x));\ while\ y \neq z\ do\ M\ od \right\rangle$$

The computation stops without a result, since the next instruction requires division by zero.

SYMBOL 224 \f "Symbol"

If the program M has a finite computation $\mathbf{Q}$ in a data structure $\mathbb{A}$ at valuation v, and if the final configuration has the form $< v', \emptyset >$, i.e. the list of programs to be executed is empty, then the valuation v' is called the *result* of the computation of M at v.

If $\mathbf{Q}$ is a finite sequence but the list of instructions left for execution is not empty, then $\mathbf{Q}$ is called an *unsuccessful computation* of M.

If $\mathbf{Q}$ is an infinite sequence which starts from the configuration $< v, M >$, then $\mathbf{Q}$ is called an *infinite computation* of M.

Let $M_{\mathbb{A}}$ denote a relation on the set of all valuations such that

$(v, v') \in M_{\mathbb{A}}$ iff there exists a finite, successful computation $\mathbf{Q}$ of the program M in the data structure $\mathbb{A}$ which, starting from the initial valuation v, halts at the valuation v'.

REMARK

Remark 3.2.6 *For any data structure $\mathbb{A}$, for any valuation v, for any program M*
$(v, v\prime) \in M_{\mathbb{A}}$ *iff* $v \xrightarrow{M_{\mathbb{A}}} v\prime$.

The obvious proof is omitted.

SYMBOL 224 \f "Symbol"

We shall use the denotation $W(\mathbb{A})$ to denote the set of all valuations in $\mathbb{A}$.

LEMMA 3.2.1

For any program M and any data structure $\mathbb{A}$, $M_{\mathbb{A}}$ is a partial function, moreover

(1) if M is of the form $(x := t)$, then $Dom(M_{\mathbb{A}}) = Dom(t_{\mathbb{A}})$ and for $v \in Dom(M_{\mathbb{A}})$

$M_{\mathbb{A}}(v) = v'$, where $v'(x) = t_{\mathbb{A}}(v)$, $v'(z) = v(z)$ for $z \neq x$

(2) if M is of the form $q := \gamma$, then $Dom(M_{\mathbb{A}}) = W(A)$ and for $v \in Dom(M_{\mathbb{A}})$

$M_{\mathbb{A}}(v) = v'$, where $v'(q) = 1$ iff $A, v \models \gamma$, $v'(z) = v(z)$ for $z \neq q$,

(3) if M is of the form *begin* K_1; K_2 *end*, then

$Dom(M_{\mathbb{A}}) = \{v : v \in Dom(K_{1_A})$ *and* $K_{1_A}(v) \in Dom(K_{2_A})\}$ and

$M_{\mathbb{A}}(v) = K_{2_A}(K_{1_A}(v))$ for $v \in Dom(M_{\mathbb{A}})$,

(4) if M is of the form *if* γ *then* K_1 *else* K_2 *fi*, then

$Dom(M_{\mathbb{A}}) = \{v : v \in Dom(K_{1_A})$ and $A, v \models \gamma\} \cup \{v : v \in Dom(K_{2_A})$ and $A, v \models \neg\gamma\}$

and for $v \in Dom(M_{\mathbb{A}})$

$$M_{\mathbb{A}}(v) = \begin{cases} K_{1_{\mathbb{A}}}(v) & \text{when } v \text{ satisfies } \gamma \\ K_{2_A}(v) & \text{when } v \text{ satisfies } \neg\gamma \end{cases}$$

(5) if M is of the form *while* γ *do* K *od*, then

$Dom(M_{\mathbb{A}}) = \{v \in Dom(K_A^i)$:where i is the least natural number such that for $j < i$, $v \in Dom(K_{\mathbb{A}}^j)$, $\mathbb{A}, K_{\mathbb{A}}^j(v) \models \gamma$ and such that $\mathbb{A}, K_{\mathbb{A}}^i(v) \models \neg\gamma\}$

and, for $v \in Dom(M_{\mathbb{A}})$, $M_{\mathbb{A}}(v) = K_{\mathbb{A}}^i(v)$, where i takes on the value defined above.

The proof is technical and is left to the reader as an exercise.

DEFINITION 3.2.4

Definition 3.2.7 *Let L be a language of first-order logic. Let P be a class of deterministic iterative programs over the language L. We say that the n-argument partial function $f(x_1, ..., x_n)$, on a data structure $\mathbb{A}$, is programmable in the class P iff there exists a program $M \in P$ which uses at least $n+1$ variables $x_1, ..., x_n, x_{n+1}$ and is such that, for arbitrary $a_j \in A$ of type ij (for $1 \leq j \leq n+1$) and any valuation v such that $v(x_i) = a_i$ for $i \leq n$,*
(1) $(a_1, ..., a_n) \in Dom(f)$ iff there exists a finite successful computation of M at
the initial valuation v,
(2) if $(a1, ..., an) \in Dom(f)$ and $v' = M_{\mathbb{A}}(v)$, then

$f(a_1, ..., a_n) = a_{n+1}$ *iff* $v'(x_{n+1}) = a_{n+1}$
Program M is then considered as an algorithmic definition of the function f.

SYMBOL 224 \f "Symbol"
REMARK

Remark 3.2.8 *Each partial recursive function (cf. Grzegorczyk [21]) is programmable in a data structure of the natural numbers.*

EXAMPLE 3.2.2

Example 3.2.9 *Let us consider the function div(x,y) in the data structure <N,0,s;≤,=>. The following program M defines this function:*

begin
r := x; i := 0;
while $y \leq r$
do
u := y; j := 0;
while $r \neq u$
do
u := s(u); j := s(j)
done;
r := j; i := s(i)
done
end

.

In fact, for any valuation v, if $v(x) < v(y)$*, then the value of the variable i after execution of the program is* 0*; otherwise, i is the greatest natural number n such that* $n * v(y) \leq v(x)$.

DEFINITION 3.2.5

Definition 3.2.10 *We say that a relation* $r(x1, ..., xm)$*, of type* $(i1 * ... * im)$*, on the data structure* $\mathbb{A}$*, is programmable (in the class P), iff there exists a program* $M \in P$ *which uses at least m variables* $x_1, ..., x_m$ *and a propositional variable q, and is such that, for arbitrary* $a_j \in A$ *of type ij (for* $1 \leq j \leq m$*), and any valuation v such that* $v(x_i) = a_i$ *for* $i \leq m$*,* $(a_1, ..., a_m) \in r$ *iff there exists a finite successful computation of M at the initial valuation v, and* $v'(q) = 1$ *for* $v' = M_{\mathbb{A}}(v)$.

REMARK

Remark 3.2.11 *Each recursively enumerable relation is programmable in the class ? (cf. Grzegorczyk [21]) in a data structure of the natural numbers.*

EXAMPLE 3.2.3

Example 3.2.12 *Let us consider a relation $\leq$ in a data structure of the natural numbers $N =< N, 0, s; =>$. This relation is programmable by the following program M :*

begin
  $u := x;$
  $w := y;$
  $q :=$ **true;**
  while $((q \wedge u \neq y) \vee (\neg q \wedge x \neq w))$
  do
    $q := \neg\ q;$
    if q then u := s(u) **else w := s(w) fi**
  done
end

Observe that program M halts for arbitrary initial data in the data structure N (i.e.for all initial values of x and y).

DEFINITION 3.2.6**true**

Definition 3.2.13 *Let Z be an arbitrary set of variables. We say that two valuations v, v' coincide up to the set Z, written symbolically as*
v = v' off Z,
if v(x)=v'(x) for all variables x which are not in Z.
*
Although, in our formalization, the input to a program is defined as the infinite sequence of the values of all variables, it is evident from the definition that the result does not depend on all of them. What are the important values for a given program M? Obviously a program does not change variables which are not in its text. Thus the following holds
v = MA(v) off V(M)
where V(M) is the set of variables occurring in the text of M.
Let us denote by Vout(M) the set of variables which occur in the program M on the left-hand side of at least one assignment instruction. If program M has a result at the initial valuation v in data structure A, then, by the

definition of a computation,
MA(v) = v off Vout(M).
It is a simple observation that the result does not depend on the values of the variables which occur only on the left-hand side of assignment instructions. They play the role of auxiliary var-iables. By analogy with bounded variables in formulas, we can call them bounded variables of the program.
Let Vin(M) denote the variables in M which occur in tests or on the right-hand side of assignment instructions. Changes to the values of these variables may change the result and, moreover,
v=v' off(V-Vin(M)) implies MA(v)=MA(v').
Variables from the set Vin(M) are similar to free variables in formulas and we call them free variables of the program. Note that the sets Vin(M) and Vout(M) are not necessarily disjoint.

To end this section we recall, once again, that a program is a formal expression. The way in which it will be executed depends on the assumed semantics and data structure, which together determine the interpretation of all objects appearing in the program.

EXAMPLE 3.2.4

Example 3.2.14 *Consider the following program M*

begin
z := x;
u := y;
while $z \neq 0 \wedge u \neq 0$
do
if z¿u then z := z-u else u := u-z fi
od;
if z=0 then z := u fi
end.

All variables that occur in the program are of the same type, $>$ *is a two-argument predicate,* $-$ *is a two-argument functor, and* 0 *is a constant.*
(1) Let
$A =< Z, 0, -; >, =, >$
be the structure of the integers with the constant 0*, one two-argument function* $-$ *(minus) and two binary relations* $>$ *(greater than) and* $=$ *(the equality relation). If v defines the initial data which satisfy* $v(x) \neq 0$ *and* $v(y) \neq 0$*, then, after execution of M, the value of z is the greatest common divisor of*

$v(x)$ and $v(y)$.
(2) Let $A2$ be the data structure
$A2 =< W[x], 0, -; >, =>$,
where $W[x]$ is a ring of polynomials over the field of rational numbers, 0 is the zero element of this ring, $-$ is a two-argument operation such that w1-w2= w1modw2 is the modulus of w1 with respect to w2, >is the ordering relation on the set of polynomials (w1>w2 iff the degree of w1 is greater then the degree of w2 or the coefficient of the greatest power of x at which the two polynomials differ is greater in w1 than in w2).
After execution of the same program M in the data structure A2, the value of z is the greatest common divisor of the polynomials given as initial values of x and y.
(3) Let
$A3 =< O, 0, -; >, =>$
be the data structure in which the universe O is the set of all segments on the real plane and such that 0 is the empty segment, $-$ is the difference of segments, and $>$ is the relation "to be longer".
After execution of M in data structure $A3$ the value of the variable z is the greatest common divisor of the given two segments. Observe that in this data structure there are initial valuations for which program M has infinite computations.

chapter:3,page:1

3.3 Semantic properties of programs

3.3.1 The halting problem

One of the fundamental questions which every programmer must answer is : whether his program M has a finite computation for arbitrary initial data? In other words whether his programs halts for all valuations. From the point of view of a user waiting for the result of the computation, a program which may never terminate is not acceptable. The problem we are talking about is called "the halting problem". We can consider three variants of it.

Definition 3.3.1 *$HALT(M) \equiv$ program M has finite computations for all data in all data structures.*
$HALT(M, \mathbb{A}) \equiv$ program M has finite computations for all initial data in the fixed data structure $\mathbb{A}$.
$HALT(M, \mathbb{A}, v) \equiv$ program M halts for the initial valuation v in data structure $\mathbb{A}$.

Example 3.3.2 *Let M_1, M_2, M_3 be programs in the language $\mathcal{L}$ such that*
M_1: ***while*** *q* ***do*** *$q := \neg q$* ***done***
M_2: ***while*** *$x \neq 0$* ***do*** *$x := x - 1$* ***done***
M_3: ***while*** *$x \neq y$* ***do*** *$x := x + 1$* ***done*** *.*
Program M_1 halts in every structure for arbitrary initial data.
Program M_2 halts for every initial data in the structure of the natural numbers with zero and predecessor -1.
Program M_3 halts in the data structure of the natural numbers with successor only if the initial data v satisfy the relation $v(x) \leq v(y)$.

3.3.2 Program correctness

We begin with an example.

Example 3.3.3 *Consider the following problem: find a program M, which computes square root of a given positive number x with a given precision ε in the structure of the real numbers $\mathbb{R}$. Let the result be stored as the value of the variable y.*
We will say that M is a proper solution to our problem if for any initial valuation v in $\mathbb{R}$, such that the value of x is positive (i.e. $v(x) > 0$), the result v' of program M satisfies the condition $(\sqrt{v(x)} - \varepsilon) < v\prime(y) < (\sqrt{v(x)} + \varepsilon)$.
The following program M :

```
begin
    z := 0;
    if x¡1 then y :=1 else y := x fi;
    while ¬ |z − y| < ε
    do
        z := y;
        y := (z + x/z)/2
    done
end.
```

is a proper solution to our problem. The computation of M creates a sequence of real numbers such that
$y_0 = max(v(x), 1)$
$y_{i+1} = (y_i + v(x)/y_i)/2$.
Note that, for every i,
if $v(x) < 1$,then $v(x) \leq y_i \leq 1$ and if $v(x) \geq 1$,then $1 \leq y_i \leq v(x)$.
Moreover, the sequence is monotonic $(\forall i) y_{i+1} \leq y_i$ if $y_0 > 1$ and $(\forall i) y_{i+1} \geq y_i$ otherwise.Thus the sequence$\{y_i\}_{i \in N}$ has a limit which is$\sqrt{v(x)}$. Hence the

program M does not loop for $v(x) \geq 0$. Additionally, the condition controlling the loop in the program M i.e.$|y_i + 1 - y_i| < \delta$ is small enough. As a consequence we have $|y - \sqrt{x}| < \varepsilon$

Definition 3.3.4 *A program M is* correct *with respect to a precondition α and a postcondition β in the data structure $\mathbb{A}$ iff for any valuation v in $\mathbb{A}$, the validity of condition α implies that the result of the program M exists and satisfies the condition β.*

Let us stress that correctness of a program M with respect to a precondition α implies the existence of a finite successful computation of M for valuations satisfying this condition. This is usually all we require, but it is very difficult to prove. Hence, we consider another, weaker property called partial correctness.

Definition 3.3.5 *A program M is* partially correct *with respect to a precondition α and a postcondition β in the data structure $\mathbb{A}$ if, for all initial data for which the program halts, the satisfaction of α by the initial data implies satisfaction of β after execution of the program M by the result.*

Example 3.3.6 *Let M be the following program*

```
begin
    z:=1;
    u:=x; w:=y;
    while y¿0
    do
        if even(w)
        then
            u:=u*u; w:=w/2
        else
            z:=z*u; w:=w-1
        fi
    done
end
```

Consider the structure $\mathbb{R}$ of the real numbers with the usual interpretation of the functors and predicates appearing in the program. Program M is partially correct with respect to the precondition ***true*** *and the postcondition $(z = x^y)$ and is correct with respect to the precondition "y is a natural number" and the postcondition $(z = x^y)$.*

3.3.3 Strongest postcondition

Let $\mathbb{A}$ be a fixed data structure and M a program. What are the properties of the results when we consider only data satisfying a fixed condition which guarantees that program M has a finite successful computation? Thus we are interested in properties of the co-domain of the partial function $M_{\mathbb{A}}$ (cf. Fig.3.6).

Fig.3.6

Definition 3.3.7 *By the* strongest postcondition *of a formula α with respect to a program M in a data structure $\mathbb{A}$, we mean a formula β which satisfies the following conditions:*
(1) for all initial data, if the computation of M has a result and the formula α is satisfied, then the result of program M satisfies the condition β (we say that β is a postcondition of α with respect to the program M) and
(2) for any condition δ, if δ is a postcondition of α with respect to the program M, then β implies δ in the structure $\mathbb{A}$ (the formula β is then the strongest postcondition of α with respect to M).

Example 3.3.8 *Consider the program M*

begin
z:=x; y:=1;
while $z - y \geq 0$
do
z:=z-y;
y:=y+2
done
end

in the structure of the real numbers $\mathbb{R}$ with the usual interpretation of the symbols $+, -, 0, 1, 2, =, \geq$.
The variable y takes on the values of the consecutive odd numbers. As a result, after execution of the ith iteration, the values of variables z, y are, respectively,
$v(x) - 1 - 3 - 5 - ... - (2i - 1)$ *and* $(2i + 1)$.
Since $\sum_{0 \leq j \leq i} (2j - 1) = i^2$,
the value of variable z after the ith iteration is $v(x) - i^2$. The instruction "while" is executed until the difference $v(x) - i2 - (2i + 1)$ is less than zero, i.e. when we find a natural number n, such that
$(n + 1)^2 > v(x)$ *and* $n^2 < v(x)$.

The value of the variable z is equal to $v(x) - \left\lfloor \sqrt{v(x)} \right\rfloor^2$, *and the value of y is* $2 \left\lfloor \sqrt{(x)} \right\rfloor + 1$. *Briefly, if the condition* $v(x) > 0$ *holds for the initial data v, then the program M has a finite, successful computation, and the value of variable z is the distance from the greatest integer square number less than* $v(x)$.
The formula
$\beta \equiv (z = x - \lfloor \sqrt{x} \rfloor^2) \wedge (y = 2 \lfloor \sqrt{x} \rfloor + 1) \wedge x > 0$
is the strongest postcondition of the formula $a \equiv x{>}0$ *with respect to program M in the data structure considered.*
Indeed, if $\mathbb{R}, v \models x > 0$, *then, after execution of the program M, the valuation* $M_{\mathbb{R}}(v)$ *satisfies the formula* β *from the above analysis. The condition (1) of the definition is thus satisfied.*
Let δ *be a formula such that for any valuation v,*
$\mathbb{R}, v \models a$ *and* $v \in Dom(M_R)$ *implies* $\mathbb{R}, M_{\mathbb{R}}(v) \models d$.
Consider any valuation v' and let R,v'$\models b$. *Then*
$v'(y) = 2 \left\lfloor \sqrt{v'(x)} \right\rfloor^2 + 1,$
$v'(z) = v'(x) - \left\lfloor \sqrt{v'(x)} \right\rfloor^2,$
$v'(x) > 0.$
Hence $\mathbb{R}, v' \models \alpha$, *and therefore the program M has a finite computation at the initial valuation v', i.e.* $v' \in Dom(M_{\mathbb{R}})$. *Using our assumption, we then have* $\mathbb{R}, M_{\mathbb{R}}(v') \models (\delta \wedge \beta)$. *Since the behaviour of the program M depends only on the value of the variable x, which, in fact, does not change during the computation, the resulting valuation* $M_{\mathbb{R}}(v')$ *may differ from the initial valuation only on variables y or z. According to the above analysis, the values of variables z, y after execution of the program M satisfy condition* β. *As a consequence,* $M_{\mathbb{R}}(v') = v'$ *and* $\mathbb{R}, v' \models \delta$ *and finally* $\mathbb{R}, v' \models (\beta \Longrightarrow \delta)$ *for all valuations v', i.e.* $\mathbb{R} \models (\beta \Longrightarrow \delta)$.

3.3.4 Weakest precondition

Let M be a program and β some fixed condition. What are the initial conditions on data which ensure that the results of program M satisfy the condition β in the structure $\mathbb{A}$ (cf. Fig.3.7)? In the other words, how can we characterize the domain of function $M_{\mathbb{R}}$?

Fig.3.7

Definition 3.3.9 *By the* weakest precondition *of a formula* β *with respect to a program M in a data structure* $\mathbb{A}$ *we mean a formula* α *which satisfies the following conditions:*

(1) for all initial data in $\mathbb{A}$, which satisfy the condition α, the results of M satisfy the condition β, (i.e. α is a precondition of the formula α with respect to the program M) ,
(2) for any formula δ, if δ is a precondition of the formula β with respect to the program M, then in the structure $\mathbb{A}$, δ implies α.

Example 3.3.10 *Let us consider the program M of the form*
x:= insert(x,y)
in the data structure of queues Q with the natural interpretation of the functor insert. We assume that x is a variable of type queue, and y a variable of type element of queue.
The weakest precondition of a formula $\neg empty(x)$ with respect to the program M is the formula $\neg empty(insert(x, y))$. Observe, that in the data structure Q, the formula $\neg empty(insert(x, y))$ is valid under each valuation.
The situation is not always so simple. Let K be a program of the form:
while ¬first(x)=y ∧ ¬empty(x)
do
x:=delete(x)
od

The weakest precondition of a formula α of the form
$(\neg empty(x) \ \wedge \ first(x) = y)$
in the structure of queues Q is the condition which states that y is an element of queue x. This condition can be written as the following infinite disjunction (which however does not belong to the language FOL) :
$((\neg empty(x) \wedge first(x) = y) \wedge$
$(\neg empty(delete(x)) \wedge first(delete(x)) = y) \wedge ...$
$(\neg empty(delete^{n-1}(x)) \wedge first(delete^{n-1}(x)) = y) \wedge ...)$
If the initial valuation is such, that for some n, $v(x)$ is an n-element queue and $v(y) \in v(x)$, then after removing at most $(n-1)$ first elements we obtain a queue with first element $v(y)$.

3.3.5 Invariants

In many tasks we are interested in what goes on inside the computation process. We are interested more in what properties are unchanged during computation than how the initial valuation differs from the result. In this class of problems we find, for example, simulation programs and operating systems.

Example 3.3.11 *Let us consider a program for the management of seat reservations in an airline-ticket agency. The property which must necessarily*

be continuously valid is, for example:
number of passengers $\leq$ *number of places*

Definition 3.3.12 *By an* invariant *of a program M in a structure data $\mathbb{A}$, we mean a property α such that, for any computation of the program M in the structure $\mathbb{A}$, if the initial valuation satisfies α, then any valuation obtained during computation of the program M also satisfies the condition α.*

3.3.6 Equivalence of programs

Let K and M be two programs obtained as solutions of some task. Are they equivalent? In what sense?

One possible criterion of equivalence of programs is as follows: for all $x \in V$, the value of variable x after execution of K is equal to the value of variable x after execution of the program M, for the same initial data.

Example 3.3.13 *Consider the two programs*

```
K : begin                    M : begin
      z := 0;                      z := y;
      while ¬z = y                 while ¬z = 0
      do                           do
          z := z + 1;                  z := z − 1;
          x := x + 1                   x := x + 1
      done                         done
  end                          end
```

in the structure $\mathbb{R}$ of the real numbers with the usual interpretation of the symbols $+, -, =, 1$.
According to the above definition of equivalence of programs, K and M are not equivalent although this violates our intuition about what equivalence should mean. Both programs computes the sum of x and y if the value of y is a natural number and both loop indefinately otherwise. The programs K, M differ only in the "auxiliary" variable z.

The above naive example shows that the criterion of equivalence of programs is too strong. This suggests that we must restrict our consideration to some important variables, from the point of view of the task to be solved. This leads us to the following example of a criterion which goes in this direction:

Definition 3.3.14 *Programs K, M are equivalent with respect to a set of variables X in the structure $\mathbb{A}$ iff*
(1) for all initial data v, K has a finite computation iff M has a finite computation
$v \in Dom(K_{\mathbb{A}})$ iff $v \in Dom(M_{\mathbb{A}})$ and
(2) if for any fixed initial data both programs have finite successful computations, then the results are identical on the set of variables X,
if $v \in Dom(K_{\mathbb{A}})$ then $K_{\mathbb{A}}(v) = M_{\mathbb{A}}(v)$ off X .

Example 3.3.15 *Let K and M be programs that compute the square root of x, when the value of variable x is a positive number greater than 1, in the structure of real numbers $\mathbb{R}$. The result of the computation is stored as the value of the variable y.*

```
K: begin
     a := 1; b := x;
     while ¬(b-a)<d
     do
       y := (a+b)/2;
       if (a²-x)(y²-x)<0
       then
         b := y else a := y
       fi
     od
   end
```

```
M: begin
     z := 0; y := x;
     while ¬ abs(z-y)<d
     do
       z := y;
       y := (z+x/z)/2
     od
   end
```

The program K computes the square root using the bisection algorithm while the program M computes the square root using the iteration method. The programs K and M are not equivalent with respect to the variable y since the obtained approximations of a square root can be different. Nevertheless we can consider these programs to be equivalent since they compute the same function.

The last example leads to another definition of equivalence: equivalence with respect to a set of properties.

Definition 3.3.16 *Programs K and M are equivalent in the structure $\mathbb{A}$ with respect to the set of properties Z iff for all $\alpha \in Z$ and for all initial data the result of one program satisfies the condition α if and only if the result of the second satisfies the condition α.*

This ends our selection of examples of semantic properties of programs. The world of semantics contains other phenomena. They may be of interest in certain cases.

In this section we have presented some interesting and useful semantic properties of programs.

Nasz dalszy program polega na skonstruowaniu jezyka (zbioru formul), w ktorym mozna wyrazic powyzsze wlasnosci, a nastepnie na dostarczeniu rachunku (logika algorytmiczna) stanowiacego narzedzie dowodzenia, badz odrzucania, tych formul, a w konsekwencji wlasnosci semantycznych programow.

In what follows we are going to express these properties by means of formulas in algorithmic language and prove them by means of logic tools which will be introduced later.

3.4 Algorithmic language

We observed earlier that the first-order language FOL is not sufficiently powerful to express some properties of algebraic structures (relational systems). Many properties of programs are also not expressible in this language. Therefore, we propose an algorithmic language $\mathcal{L}_{\mathrm{A\,lg}}$, an extension of languages of first-order formulas and of deterministic iterative programs. The new language contains both and extends the set of formulas.

Lemma 3.4.1

Lemma 3.4.1 *The halting property is not expressible in the first-order language.*

Proof. Assume the contrary, that for any program M there exists a formula α_M such that for every data structure
(3.1) $\alpha_M \equiv$ program M halts on all input data
Consider the program M of the form
begin $x := 0$; **while** $\neg y = x$ **do** $x := succ(x)$ **od end**
and the class of structures Nat similar to the standard model of the natural numbers $\mathbb{N} =< N, 0, +1; =>$. We define Nat to be the class of models for the following set of classical first-order formulas
$(\forall x)\ succ(x) = succ(x)$
$(\forall x)\ \neg(succ(x) = 0)$
$(\forall x)(\forall y)\ (succ(x) = succ(y) \implies x = y)$. ■

Remark 3.4.2 *If $\mathbb{A} \in Nat$ and the program M halts for all initial data in the structure $\mathbb{A}$, then $\mathbb{A}$ is isomorphic to $\mathbb{N}$.*

Proof. Let $\mathbb{A} =< A, const, f; =>$ and $\mathbb{A} \in Nat$. Then
(3.2) $f(a) \neq const$ for every $a \in A$, and

(3.3) f is a one-to-one total function.
Let h be the function defined on the set $\mathbb{N}$ such that
$h(0) = const$ and $h(n) = f^n(const)$ for $n \geq 1$. Note that h is a homomorphism which maps $\mathbb{N}$ into $\mathbb{A}$, since, for all n, we have

$$h(n+1) \;=\; f^{n+1}(const) \;=\; f(f^n(const)) \;=\; f(h(n)).$$

From the definition of the function h, the constant *const* is the value of h at 0. If $a \in A$, then for the initial valuation v such that $v(y) = a$, the program M has result v' and for some $m \in \mathbb{N}$, $v'(x) = f^m(const) = v(y)$. Hence $h(m) = a$, which means that a is a value of h. As a consequence, h maps $\mathbb{N}$ onto the set A.
Consider two different natural numbers $n \neq m$ such that $n > m$. If $h(n) = h(m)$, then $f^n(const) = f^m(const)$. By property (3.3), applied m times, $f(f^{n-m-1}(const)) = const$ and $n - m - 1 \geq 0$. Since the last equality contradicts (3.2), we can deduce that $h(n) \neq h(m)$ for $n \neq m$.
We can then conclude that h is an isomorphism from $\mathbb{N}$ onto $\mathbb{A}$, which ends the proof of remark. ■

Proof. (ctd.) If $\mathbb{A}$ is a data structure isomorphic to $\mathbb{N}$, then program M halts in $\mathbb{A}$ for any initial data (cf. section 2.3). Hence by property (3.1) we have
$\vdash \alpha_M$ iff
$HALTS(M, \mathbb{A})$ iff
M halts for any initial data in the structure $\mathbb{A}$ iff
$\mathbb{A}$ is isomorphic to the standard structure of the natural numbers $\mathbb{N}$.
Since this last property is not expressible in a first-order language (cf. Grzegorczyk [21]), there is no first-order formula α_M which expresses the halting property of the program $.M.$ in the class Nat. ■

The above example shows that in order to formally express, and later to analyze, the properties of programs or data structures, we must extend the language defined in the previous chapter, cf. [40].

Let Π denote the class of iterative programs over the first-order language $\mathcal{L} =< A, T, F >$ where A is the alphabet, T is the set of terms and F is the set of formulas.

definition 3.4.1

Definition 3.4.3 *By the set of algorithmic formulas over* Π*, we mean the set* $F(\Pi)$*, which is the smallest set satisfying the following conditions:*
(1) $F \subseteq F(\Pi)$*,*
(2) If $\alpha, \beta \in F(\Pi)$*, then the expressions* $\neg\alpha, (\alpha \vee \beta), (\alpha \wedge \beta), (\alpha \Longrightarrow \beta), (\exists x)\alpha$*,* $(\forall x)\alpha$ *are elements of* $F(\Pi)$*,*

(3) If $\alpha \in F(\Pi)$ and $K \in \Pi$, then expressions $(K\alpha)$, $\bigcup K\alpha$, $\bigcap Ka$ are in $F(\Pi)$.

The symbols $\bigcup$ and $\bigcap$ will be called iteration quantifiers in contrast to classical quantifiers $\exists$ and $\forall$.

definition 3.4.2

Definition 3.4.4 *By the algorithmic language over a class of programs Π, we mean the system $\mathcal{L}(\Pi) =< A(\Pi), T, F(\Pi), \Pi >$ such that $A(\Pi)$ is an extension of the alphabet of $\mathcal{L}$ to include the programming constructs* ***begin, end, if, then, else, fi, while, do, done****, T is the set of terms, $F(\Pi)$ is the set of formulas, and Π is the set of programs.*

The meaning of terms and classical formulas of the language $\mathcal{L}(\Pi)$ in a data structure $\mathbb{A}$ is defined as for the first-order language (cf. section 2.3). To give a complete definition of the semantics of the algorithmic language $\mathcal{L}(\Pi)$ it is sufficient to define the semantics of algorithmic formulas.

Let M be an arbitrary program, $M \in \Pi$, α an arbitrary formula, $\alpha \in F(\Pi)$, and v an arbitrary valuation in the data structure $\mathbb{A}$.

$\mathbb{A}, v \models (M\alpha)$ iff the computation of the program M on initial data v is finite and successful, and the result of this computation satisfies the formula α.

Thus the meaning of algorithmic formula $(M\alpha)$ is obtained as a composition of meaning of program M together with the meaning of the formula α (cf. Fig. 3.8).

Fig. 3.8

The meaning of the iterative quantifiers is defined by:
$A, v \models \bigcup M\alpha$ iff $(\exists i)A, v \models (M^i\alpha)$
$A, v \models \bigcap Ma$ iff $(\forall i)A, v \models (M^i\alpha)$.

EXAMPLE 3.4.2

Example 3.4.5 *Let us determine the value of the following algorithmic formula :*
$(q := \gamma)\alpha \equiv (if\ \gamma\ do\ q := true\ else\ q := false\ fi)\alpha$
in an arbitrarily chosen data structure $\mathbb{A}$ at an arbitrary valuation v.
$\mathbb{A}, v \models (q := \gamma)\alpha$ *iff* $\mathbb{A}, (q := \gamma)_{\mathbb{A}}(v) \models \alpha$ *iff*
$\mathbb{A}, v' \models \alpha$ *and* $v'(q) = \gamma_{\mathbb{A}}(v), v'(z) = v(z)$ *off* q *iff*
$[\mathbb{A}, v' \models \alpha$ *and* $v'(q) = true$, $v'(z) = v(z)$ *off* q *and* $\mathbb{A}, v \models \gamma$ *or*
$\mathbb{A}, v' \models \alpha$ *and* $v'(q) = false$, $v'(z) = v(z)$ *off* q *and* $\mathbb{A}, v \models \neg\gamma]$ *iff*
$[\mathbb{A}, v' \models \alpha$ *and* $v' = (q := true)_{\mathbb{A}}(v)$ *and* $\mathbb{A}, v \models \gamma$ *or*
$\mathbb{A}, v' \models \alpha$ *and* $v' = (q := false)_{\mathbb{A}}(v)$ *and* $\mathbb{A}, v \models \neg\gamma]$ *iff*

$\mathbb{A}, v' \models \alpha$ *and* $\mathbb{A}, v \models \gamma$, $v' =$ *(if* γ *do* $q :=$ *true else* $q :=$ *false fi)*$_{\mathbb{A}}(v)$ *iff*
$\mathbb{A}, v \models$ *(if* γ *do* $q :=$ *true else* $q :=$ *false fi)*α .
The above remarks prove that the formula
$(q := \gamma)\alpha \equiv$ *(if* γ *do q:=true else q:=false fi)*α.
is valid at any valuation in any data structure.

REMARK

Remark 3.4.6 *If in some data structure* $\mathbb{A}$*, for a valuation* v *the computation of a program* M *is infinite or unsuccessful then independently of the value of* α*, the value of the algorithmic formula* $(M\alpha)$ *at the valuation* v *is false.*

LEMMA 3.4.1

Lemma 3.4.7 *The formula*
(3.4) $(\textbf{\textit{if}}\ \gamma\ \textbf{\textit{then}}\ K\ \textbf{\textit{else}}\ M\ \textbf{\textit{fi}})\alpha \equiv ((\gamma \wedge K\alpha) \vee (\neg\gamma \wedge M\alpha))$
is valid in any data structure for the algorithmic language $L(\Pi)$*, in which* α*,* γ *are formulas, and* K, M *are programs.*

PROOF

Proof. Let $\mathbb{A}$ be arbitrary fixed data structure for $L(\Pi)$, and let v be arbitrary valuation in $\mathbb{A}$ such that
$\mathbb{A}, v \models$ **if** γ **then** K **else** M **fi** α.
By the definition of the semantics of formulas there exists a finite successful computation of the program **if** γ **then** K **else** M **fi**, whose result v' satisfies α. By lemma 3.2.1, we have
$v' \in$ (**if** γ **then** K **else** M **fi**)$_{\mathbb{A}}(v)$ iff
$\mathbb{A}, v \models \gamma$ and $v\prime = K_{\mathbb{A}}(v)$ or $\mathbb{A}, v \models \neg\gamma$ and $v' = M\mathbb{A}(v)$.
Putting together the above facts we have
$\mathbb{A}, v \models ((\gamma\ \wedge\ K\alpha)\ \vee\ (\neg\gamma\ \wedge\ M\alpha))$.
Since all steps of our reasoning are equivalent statements,
$\mathbb{A}, v \models$ (**if** γ **then** K **else** M **fi**) $\alpha \equiv ((\gamma \wedge K\alpha) \vee (\neg\gamma \wedge M\alpha))$.
We have thus proved that the formula (3.4) is satisfied for any valuation in the structure $\mathbb{A}$; it is therefore valid in $\mathbb{A}$. ■

LEMMA 3.4.2

Lemma 3.4.8 *The formula*
(3.5) $\textbf{\textit{while}}\ \gamma\ \textbf{\textit{do}}\ M\ \textbf{\textit{od}}\ \alpha \equiv \bigcup \textbf{\textit{if}}\ \gamma\ \textbf{\textit{then}}\ M\ \textbf{\textit{fi}}\ (\neg\gamma \wedge \alpha)$
(in which α*,* γ *are formulas, and* M *is a program) is valid in any data structure for the algorithmic language* $\mathcal{L}(\Pi)$*.*

Proof. Let $\mathbb{A}$ be an arbitrary data structure for $\mathcal{L}(\Pi)$, and v be an arbitrary valuation in $\mathbb{A}$ such that $\mathbb{A}, v \models$ **while** γ **do** M **od** α.
There exists a finite, successful computation of the program while γ do M od, and the result of the computation satisfies α. Let $v' =$ (**while** γ **do** M **od)**$_{\mathbb{A}}(v)$. By lemma 3.2.1 there exists an $i \in N$, such that $\mathbb{A}, v \models M^j\gamma$ for $j < i$ and $v' = M^i_{\mathbb{A}}(v)$ and $\mathbb{A}, v \models M^i\neg\gamma$. As a consequence, the computation of the program (if γ then M fi)i is finite and successful and $v' =$ (**if** γ **then** M **fi**)$^i_{\mathbb{A}}(v)$ and $A, v' \models (\neg\gamma \wedge \alpha)$.
From the definition of the iteration quantifier we have
(3.6) $\mathbb{A}, v \models \bigcup$**if** γ **then** M **fi**$(\neg\gamma \wedge \alpha)$.
Conversely, let condition (3.6) be satisfied for some valuation v in $\mathbb{A}$. Then there exists an i such that
$\mathbb{A}, v \models$ (**if** γ **then** M **fi**)$^i(\neg\gamma \wedge \alpha)$.
Let n be the least natural number i with the above property. Since the computation of (**if** γ **then** M **fi**)^n is finite and successful, the computations of all programs (**if** γ **then** M **fi**)^j for all $j < n$ are also finite successful, and, in particular, the value of the test γ is defined. Moreover, if for some j, the result of the program (**if** γ **then** M **fi**)^j satisfies $\neg\gamma$, then the result of the program (**if** γ **then** M **fi**)$^{j+1}$ also satisfies $\neg\gamma$ (the results of these programs are identical). Thus n is the least natural number such that
$\mathbb{A}, v \models$ (**if** γ **then** M **fi**)$^n(\neg\gamma \wedge \alpha)$
$\mathbb{A}, v \models$ (**if** γ **then** M **fi**)$^j\gamma$ for $j < n$ and
(**if** γ **then** M **fi**)$^j_{\mathbb{A}}(v) = M^j_{\mathbb{A}}(v)$ for $j \leq n$.
Hence, n is the least natural number such that $\mathbb{A}, v \models M^j\gamma$ for $j < n$ and $\mathbb{A}, v \models M^n(\neg\gamma \wedge \alpha)$. By lemma 3.2.1, the program **while** γ **do** M **od** has a finite computation, and its result satisfies the formula α, i.e.
$\mathbb{A}, v \models$ **while** γ **do** M **od** α.
We have therefore proved that property (3.5) is valid in any data structure $\mathbb{A}$. ■

LEMMA 3.4.3

Lemma 3.4.9 *The following formula is valid in every data structure*

Proof. (a $\wedge \bigcap$K(a $\Longrightarrow$Ka))$\Longrightarrow \bigcap$Ka.
Assume that for some valuation v in the structure A,
A,v$\models$ a$\wedge \bigcap$K(a$\Longrightarrow$Ka).
By the definition of the quantifier $\bigcap$, we have A,v$\models$ a and, for every i, A,v$\models$ Ki(a $\Longrightarrow$ Ka). Thus, for every i$\in$N, if A,v$\models$ Kia, then A,v$\models$ Ki+1a, and A,v$\models$ a. By the principle of mathematical induction, for every i^3 0, A,v$\models$ Kia, and therefore A,v$\models \bigcap$Ka. Since both the structure and the valuation were arbitrary, the formula ((a$\wedge \bigcap$K(a$\Longrightarrow$Ka))$\Longrightarrow \bigcap$Ka) is an algorithmic tautology. ■

The formula whose validity we have just proved can be viewed as an algorithmic version of the principle of induction.

EXAMPLE 3.4.3

Example 3.4.10 *Let $\mathbb{S}$ be the data structure of stacks. Consider the formula of the form*
$M(empty(x) \wedge empty(y) \wedge bool)$,
where bool is a propositional variable, and M is the following program :

begin
bool:= true;
while *($\neg empty(x) \wedge \neg empty(y) \wedge bool$)*
do
bool:= bool $\wedge$ top(x)=top(y);
x:=pop(x);
y:=pop(y)
od
end

For any valuation v in the structure $\mathbb{S}$, we have
$\mathbb{S}, v \models M(empty(x) \wedge empty(y) \wedge bool)$
if and only if the stack x has the same contents as the stack y.
Let K denote the program occurring in the above example between ***do****, and* ***od*** *and let γ denote the formula $(\neg empty(x) \wedge \neg empty(y) \wedge bool)$. Then*
$\mathbb{S} \models \bigcup$ **if** γ **then** K **fi**$\neg\gamma$
for every valuation $\boldsymbol{v}$ *in $\mathbb{S}$. In fact, if v is an arbitrary valuation and i is equal to the minimum of lengths of the stacks $v(x)$ and $v(y)$, then after the i-th deletion of elements, one of the stacks will be empty. Hence*
$\mathbb{S}, v \models$ (**if** γ **then** K **fi**)$^i\neg\gamma$.
If we know that the value of variable x at the valuation v is a stack with n elements, then
$\mathbb{S}, v \models \bigcap$ **if** γ **then** K **fi** $true \equiv$ (**if** γ **then** K **fi**)$^n\neg\gamma$.

Algorithmic formulas of the form $M\alpha$ are natural and useful tools which allow us to formulate properties of algorithms. We continue the discussion of this problem in the next section. In some applications it is convenient to use (cf. Chapter 5) a version of the algorithmic language which allow us to use notonly algorithmic formulas but also algorithmic terms.

definition 3.4.3

Definition 3.4.11 *The set of algorithmic terms is the smallest set which contains all individual variables and such that*
(1) If τ_1,..,τ_n are algorithmic terms of type $t_1, .., t_n$ and f is n-argument functor of type $t_1 \times ... \times t_n \longrightarrow t$, then the expression $f(\tau_1, .., \tau_n)$ is an algorithmic

term of type t. then the expression of the form $f(\tau_1, .., \tau_n)$ is an algorithmic term of type t,
(2) If τ is an algorithmic term of type t and M is an arbitrary program, then the expression $M\tau$ is an algorithmic term of type t.

REMARK

Remark 3.4.12 *According to above definition, the set of classical terms is a proper subset of the set of algorithmic terms.*

The semantics of algorithmic terms of the form $M\tau$ is based on the same idea as the semantics of formulas of the form $M\alpha$ and is illustrated in figure 3.9.

Fig. 3.9

For all fixed data structures $\mathbb{A}$ and valuations v, the result $(M\tau)_{\mathbb{A}}(v)$ is defined iff the program M has a finite successful computation at the valuation v and for result v' of this computation $\tau_{\mathbb{A}}(v')$ is defined. If the value $(M\tau)_{\mathbb{A}}(v)$ is defined then the following equality holds

$(M\tau)_{\mathbb{A}}(v) = \tau_{\mathbb{A}}(M(v))$.

EXAMPLE 3.4.4

Example 3.4.13 *Let M be the program of the form*
while *$x \geq y$* ***do*** *x:=x-y* ***done****.*
The algorithmic term Mx defines, in the structure of the natural numbers, the two-argument function whose value, for arguments x, y, is the remainder of the division of x by y.

Remark

Remark 3.4.14 *For arbitrary $\mathbb{A}$ and v, if the values of all the terms which occur in all the expressions below are defined, then*
$\mathbb{A}, v \models Mf(\tau_1, ..., \tau_n) = f((M\tau_1), ..., (M\tau_n))$
$\mathbb{A}, v \models M'(M\tau) = (\textbf{begin } M'; M \textbf{ end})\tau$.

This simple observation allows us to formulate the following lemma. The proof is by induction with respect to the structure of terms, and is left to the reader.

LEMMA 3.4.4

Lemma 3.4.15 *For any algorithmic term τ, there exists a program M and a variable x such that for every $\mathbb{A}$ and v*
(1) $v \in Dom(M_{\mathbb{A}})$ iff $v \in Dom(\tau_{\mathbb{A}})$ and
(2) if $(Mx)_{\mathbb{A}}(v), \tau_{\mathbb{A}}(v)$ are defined then $(Mx)_{\mathbb{A}}(v) = \tau_{\mathbb{A}}(v)$.

3.5 Expressiveness of the AL language

In this section we will show that all properties of programs and of data structures which are mentioned earlier are expressible in the algorithmic language $\mathcal{L}(\Pi)$.

We shall assume as usual that $\mathbb{A}$ is an arbitrary data structure, M, K are arbitrary programs and α, β are arbitrary formulas in the language $\mathcal{L}(\Pi)$.

LEMMA 3.5.1

Lemma 3.5.1 $\mathbb{A} \models M$ **true** *iff all computations of* M *in the structure* $\mathbb{A}$ *are finite and successful.*

PROOF

If the result of a computation of M is defined then it obviously satisfies the formula **true**. Conversely, if for some valuation$v$, $\mathbb{A}, v \models M$ **true**, then according to assumed meaning of algorithmic formulas, there exists a successful, finite computation of program M.

$\diamondsuit$

Definition 3.5.2 *Let us denote by* $loop(M)$ *the formula defined recursively as follows*

$loop(x := \omega) \stackrel{df}{=} \boldsymbol{false}$

$loop(\boldsymbol{begin}\ M_1;M_2\ \boldsymbol{end}) \stackrel{df}{=} loop(\ M_1) \vee (M_1\ loop(M_2))$

$loop(\boldsymbol{if}\ \gamma\ \boldsymbol{then}\ M_1\ \boldsymbol{else}\ M_2\ \boldsymbol{fi}) \stackrel{df}{=} ((\gamma \wedge\ loop(M_1)) \vee (\neg\gamma \wedge loop(M_2)))$

$loop(\boldsymbol{while}\ \gamma\ \boldsymbol{do}\ M\ \boldsymbol{od}) \stackrel{df}{=} (\neg M\ \gamma) \vee (\gamma \wedge \bigcup(\ \boldsymbol{if}\ \gamma\ \boldsymbol{then}\ M\ \boldsymbol{fi})(\gamma \wedge loop(M)))$.

LEMMA 3.5.2

Lemma 3.5.3 *For any valuation* v *in* $\mathbb{A}$, $\mathbb{A}, v \models loop(M)$ *iff* M *has an infinite computation for initial data* v.

PROOF

The proof proceeds by induction with respect to the structure of the program M. If program M is an assignment instruction, then M has no infinite computation no matter what the data structure and valuation are. Assume that the theorem is valid for program K and that M is of the form **while** γ **do** K **od**. If $\mathbb{A}, v \models loop(M)$, then by the definition of loop,

$\mathbb{A}, v \models \gamma \wedge \bigcup$ **if** γ **then** K **fi** $(\gamma \wedge loop(K))$ or $\mathbb{A}, v \models \neg K\gamma$.

In the first case $\mathbb{A}, v \models \gamma$ and there exists a natural number n such that

$\mathbb{A}, v \models$ **if** γ **then** K **fi**^{n} $(\gamma \wedge loop(K))$.

Therefore, there are valuations $v_0 = v, v_1, ..., v_{n-1}, v_n$ such that $v_i = K_{\mathbb{A}}(v_{i-1})$ for $i < n$ and $\mathbb{A}, v_n \models (\gamma \wedge loop(K))$. Let Θ_i be a computation of the program K with initial valuation v_i, $i \leq n$ (by our inductive hypothesis Θ_n is an infinite computation). Then the sequence $\Theta_1\Theta_2...\Theta_n$ is an infinite computation of the program **while** γ **do** K **od**.
If $\mathbb{A}, v \models \neg K\gamma$, then each iteration of program K has a finite computation and its result satisfies γ. This means that K in the program **while** γ **do** K **od** will be executed infinitely many times.
To prove the converse, observe that $\mathbb{A}, v \models \gamma$ is a necessary condition for the existence of an infinite computation of program M at initial valuation v. Moreover
(1) either after some finite number of iterations program K has an infinite computation
(2) or each iteration of program K has a successful computation which result satisfies formula γ.
These two cases corresponds to the validity of the formula $\neg K\gamma$ or the formula
$(\gamma \wedge \bigcup$**if** γ **then** K **fi**$(\gamma \wedge loop(K)))$.
As a result $\mathbb{A}, v \models loop(M)$.
The proof for the other forms of program M follow analogously.

$\diamondsuit$

Next, let us consider unsuccessful computations. A computation is unsuccessful if during the computation an operation is encountered which at the current valuation has an undefined value. Similarly as in the case of looping, this property can be defined recursively.

Definition 3.5.4 *Let us denote by fail(M) a formula defined recursively:*
$fail(x := \omega) \stackrel{df}{=} \neg(x := \omega)\boldsymbol{true}$
$fail(q := \gamma) \stackrel{df}{=} \boldsymbol{false}$
$fail(\boldsymbol{if}\ \gamma\ \boldsymbol{then}\ M_1\ \boldsymbol{else}\ M_2\ \boldsymbol{fi}) \stackrel{df}{=} (\gamma \wedge fail(M_1) \vee \neg\gamma \wedge fail(M_2))$
$fail(\boldsymbol{begin}\ M_1; M_2\ \boldsymbol{end}) \stackrel{df}{=} (fail(M_1) \vee M_1 fail(M_2))$
$fail(\boldsymbol{while}\ \gamma\ \boldsymbol{do}\ M\ \boldsymbol{od}) \stackrel{df}{=} (\gamma \wedge \bigcup if\ \gamma\ then\ M\ fi\ (\gamma \wedge fail(M)))$.

LEMMA 3.5.3

Lemma 3.5.5 *For any valuation v in $\mathbb{A}$, $\mathbb{A}, v \models fail(M)$ iff M has an unsuccessful computation at the initial valuation v in $\mathbb{A}$.*

The proof proceeds by induction with respect to the structure of programs and is similar to those presented above.

$\diamondsuit$
LEMMA 3.5.4

Lemma 3.5.6 $\mathbb{A} \models(\alpha \Longrightarrow M\beta)$ *iff program* M *is correct with respect to the precondition* α *and postcondition* β *in* $\mathbb{A}$.

PROOF
Suppose that $\mathbb{A}, v \models(\alpha \Longrightarrow M\beta)$ for some valuation. If $\mathbb{A}, v \models \alpha$, then by this assumption $\mathbb{A}, v \models M\beta$. According to the semantics of formulas, there exists a finite computation of M at initial valuation v and the result $v' = M_{\mathbb{A}}(v)$ of this computation satisfies β, i.e. $\mathbb{A}, v' \models \beta$. Hence the post-condition is valid.
Conversely, the validity of the precondition α guarantees the existence of a finite successful computation (cf. def.3.3.2) and guarantees that the result satisfies β. Thus, if for some valuation v, $\mathbb{A}, v \models \alpha$, then . Therefore

$$\mathbb{A}, v \models (\alpha \Longrightarrow M\beta) \tag{3.1}$$

.$\diamondsuit$

The partial correctness property of a program M with respect to formulas α and β is expressible by the algorithmic formula $((\alpha \wedge Mtrue) \Longrightarrow M\beta)$. Validity of this formula in the data structure $\mathbb{A}$ means that whenever the initial data satisfies precondition α and the program M has a finite successful computation at v in $\mathbb{A}$, then the result of the program M satisfies formula β.
LEMMA 3.5.5

Lemma 3.5.7 *The formula of the form* $(M\alpha)$ *is the weakest precondition of the program* M *with respect to the formula* α.

PROOF
Let $\mathbb{A}, v \models(M\alpha)$ for some valuation v in $\mathbb{A}$. According to the semantics of algorithmic formulas, the result v' of program M is defined and $\mathbb{A}, v' \models \alpha$. Thus $M\alpha$ is a precondition of M with respect to the formula α.

Consider any formula δ such that the property $\mathbb{A}, v \models \delta$ implies the existence of a valuation v' satisfying $v' = M_{\mathbb{A}}(v)$ and $\mathbb{A}, v' \models \alpha$. Hence $\mathbb{A}, v \models \delta$ implies $\mathbb{A}, v \models M\alpha$, and therefore for any valuation v, $\mathbb{A}, v \models (d \Longrightarrow M\alpha)$. This proves that $(M\alpha)$ is the weakest precondition of M with respect to the formula α.
$\diamondsuit$

In what follows we use the following denotations and definitions.

An element of a vector $\mathbf{x}$ will be denoted by x, the i-th element is denoted as usual by x_i. Two vectors of variables $\mathbf{x}, \mathbf{y}$ will be called *corresponding* iff

their lengths are equal, say equal to n, and, for $i = 1, ..., n$, the type of x_i is the same as the type of y_i. If $\mathbf{x}, \mathbf{y}$ are corresponding vectors and $x \in \mathbf{x}$, then the corresponding element of the vector $\mathbf{y}$ is denoted by y. Let $\mathbf{x}$ be the sequence of all variables that occur in the formula $M\alpha$ and let $\mathbf{y}$ be a sequence of different variables corresponding to $\mathbf{x}$ such that $\mathbf{x} \cap \mathbf{y} = \emptyset$. Let $\alpha(\mathbf{x}/\mathbf{y})$ denote the formula and $M(\mathbf{x}/\mathbf{y})$ the program, obtained, respectively from a formula α and a program M, by simultaneous replacement of all occurrences of variables from the sequence $\mathbf{x}$ by the corresponding variables from the sequence $\mathbf{y}$. By $(\exists \mathbf{y})$ we denote the sequence of quantifiers $(\exists y_1)...(\exists y_n)$.

LEMMA 3.5.6

Lemma 3.5.8 *The formula $\beta = (\exists y)(\alpha(x/y) \wedge M(x/y)(x = y))$ is the strongest postcondition of the program M with respect to the formula α.*

PROOF

Consider a valuation v in $\mathbb{A}$ such that

(3.7)

$$\mathbb{A}, v \models (\alpha \wedge Mtrue) \tag{3.2}$$

There exists a valuation v' such that

$v' = M_{\mathbb{A}}(v)$ and $\mathbb{A}, v \models \alpha$.

Denote by v'' the valuation obtained from v' by the following changes of the values of the variables in y,

$v''(y) = v(x)$ for $y \in \mathbf{y} v''(x) = v'(x)$ for $x \in \mathbf{x}$.

By our assumption, we have

$\mathbb{A}, v'' \models \alpha(x/\mathbf{y})$ and $\mathbb{A}, M(x/y)_{\mathbb{A}}(v'') \models x = y$.

Hence and from the definition of the existential quantifier,

$\mathbb{A}, v' \models (\exists \mathbf{y})(\alpha(\mathbf{x}/\mathbf{y}) \wedge M(\mathbf{x}/\mathbf{y})(\mathbf{x} = \mathbf{y}))$.

Since $v' = M_{\mathbb{A}}(v)$, and using (3.7), we have finally $\mathbb{A}, v \models ((\alpha \wedge M\ \mathbf{true}) \Longrightarrow M\beta)$. The above reasoning is valid for any valuation and we have therefore proved that the program M is partially correct with respect to the formulas α and β,

$\mathbb{A} \models ((\alpha \wedge M\ \mathbf{true}) \Longrightarrow M\beta)$.

Now, assume now that δ is any formula such that

$\mathbb{A} \models ((\alpha \wedge M\ \mathbf{true}) \Longrightarrow M\delta)$.

We will prove that $\mathbb{A} \models (\beta \Longrightarrow \delta)$. Assume on the contrary, that for some valuation v,

(3.8)$\mathbb{A}, v \models \beta$ and non $\mathbb{A}, v \models \delta$.

The formula β forces the existence of an initial valuation v' satisfying α and for which the program M has a finite computation.

$\mathbb{A}, v \models \beta$ iff $(\exists v')\mathbb{A}, v' \models \alpha$ and $v = M_{\mathbb{A}}(v')$.

Hence, and from assumption (3.8), non $\mathbb{A}, v' \models M\delta$ and simultaneously $\mathbb{A}, v' \models \alpha$ and $\mathbb{A}, v' \models M$ true. Thus non $\mathbb{A}, v' \models ((\alpha \wedge Mtrue) \Longrightarrow M\delta)$, which contradicts our assumption that δ was a postcondition.

$\diamondsuit$

By an invariant of a program, we mean a formula which, if valid on the initial data, is then also valid during all steps of the computation. We shall try to define this property by means of algorithmic formula.

Definition 3.5.9 *Let $inv_\alpha(M)$ be a formula defined recursively with respect to the structure of a program M as follows*
$inv_\alpha(x := \omega) \stackrel{df}{=} ((x := \omega)true \Longrightarrow (x := \omega)\alpha)$
$inv_\alpha(q := \gamma) \stackrel{df}{=} (q := \gamma)\alpha$
$inv_\alpha(\textbf{\textit{if}}\ \gamma\ \textbf{\textit{then}}\ M_1\ \textbf{\textit{else}}\ M_2\ \textbf{\textit{fi}}) \stackrel{df}{=} ((\gamma \wedge inv_\alpha(M_1)) \vee (\neg\gamma \wedge inv_\alpha(M_2))$
$inv_\alpha(\textbf{\textit{begin}}\ M1; M2\ \textbf{\textit{end}}) \stackrel{df}{=} (inv_\alpha(M_1) \wedge M_1 inv_\alpha(M_2))$
$inv_\alpha(\textbf{\textit{while}}\ \gamma\ \textbf{\textit{do}}\ M\ \textbf{\textit{od}}) \stackrel{df}{=} (\bigcap \textbf{\textit{if}}\ \gamma\ \textbf{\textit{then}}\ M\ \textbf{\textit{fi}}\ (\gamma \Longrightarrow inv_\alpha(M))$.

LEMMA 3.5.7

Lemma 3.5.10 *For every formula α,*

$\mathbb{A} \models (\alpha \Longrightarrow inv_\alpha(M))$ iff α is an invariant of M.

PROOF

If M is an assignment instruction $x := \omega$, where x is an individual variable and ω is a term, then for any valuation v, M has either a finite successful computation or a finite but unsuccessful computation. Thus

$\mathbb{A} \models\ ((\alpha \wedge (x := \omega)true) \Longrightarrow (x := \omega)\alpha)$

exactly when α is an invariant of $x := \omega$. Analogously for the assignment instruction of the form $q := \gamma$.

Assume (inductive assumption), that, for all programs K which are of simpler structure than M, the lemma holds, i.e.

$\mathbb{A} \models (\alpha \Longrightarrow inv\alpha(K))$ iff α is an invariant of K.

Consider a program M of the form **while** γ **do** K **od** and suppose that (3.9)

$$\mathbb{A} \models (\alpha \Longrightarrow \bigcap \textbf{if}\ \gamma\ \textbf{then}\ K\ \textbf{fi}\ (\gamma \Longrightarrow inv_\alpha(K)) \tag{3.3}$$

Suppose that the sequence of all valuations that occur in the computation of M, at the initial valuation v, has the following form:

$\Theta = v_1, \Theta_1, v_2, \Theta_2, ..., v_n, \Theta_n, v_{n+1}, ...$

where $v_1 = v$ and v_i, Θ_i, v_{i+1} denote the sequence of valuations which occur in the computation of K which leads from v_i to v_{i+1} for $i \leq n$. Suppose that, for $i < n$, all valuations in the sequence v_i, Θ_i, v_{i+1} satisfy α. In

particular, $\mathbb{A}, v \models \alpha$ for $i \leq n$. Since $v_1, \Theta_1, v_2, \Theta_2, ...\Theta_{n-1}, v_n$ is the sequence of all valuations that occur in the computation of (**if** γ **then** K **fi**)$^{n-1}$, and by assumption (3.9), we have

$\mathbb{A}, v \models$ (**if** γ **then** K **fi**)$^{n-1}(\gamma \Longrightarrow inv_\alpha(K))$,

and therefore

(3.10)$\mathbb{A}, vn \models (\gamma \Longrightarrow inv_\alpha(K))$.

If $\mathbb{A}, v_n \models \neg\gamma$, then the computation of M is finite and either unsuccessful or vn is its result. In both cases, all states of the computation satisfy formula α. If $\mathbb{A}, v_n \models \gamma$, then, by property (3.10), $\mathbb{A}, v_n \models inv\alpha(K)$. Since we also have $\mathbb{A}, v_n \models \alpha$, by our inductive assumption, all states of the computation of K from the initial state v_n satisfy α. Thus all elements of the sequence $v_1, \Theta_1, ..., v_n, \Theta_n, v_{n+1}$ satisfy the formula α. Standard inductive reasoning leads to the conclusion that each state of each computation of M satisfies the formula α. This justifies the lemma in the case that $M =$ **while** γ **do** K **od**.

We leave to the reader the verification of the remaining cases.

$\diamondsuit$

Definition 3.5.11 *Assume the following definition:*

$\perp_\alpha (z := \omega) \stackrel{df}{=} (\alpha \vee (z := \omega)\alpha)$

$\perp_\alpha$ (**if** γ **then** M_1 **else** M_2 **fi**)$\stackrel{df}{=}(\alpha \vee (\gamma \wedge \perp_\alpha (M_1)) \vee (\neg\gamma \wedge \perp_\alpha (M_2)))$

$\perp_\alpha$ (**begin** $M_1; M_2$ **end**)$\stackrel{df}{=}(\perp_\alpha (M_1) \vee M_1(\perp_\alpha (M_2)))$

$\perp_\alpha$ (**while** γ **do** M **od**)$\stackrel{df}{=}$ $(\alpha \vee \bigcup$ **if** γ **then** M **fi**$(\gamma \wedge \perp_\alpha (M)))$

LEMMA 3.5.8

Lemma 3.5.12 *For any formula α and any program M, $\mathbb{A} \models \perp_\alpha M$ iff each computation of M in $\mathbb{A}$ has a state which satisfies the formula α.*

Proof.

It suffices to show, that, for any valuation v in the data structure $\mathbb{A}$, the following property holds:

(3.11) $\mathbb{A}, v \models \perp_\alpha M$ iff there exists a state in the computation of M, from the initial valuation v, which satisfies α.

Assume that the formula α is not satisfied by the valuation v, since otherwise the lemma is trivial. The proof proceeds by structural induction.

If M is an assignment instruction of the form $x := \omega$, then the valuation $[x := \omega]_{\mathbb{A}}(v)$ is defined and satisfies the formula α. Obviously the theorem holds in this case.

Assume property (3.11) for all programs which are simpler than M. If M is of the form **if** γ **then** M_1 **else** M_2 **fi** then the computation of M is

identical to the computation of M_1 or to the computation of M_2 depending on the value of formula γ. By our inductive assumption, the property (3.11) is therefore valid. In the case of the composed instruction **begin** $M_1; M_2$ **end** the computation of M consists of the computation of M_1, followed by the computation of M_2 starting from the initial valuation $M_{1\mathbb{A}}(v)$ (provided that the computation of M_1 was successful). The valuation which satisfies the formula α occur in the computation of M_1 or in the computation of M_2. By our inductive assumption on the programs M_1 and M_2,property (3.11) is valid for M.

We now examine the case of the iteration instruction, i.e. let $M =$ **while** γ **do** K **od**. Consider the computation of M from the initial valuation v in $\mathbb{A}$. Independently of the length of the computation, the consecutive valuations that occur in it can be obtained as the elements of the computation (from the same initial state v) of **if** γ **then** K **fi**^{n}, for some n. In particular, the valuation v' satisfying the formula α, is an element of the computation of the program **if** γ **then** K **fi**^{n}, for some n. Let n be the least natural number with this property. Thus

- v' does not belong to the computation of program **if** γ **then** K **fi**$^{n-1}$
- the result v'' of **if** γ **then** K **fi**^{n} satisfies γ, $\mathbb{A}, v'' \models \gamma$ and
- v' belongs to the computation of K from the valuation v''.

Hence $\mathbb{A}, v \models$ **if** γ **then** K **fi**$^{n-1}(\gamma \wedge \ \perp_\alpha K)$ for some n, which means that $\mathbb{A}, v \models \bigcup$ **if** γ **then** K **fi**$(\gamma \wedge \ \perp_\alpha K)$. Thus the property (3.11) has been proved. The lemma follows immediately. ■ ♢

Algorithmic formulas also allow to express properties of a different character. As an example, consider the formulas
(**if** γ **then** M **fi**)$^n \neg\gamma$
and
(**if** γ **then** M **fi**)$^n\gamma$.
Validity of the first formula in the structure $\mathbb{A}$ means that the length of all computations of the program **while** γ **do** M **od** in $\mathbb{A}$ are bounded by $(n * const)$, assuming that *const* is the length of all computations of M in $\mathbb{A}$. The validity of the second formula means that a lower bound for all computations of **while** γ **do** M **od** in $\mathbb{A}$ is $(n * const)$.

In the previous section we presented some definitions of equivalence of programs. In two subsequent lemmas, we show how to express equivalence of programs by means of algorithmic formulas. The simple proofs of these lemmas are omitted.

Let $\mathbf{x}$ be the set of all variables that occur in the programs K and M, and let $\mathbf{y}$ be a corresponding sequence of different variables $\mathbf{x} \cap \mathbf{y} = \emptyset$.

LEMMA 3.5.9

Lemma 3.5.13 *Programs K and M are equivalent with respect to the set of variables $\boldsymbol{z}$ in the structure $\mathbb{A}$, $\boldsymbol{z} \subseteq V(K) \bigcup V(M)$ (cf. Definition 3.3.8) iff* $\mathbb{A} \models (\mathbf{y} := \mathbf{x})\, K(M(\mathbf{x}/\mathbf{y}) \wedge \bigwedge_{x \in \mathbf{z} \cap V_i} (x = y)) \wedge \bigwedge_{q \in \mathbf{z} \cap V_0} (Kq \equiv Mq) \wedge (Ktrue \equiv Mtrue)$

□
LEMMA 3.5.10

Lemma 3.5.14 *Programs K and M are equivalent in the structure $\mathbb{A}$ with respect to a set of formulas Z (cf. Definition 3.3.8) iff for any formula $\alpha \in Z$,* $\mathbb{A} \models (M\alpha \equiv K\alpha)$.

□

Algorithmic formulas allow us to express some properties of data structures in which the computations are performed. For example,we can mention two properties: "to be a natural number" and "to be a finite stack". The formula
$(x := 0)(\mathbf{while}\ \neg x = y\ \mathbf{do}\ x := x + 1\ \mathbf{od}\ \mathbf{true})$
is satisfied in the data structure of real numbers by a valuation v if and only if $v(y)$ is a natural number.

The formula
$\mathbf{while}\ \neg \mathit{empty}(x)\ \mathbf{do}\ x := \mathit{pop}(x)\ \mathbf{od}\ \mathbf{true}$
is satisfied in the structure of stacks by a valuation v if and only if $v(x)$ is a finite stack. We shall discuss other properties of data structures in Chapter 5.

Chapter 4

Algorithmic Logic

This chapter presents the formal system of algorithmic logic. The logic was designed to allow formal proofs of semantic properties of programs and data structures. Henceforth, this system will be denoted by $AL(\Pi)$, in this way we indicate that the system is relative to the class of deterministic iterative programs. In later chapters, we discuss algorithmic logics of others classes of programs.

4.1 Axiomatization

The goal of axiomatization is achieved when a set of formulas, called axioms, and a set of inference rules is given, such that all formulas valid in the assumed semantics, and only these formulas, are provable from the axioms. For this reason, the axioms cannot be an arbitrary set of formulas. They must themselves be valid formulas. Moreover, the inference rules must preserve the validity of formulas, i.e. they must lead to valid conclusions if the given premises are valid. The process of deduction of a formula from a given set of axioms with the help of given inference rules is called a formal proof.

Studying the validity of a formula using only semantic methods only is, in general, a complicated and tedious activity. The axiomatic method of inferring formulas from a set of admitted premises (i.e. axioms) sometimes reduces and simplifies this process.

Definition 4.1.1

Definition 4.1.1 *Let X be a set of formulas, and let β be a formula. A pair (X, β) is called an* inference rule. *The formulas from the set X are called the premises, and the formula β is called the conclusion of the inference rule.*

Traditionally an inference rule is denoted by

$$\frac{X}{\beta}$$

If we are able to prove all premises in an inference rule, then by applying the rule we prove the conclusion of the rule. Inference rules are usually presented in the form of schemes. As an example, the following scheme is an inference rule for all formulas α , γ and all programs K, M.

$$\frac{K\alpha, M\alpha}{\textbf{if } \gamma \textbf{ then } K \textbf{ else } M \textbf{ fi } \alpha}$$

The rule says: if the formula α is satisfied after any execution of the program K and after any execution of the program M, then the formula α is satisfied after any execution of the program **if** γ **then** K **else** M **fi** independently of the form of the formula γ.

We say that a formal deductive system is defined when a triple consisting of a formal language, a set of axioms and a set of inference rules is given.

definition 4.1.2

Definition 4.1.2 *By algorithmic logic of deterministic, iterative programs we understand the system $\mathcal{AL}(\Pi)$ determined by the language $\mathcal{L}(\Pi)$ (c.f. 3.2) and by the set Ax of axioms and the set IR of inference rules listed below.*

Axioms

Ax_1 $((\alpha \Longrightarrow \beta) \Longrightarrow ((\beta \Longrightarrow \delta) \Longrightarrow (\alpha \Longrightarrow \delta)))$

Ax_2 $(\alpha \Longrightarrow (\alpha \vee \beta))$

Ax_3 $(\beta \Longrightarrow (\alpha \vee \beta))$

Ax_4 $((\alpha \Longrightarrow \delta) \Longrightarrow ((\beta \Longrightarrow \delta) \Longrightarrow ((\alpha \vee \beta) \Longrightarrow \delta)))$

Ax_5 $((\alpha \wedge \beta) \Longrightarrow \alpha)$

Ax_6 $((\alpha \wedge \beta) \Longrightarrow \beta)$

Ax_7 $((\delta \Longrightarrow \alpha) \Longrightarrow ((\delta \Longrightarrow \beta) \Longrightarrow (\delta \Longrightarrow (\alpha \wedge \beta))))$

Ax_8 $((\alpha \Longrightarrow (\beta \Longrightarrow \delta)) \equiv ((\alpha \wedge \beta) \Longrightarrow \delta))$

Ax_9 $((\alpha \wedge \neg\alpha) \Longrightarrow \beta)$

Ax_{10} $((\alpha \Longrightarrow (\alpha \wedge \neg\alpha)) \Longrightarrow \neg\alpha)$

Ax_{11} $(\alpha \vee \neg\alpha)$

Ax_{12} $((x := \tau)true \Longrightarrow ((\forall x)\alpha(x) \Longrightarrow (x := \tau)\alpha(x)))$
where term τ is of the same type as the variable x

Ax_{13} $(\forall x)\alpha(x) \equiv \neg(\exists x)\neg\alpha(x)$

Ax_{14} $K((\exists x)\alpha(x)) \equiv (\exists y)(K\alpha(x/y))$ for $y \notin V(K)$

Ax_{15} $K(\alpha \vee \beta) \equiv ((K\alpha) \vee (K\beta))$

Ax_{16} $K(\alpha \wedge \beta) \equiv ((K\alpha) \wedge (K\beta))$

Ax_{17} $K(\neg\alpha) \Longrightarrow \neg(K\alpha)$

Ax_{18} $((x := \tau)\gamma \equiv (\gamma(x/\tau) \wedge (x := \tau)true)) \ \wedge ((q := \gamma\prime)\gamma \equiv \gamma(q/\gamma\prime))$

Ax_{19} **begin** $K; M$ **end** $\alpha \equiv K(M\alpha)$

Ax_{20} **if** γ **then** K **else** M **fi** $\alpha \equiv ((\neg\gamma \wedge M\alpha) \vee (\gamma \wedge K\alpha))$

Ax_{21} **while** γ **do** K **od** $\alpha \equiv ((\neg\gamma \wedge \alpha) \vee (\gamma \wedge K(\textbf{while } \gamma \textbf{ do } K \textbf{ od}(\neg\gamma \wedge \alpha))))$

Ax_{22} $\bigcap K\alpha \equiv (\alpha \wedge (K \bigcap K\alpha))$

Ax_{23} $\bigcup K\alpha \equiv (\alpha \vee (K \bigcup K\alpha))$

Inference rules IR :

R_1 $\dfrac{\alpha, (\alpha \Rightarrow \beta)}{\beta}$

R_2 $\dfrac{(\alpha \Rightarrow \beta)}{(K\alpha \Rightarrow K\beta)}$

R_3 $\dfrac{\{s(\textbf{if } \gamma \textbf{ then } K \textbf{ fi})^i(\neg\gamma \wedge \alpha) \Longrightarrow \beta\}_{i\in N}}{(s(\textbf{while } \gamma \textbf{ do } K \textbf{ od } \alpha) \Longrightarrow \beta)}$

R_4 $\dfrac{\{(K^i\alpha \Longrightarrow \beta)\}_{i\in N}}{(\bigcup K\alpha \Longrightarrow \beta)}$

R_5 $\dfrac{\{(\alpha \Longrightarrow K^i\beta)\}_{i\in N}}{(\alpha \Longrightarrow \bigcap K\beta)}$

R_6 $\dfrac{(\alpha(x) \implies \beta)}{((\exists x)\alpha(x) \implies \beta)}$

R_7 $\dfrac{(\beta \implies \alpha(x))}{(\beta \Longrightarrow (\forall)\alpha(x))}$

In rules R_6 and R_7, it is assumed that x is a variable which is not free in β, i.e. $x \notin FV(\beta)$. The rules are known as the rule for introducing an existential quantifier into the antecedent of an implication and the rule for introducing a universal quantifier into the successor of an implication. The

rules R_4 and R_5 are algorithmic counterparts of rules R_6 and R_7. They are of a different character, however, since their sets of premises are infinite. The rule R_3 for introducing a **while** into the antecedent of an implicationis of a similar nature. These three rules are called ω-rules.

The rule R_1 is known as *modus ponens*, or the *cut* rule.

In all the above schemes of axioms and inference rules, α, β, δ are arbitrary formulas, γ and γ' are arbitrary open formulas, τ is an arbitrary term, s is a finite sequence of assignment instructions, and K and M are arbitrary programs.

The deductive system consisting of the language containing only propositional variables and logical connectives $\wedge$, $\vee$, $\neg$, $\Rightarrow$ axioms Ax_1 - Ax_{11} and the inference rule R_1 is called *classical propositional calculus*. (c.f. [44,33]).

The deductive system determined by the first-order language L, the axioms $Ax_1 - Ax_{13}$ and the rules R_1, R_6, R_7 is called *classical predicate calculus* or classical logic (c.f.[44,33]).

Algorithmic logic is an extension of classical predicate calculus to include certain axioms and rules that characterize the new types of operators which appear in the algorithmic language. We observe that the rules R_1, R_6 and R_7 have finitely many premises. In algorithmic logic we also encounter rules with infinitely many premises. Since the sets of axioms and inference rules determine the notion of formal proof, the consequences of admitting the ω-rules will become evident in the process of proving formulas.

definition 4.1.3

Definition 4.1.3 *By a (formal)* proof *of a formula α starting from the set of formulas Z we mean a pair (D, d), where D is a set of finite sequences of natural numbers, and d is a function from the set D into the set $F(\Pi)$ of formulas. The set D is ordered by the relation* to be an initial segment*. The function d and the set D satisfy the following conditions:*

(1) every linearly ordered subset of the set D is finite,

(2) if $c = (i_1, ..., i_n) \in D$, then $d(c) \in Ax \bigcup Z$ if and only if there is no $j \in N$ such that $(i_1, ..., i_n, j) \in D$,

(3) if $(i_1, ..., i_n, j) \in D$ and $d((i_1, .., i_n, j)) = \alpha_j$, $j \in J, J \subset N$ then $c = (i_1, ..., i_n) \in D$ and $d(c)$ is the conclusion in a certain inference rule in which the formulas $\{\alpha_j\}_{j \in J}$ are the premises,

(4) the empty sequence $\emptyset$ belongs to D and $d(\emptyset) = \alpha$.

We observe that the set D in the definition forms a tree. The elements of D are its nodes. The nodes which are n-element sequences of natural numbers

form the n-th level of the tree. Two nodes c and c' are connected by an edge if and only if c is a node on level n, of the form $(i1, ..., in)$, and c' is a node on level $n+1$ and $c' = (i1, ..., in, j)$ for certain $i1, ..., in, j \in N$. The node c' is called the successor, or the son, of the node c. The nodes which have no son are called *leaves* of the tree D. The node which is not a son of any node is called the root of D. The tree D, labeled by formulas in accordance with the above definition, is called the proof tree of the formula associated with the root of D.

Example 4.1.2

Example 4.1.4 *Let $Z = \{\alpha, Mtrue\}$, where α is an arbitrary formula and M is an arbitrary program. Figure 4.1 represents the proof of formula $M\alpha$ from the set Z.*

Fig.4.1

example 4.1.3

Example 4.1.5 *Let α, β, γ be any formulas. Figure 4.2 represents the formal proof of the formula*

$$((\alpha \vee \gamma) \Longrightarrow (\beta \vee \gamma))$$

from the assumption $(\alpha \Rightarrow \beta)$.

Fig.4.2

If the rules which we apply in a proof have finitely many premises then the tree is finite: - it contains only finitely many nodes. Hence in classical logic every proof is finite. As an immediate corollary we obtain the following property: if a formula α has a proof in classical logic, from a set Z of formulas, then it has a proof from a certain finite subset $Z_0 \subset Z$. the set Z_0 contains precisely those formulas of Z which appear in a certain formal proof of α. Thus every theorem of classical logic is a consequence of a finite set of premises.

Unfortunately, in algorithmic logic the situation is much more complicated. A formal proof is not necessarily finite. If a rule with infinitely many premises is used at least once then the width of the tree is infinite. It may also happen that all branches of a tree are finite but that no upper bound on their lengths exists: this means that the tree has infinitely many levels.

example 4.1.4

Example 4.1.6 *Let Z be the infinite set of formulas*

$$Z = \{0 \leq 0, 0 \leq s(0), 0 \leq s(s(0)), \ldots 0 \leq s^i(0), \ldots\}$$

and let α be the formula of the form

$$\neg\textbf{begin } x := 0;\ \textbf{while } 0 \leq x \textbf{ do } x := s(x) \textbf{ od end true}.$$

A formal proof of the formula α from the set Z is given in Figure 4.3. Note that all branches of the proof are finite but, for every natural number n, there exists in the tree a branch of the length n.

Fig.4.3

To check whether a given, finite tree labeled by formulas is a proof, is a rather simple task. Both the set of axioms and the set of inference rules contain only finitely many schemes. Definition 4.1.3 leads us to a straightforward algorithm to answer this question in the case of a finite tree. To construct a proof of a certain formula α is, however a completely different problem. In general, we cannot guess whether the formula has a proof or not. If it has one, then it is impossible to guess its form. Obviously one formula can have many proofs. We cannot say a priori which rules will be used, which assumptions are necessary. The process of building a proof is a creative process requiring a certain ingenuity.

The system presented in this section, a so called Hilbert-like system, is not appropriate for the goal of automatization. There exist other formal systems of algorithmic logic which permit a mechanized proofs, up to a certain extent. One such system is a Gentzen-like system which will be presented later in this book.

definition 4.1.4

Definition 4.1.7 *If a formula α has a proof from a set Z of formulas we denote this by $Z \vdash \alpha$ and read: the formula α has a proof from the set Z. If the formula has a proof from the axioms of algorithmic logic only, we write $\vdash \alpha$, and we call the formula a theorem of algorithmic logic. The function $\mathcal{C} : 2^{\mathcal{F}} \to 2^{\mathcal{F}}$ such that, for a given set Z of formulas, $\mathcal{C}(Z)$ is the smallest set containing $Ax \cup Z$ and closed with respect to the inference rules is called the operation of* syntactical consequence. *If $\alpha \in \mathcal{C}(Z)$ we say that α is a syntactical consequence of the set Z.*

The following properties of the operation $\mathcal{C}$ are a direct consequence this definition.

Lemma 4.1.1

Lemma 4.1.8 *For arbitrary sets of formulas Z, Z_1, Z_2 the following properties hold*

(1) if $Z_1 \subseteq Z_2$, then $\mathcal{C}(Z_1) \subseteq \mathcal{C}(Z_2)$,

(2) $Z \subseteq \mathcal{C}(Z)$,

(3) $\mathcal{C}(\mathcal{C}(Z)) = C(Z)$.

(4) For every formula $\alpha, Z \vdash \alpha$ *iff* $\alpha \in \mathcal{C}(Z)$.

In other words, if a formula α is a consequence of a set Z then it is also a consequence of each superset of Z. Each formula of the set Z is also the consequence of the set. The consequences of the consequences of a set Z are in $\mathcal{C}(Z)$. The last property asserts the equivalence of the relations "to possess a proof from Z" and "to be a consequence of Z".

We can formulate the following, simple lemma which follows immediately.
lemma 4.1.2

Lemma 4.1.9*(1) If* α *is a theorem of classical propositional calculus then* α *is a theorem of algorithmic logic.*

(2) If α *is a theorem of classical predicate calculus then it is a theorem of algorithmic logic.*

DEfinition 4.1.5

Definition 4.1.10 *Let* Z *be a certain set of formulas. If from the fact that for every* $i \in I$, $\alpha_i \in Z$ *it follows that the formula* $\beta \in \mathcal{C}(Z)$, *then the scheme*

$$\frac{\{\alpha_i\}_{i \in I}}{\beta}$$

is called a secondary inference rule.

jak sie ma Z do I?

An example of a secondary inference rule is

$$\frac{\alpha \Rightarrow \beta}{(\alpha \vee \delta) \Rightarrow (\beta \vee \delta)}$$

(c.f.example 4.1.3).

Below we present other examples of secondary inference rules. These rules are helpful for they permit the simplification of the proofs.

example 4.1.5

Example 4.1.11 *For every* $i \in N$, *for every program* M, *for any open formula* γ *and any formula* α, *the following formula is a theorem of algorithmic logic* $AL(\Pi)$.

$$(\textbf{if } \gamma \textbf{ then } M \textbf{ fi})^i(\neg\gamma \wedge \alpha) \Rightarrow \textbf{while } \gamma \textbf{ do } M \textbf{ od}(\neg\gamma \wedge \alpha). \qquad (4.1)$$

Proof. the proof proceeds by induction with respect to the number i. For $i = 0$ we have

$$\vdash (\neg\gamma \wedge \alpha) \Rightarrow \textbf{while } \gamma \textbf{ do } M \textbf{ od}(\neg\gamma \wedge \alpha)$$

as a simple consequence of axioms Ax_{21} and Ax_3.
Let us assume (inductive hypothesis) that

$$\vdash (\textbf{if}\gamma\textbf{then } M \textbf{ fi})^k(\neg\gamma \wedge \alpha) \Rightarrow \textbf{while}\gamma\textbf{do } M \textbf{ od}(\neg\gamma \wedge \alpha).$$

By axiom Ax_{20}

$$\vdash (\textbf{if } \gamma \textbf{ then } M \textbf{ fi})^{k+1}(\neg\gamma\wedge\alpha) \Rightarrow (\gamma\wedge M(\textbf{if } \gamma \textbf{ then } M\textbf{fi})^k(\neg\gamma\wedge\alpha))\vee(\neg\gamma\wedge(\textbf{if } \gamma \textbf{ then } M \textbf{ fi})^k(\neg\gamma/$$

Applying rule R_2 to the inductive hypothesis and making use of axioms $Ax_1 - Ax_4$ (c.f.example 4.6.1), we obtain

$$\vdash (\wedge M(\textbf{if } \gamma \textbf{ then } M \textbf{ fi})^k(\neg\gamma \wedge \alpha)) \Rightarrow (\gamma \wedge M(\textbf{while } \gamma \textbf{ do } M \textbf{ od}(\neg\gamma \wedge \alpha)))$$

and

$$\vdash (\neg\gamma \wedge (\textbf{if } \gamma \textbf{ then } M \textbf{ fi})^k\beta \Rightarrow (\neg\gamma \wedge \alpha)).$$

Hence by axioms $Ax_1, Ax_5 - Ax_9$ (c.f. example 4.1.3),

$$\vdash (\textbf{if } \gamma \textbf{ then } M \textbf{ fi})^{k+1}(\neg\gamma \wedge \alpha) \Rightarrow (\gamma \wedge M(\textbf{while } \gamma \textbf{ do } M \textbf{ od}(\neg\gamma \wedge \alpha)) \vee \neg\gamma).$$

Finally by the axiom Ax_{21}, we obtain

$$\vdash (\textbf{if } \gamma \textbf{ then } M \textbf{ fi})^{k+1}(\neg\gamma \wedge \alpha) \Rightarrow \textbf{while } \gamma \textbf{ do } M \textbf{ od}(\neg\gamma \wedge \alpha).$$

By the rule of mathematical induction we have proved that the formula (4.1) has a proof for any natural number i. ■ example 4.1.6

Example 4.1.12 *For every set Z of formulas, the set $C(Z)$ is closed with respect to the following secondary inference rule called the invariant. rule*

$$\frac{(\alpha \Rightarrow M\alpha)}{((\alpha \wedge \textbf{while } \gamma \textbf{ do } M \textbf{ od true}) \Rightarrow \textbf{while } \gamma \textbf{ do } M \textbf{ od}\alpha)}$$

where α is an arbitrary formula, M any program and γ is any open formula (i.e. a formula without quantifiers or programs).

Proof. We will prove that if the premise of the rule has a proof in AL, then the conclusion also has a proof. The rule is a helpful tool in proving properties of iterative programs. Suppose that

$$Z \vdash (\alpha \Rightarrow M\alpha) \tag{4.2}$$

We shall prove, by induction with respect to i, that the formula

$$((\alpha \wedge \textbf{if } \gamma \textbf{ then } M \textbf{ fi}^i \neg\gamma) \Rightarrow \textbf{while } \gamma \textbf{ do } M \textbf{ od}\alpha) \tag{4.3}$$

has a formal proof from the set Z in algorithmic logic. Denote by IF the program **if** γ **then** M **fi** and by WH the program **while**γ**do** M **od**. It is fairly obvious, by axioms Ax_2, Ax_{21} and by modus ponens, that

$$\vdash ((\alpha \wedge \neg\gamma) \Rightarrow \mathrm{WH}\alpha)$$

Let us assume (inductive hypothesis) that for a certain j

$$Z \vdash ((\alpha \ \mathrm{IF}j\neg\gamma) \Rightarrow \mathrm{WH}\alpha).$$

By axiom Ax_{20} we have,

$$\vdash \mathrm{IF}^{j+1}\neg\gamma \Rightarrow (\gamma \wedge M(\mathrm{IF}^j\neg\gamma) \vee \neg\gamma \wedge (\mathrm{IF}^j\neg\gamma)),$$

and using Ax_5 and the assumption 4.2 we obtain,

$$Z \vdash (\alpha \wedge \mathrm{IF}^{j+1}\neg\gamma) \Rightarrow ((\gamma \wedge M\alpha \wedge M(\mathrm{IF}^j\neg\gamma)) \vee (\neg\gamma \wedge \alpha))$$

Putting together the last formula, axiom Ax_{16}, the inductive hypothesis and formula 4.1 proved in the previous example, we obtain

$$Z \vdash (\alpha \wedge \mathrm{IF}^{j+1}\neg\gamma) \ \Rightarrow ((\gamma \wedge M(\mathrm{WH}\alpha)) \ \vee (\alpha \wedge \neg\gamma)).$$

By axiom Ax_{21} we have

$$Z \vdash (\alpha \wedge \mathrm{IF}^{j+1}\neg\gamma) \Rightarrow \mathrm{WH}\alpha$$

and hence formula 4.3 has been proved for every natural number i. Applying axiom Ax_8 we can transform formula 4.3 to the form required in ω-rule R_3. The rule leads to the conclusion

$$Z \vdash (\mathrm{WH}\ \textbf{true} \Rightarrow (\alpha \Rightarrow \mathrm{WH}\alpha)).$$

Applying axiom Ax_8 once again, we obtain

$$Z \vdash ((\alpha \wedge \mathrm{WH}\ \textbf{true}) \Rightarrow \mathrm{WH}\alpha),$$

which is what we were trying to prove. ■ Example 4.1.7

Example 4.1.13 *The following scheme is a secondary inference rule in AL*

$$\frac{(\alpha \Rightarrow \beta)}{\textbf{while } \beta \textbf{ do } K \textbf{ od true} \Rightarrow \textbf{while } \alpha \textbf{ do } K \textbf{ od true}}$$

Proof. We will give a proof that, for every set Z of formulas, for all formulas α, β and every program K, if the premise has a proof from Z then the conclusion of the rule has a proof from Z .

$Z \vdash (\alpha \Rightarrow \beta)$ {assumption}

$Z \vdash (\neg\beta \Rightarrow \neg\alpha)$ {propositional calculus}

$Z \vdash (\neg\beta \Rightarrow \textbf{while } \alpha \textbf{ do } K \textbf{ od true})$ {Ax_2 and Ax_{21}}

Assume (inductive hypothesis) that, for some $i \in N$,

$Z \vdash \textbf{if } \beta \textbf{ then } K \textbf{ fi}^i \neg\beta \Rightarrow \textbf{while } \alpha \textbf{ do } K \textbf{ od true}$.

Let IF $\stackrel{df}{=}$ **if** β **then** K **fi** and WH $\stackrel{df}{=}$ **while** α **do** K **od**.

$\vdash \textbf{if } \beta \textbf{ then } K \textbf{ fi}^{i+1} \textbf{ true} \Rightarrow (\neg\beta \wedge \text{IF}^i \textbf{ true} \vee \beta \wedge K(\text{IF}^i \textbf{ true}))$

$\vdash (\neg\beta \wedge \text{IF}^i \textbf{ true} \vee \beta \wedge K(\text{IF}^i \textbf{ true})) \Rightarrow (\neg\beta \vee \beta \wedge K(\text{IF}^i \textbf{ true}))$

{The following line comes from the inductive hypothesis, by R_2 andthe secondary rule of I

$Z \vdash (\neg\beta \vee \beta \wedge K(\text{IF}^i \textit{ true})) \Rightarrow ((\neg\alpha \vee \beta \wedge K(\textbf{while } \alpha \textbf{ do } K \textbf{ od true}))$

But

$\vdash \neg\alpha \vee \beta \wedge K(\text{WH } \textbf{true}) \equiv \neg\alpha \vee \neg\alpha \wedge \beta \wedge K(\text{WH } \textbf{true}) \vee \alpha \wedge \beta \wedge K(\text{WH } \textit{true})$,

and hence

$Z \vdash (\neg\beta \vee \beta \wedge K(\text{IF}^i \textbf{ true})) \Rightarrow (\neg\alpha \vee \alpha \wedge K(\textbf{while } \alpha \textbf{ do } K \textbf{ od true}))$

$Z \vdash (\neg\alpha \vee \alpha \wedge K(\textbf{while } \alpha \textbf{ do } K \textbf{ od true})) \Rightarrow \textbf{while } \alpha \textbf{ do } K \textbf{ od true}$

From this sequence of statements, by axiom Ax_1 and by modus ponens, we obtain

$Z \vdash \textbf{if } \beta \textbf{ then } K \textbf{ fi}^{i+1} \textbf{ true} \Rightarrow \textbf{while } \alpha \textbf{ do } K \textbf{ od true}$.

By the principle of induction, we can assert that, for every natural number$i$,

$Z \vdash \textbf{if } \beta \textbf{ then } K \textbf{ fi}^i \textbf{ true} \Rightarrow \textbf{while } \alpha \textbf{ do } K \textbf{ od true}$.

We can now apply the rule R_3 and conclude the proof

$Z \vdash \textbf{while } \beta \textbf{ do } K \textbf{ od true} \Rightarrow \textbf{while } \alpha \textbf{ do } K \textbf{ od true}$.

■

4.2 On the completeness of algorithmic logic

In this section we justify the choice of axioms and inference rules. We prove that the formal system of algorithmic logic is adequate for the semantics we defined earlier. It means that, the system does not admit a proof of a false formula, all formulas which possess a proof are valid. Moreover, for every valid formula there exists a proof.

Lemma 4.2.1 *All axioms of the system $AL(\Pi)$ are tautologies.*

Proof. (a) Consider axiom Ax_{18} in which τ is a term, γ is an open formula, and x is a variable

$$(x := \tau)\gamma(x) \Leftrightarrow (\gamma(x/\tau) \wedge (x := \tau)\textit{true}).$$

Let $\mathbb{A}$ be a fixed data structure and v a valuation of variables. Observe that the formula $(x := \tau)true$ is satisfied by every valuation v such that, v belongs to the domain of $\tau_{\mathbb{A}}$ (c.f. section 2.3). Therefore

$$\mathbb{A}, v \models (x := \tau)\gamma(x) \quad \text{iff} \quad v \in Dom(\tau_{\mathbb{A}})$$

and there exists a finite, successful computation of the program $(x := \tau)$, and the result v' of the computation satisfies the formula $\gamma(x)$.
For every valuation v' such that, $v'(x) = \tau_{\mathbb{A}}(v)$, $\mathbb{A}, v \models \gamma(x/\tau)$ iff $\mathbb{A}, v' \models \gamma(x)$, hence finally $\mathbb{A}, v \models (x := \tau)\gamma(x)$ iff $\mathbb{A}, v \models (\gamma(x/\tau) \wedge (x := \tau)true)$.
(b) Consider the axiom Ax_{21}

$$\mathbf{while}\gamma\mathbf{do}M\mathbf{od}\alpha \Leftrightarrow ((\neg\gamma \wedge \alpha) \vee (\gamma \wedge M(\mathbf{while}\gamma\mathbf{do}M\mathbf{od}\alpha))).$$

We will show that every formula of the above scheme, is satisfied in every data structure $\mathbb{A}$ by every valuation v.
$\mathbb{A}, v \models \mathbf{while}\ \gamma\ \mathbf{do}\ M\ \mathbf{od}\alpha)$ iff there exists a finite, successful computation of the program $\mathbf{while}\ \gamma\ \mathbf{do}\ M\ \mathbf{od}$, the result of which satisfies the formula α iff there exists $i \in N$ such that $v \in Dom(M_j)_{\mathbb{A}}$, for $j < i$, and $\mathbb{A}, v \models M_j\gamma$ for $j < i$ and $\mathbb{A}, v \models M_i(\neg\gamma \wedge \alpha)$ iff $\mathbb{A}, v \models (\neg\gamma \wedge \alpha)$ or there exists $i^1 \neq 0$, such that, for $j + 1 < i$ $\mathbb{A}, v \models M(M^j\gamma$ and $\mathbb{A}, v \models M(M^{i-1}(\neg\gamma \wedge \alpha)$, $v \in Dom(M_{\mathbb{A}})$ and $M_{\mathbb{A}}(v) \in Dom(M_{\mathbb{A}}^{i-1})$ iff $\mathbb{A}, v \models (\neg\gamma\wedge\alpha)$ or $v \in Dom(M_{\mathbb{A}})$ and there exists $i^1 \neq 0$, such that, for $j < i - 1$, $\mathbb{A}, M_{\mathbb{A}}(v) \models M^j\gamma, M_{\mathbb{A}}(v) \in Dom(M^{i-1}\mathbb{A})$ and $\mathbb{A}, M_{\mathbb{A}}(v) \models M^{i-1}(\neg\gamma \wedge \alpha)$ iff $\mathbb{A}, v \models (\neg\gamma \wedge \alpha)$ or $v \in Dom(M_{\mathbb{A}})$ and there exists $i \in N$, such that $M\mathbb{A}(v) \in Dom(Mj\mathbb{A})$ for $j < i$ and $\mathbb{A}, M_{\mathbb{A}}(v) \models M^j$, for $j < i$, and $\mathbb{A}, M_{\mathbb{A}}(v) \models M^i(\neg\gamma \wedge \alpha)$ iff $\mathbb{A}, v \models (\neg\gamma\wedge\alpha)$ or $\mathbb{A}, v \models \gamma$ and $v \in Dom(M\mathbb{A})$ and $\mathbb{A}, M_{\mathbb{A}}(v) \models \mathbf{while}\gamma\ \mathbf{do}\ M\ \mathbf{od}\ \alpha$ iff $\mathbb{A}, v \models ((\neg\gamma \wedge \alpha)\ \vee\ (\gamma \wedge M\ \mathbf{while}\ \gamma\ m\ M\ od\ \alpha))$. In the remaining cases the proof of the lemma is similar. ■ The lemma we just proved, asserts that every axiom of the system $A(\Pi)$ is a scheme of formulas which are valid in any data structure. Now, we shall prove that every inference rule of the system leads from valid premises to valid conclusions.

Lemma 4.2.2 *For every inference rule of the system $AL(\Pi)$ of the form (Z, α), and for every data structure $\mathbb{A}$, if $\mathbb{A} \models Z$, then $\mathbb{A} \models \alpha$.*

Proof. The proof consists in verifying that, for every rule, the validity of its premises implies the validity of its conclusion.
Consider rule R2

$$\frac{\alpha \Rightarrow \beta}{M\alpha \Rightarrow M\beta}$$

Let ($\alpha \Rightarrow \beta$) be a formula which is valid in a data structure $\mathbb{A}$. Assume that, for a certain valuation v, $\mathbb{A}, v \models M\alpha$. From this, it follows that

$v \in Dom(M_{\mathbb{A}})$ and $\mathbb{A}, M_{\mathbb{A}}(v) \models \alpha$. By our assumption on the validity of the premise of the rule, we have $\mathbb{A}, M_{\mathbb{A}}(v) \models \beta$, hence $\mathbb{A}, v \models M\beta$. The valuation v was chosen arbitrarily, hence $\mathbb{A} \models (M\alpha \Rightarrow\ M\beta)$, which ends the proof of the soundness of rule R2.
Consider rule R3

$$\frac{(s(\textbf{if } \gamma \textbf{ then } M \textbf{ fi})^i(\neg\gamma \wedge \alpha) \Rightarrow \beta)_{i\in N}}{s\textbf{while } \gamma \textbf{ do } M \textbf{ od}\alpha) \implies \beta}$$

where s is any sequence of assignment instructions, γ is an open formula, α , β are arbitrary formulas and K, M are any programs.
Let $\mathbb{A}$ be a data structure and v a valuation such that

$$\mathbb{A}, v \models s\ (\textbf{while } \gamma \textbf{ do } M \textbf{ od } \alpha)\text{andnon } \mathbb{A}, v \models \beta.$$

Then, by the lemma 3.4.2, there exists $i \in N$, such that

$$\mathbb{A}, s\mathbb{A}(v) \models (\textbf{if } \gamma \textbf{ then } M \textbf{ fi})^i(\neg\gamma\wedge\alpha)\text{and } \mathbb{A}, v \models s\ (\textbf{if } \gamma \textbf{ then } M \textbf{ fi})^i(\neg\gamma\wedge\alpha).$$

By our assumption that *non* $\mathbb{A}, v \models \beta$, there exists $i \in N$, such that

$$non\ \mathbb{A}, v \models (s(\textbf{if } \gamma \textbf{ then } M \textbf{ fi})^i(\neg \wedge \alpha) \Rightarrow \beta).$$

Therefore the i-th premise of the considered rule is false. By contradiction we have thus proved that if all premises of the rule R_3 are formulas valid in the data structure $\mathbb{A}$, then the conclusion of rule R_3 is also valid in the data structure $\mathbb{A}$.
Consider rule R5 $\dfrac{\{(\beta \Rightarrow K^i\alpha)\}_{i\in N}}{(\beta \Rightarrow \bigcap K\alpha)}$ Assume that for every $i \in N$, $\mathbb{A} \models (\beta \Rightarrow K^i\alpha)$ and suppose that for a certain valuation v $\mathbb{A}, v \models \beta$ and *non* $\mathbb{A}, v \models \bigcap K\alpha$. By the definition of iteration quantifier (c.f. section 3.4) there exists an $n \geq 0$ such that *non*$\mathbb{A}, v \models K^n\alpha$ and consequently

$$non\ \mathbb{A} \models (\beta\ \Rightarrow\ K^n\alpha)$$

which leads to the negation of the assumption. Consider rule R7

$$\frac{\beta \Rightarrow \alpha(x)}{\beta \Rightarrow (\exists x)\alpha(x)}$$

Suppose that $\mathbb{A} \models \beta \Rightarrow \alpha(x))$ and assume that, for a certain valuation v, $\mathbb{A}, v \models \beta$ and *non* $\mathbb{A}, v \models (\exists x)\alpha(x)$. By the definition of the quantifier $\exists$ (cf. section 2.3), there exists a value a of variable x such that $\mathbb{A}, v^a_x \models \neg\alpha(x)$. Since the variable x does not occur as a free variable in the formula β, we

see that $\mathbb{A}, v_x^a \models \beta$. From this it follows that *non* $\mathbb{A}, v_x^a \models (\beta \Rightarrow \alpha(x))$, which contradicts our assumption that the premise of the rule is valid at every valuation.
The proofs in the remaining cases are analogous to the cases already considered. ■

The theorem which we will now prove is called the adequacy of axiomatization theorem and summarizes the above discussion.

Theorem 4.2.3 *For an arbitrary formula α and every set Z of formulas, if there exists a proof of α from the set Z, then the formula α is valid in every model $\mathbb{A}$ for Z.*
$Z \vdash \alpha$ implies $Z \models \alpha$.

Proof. From the assumption $Z \vdash \alpha$ it follows that there exists a formal proof $\langle D, d\rangle$ of the formula α from the set Z. Let $\mathbb{A}$ be a model for the set Z. Observe that if there is a formula β_n on level n of the proof tree $\langle D, d\rangle$ which is not valid in $\mathbb{A}$, then by lemma 4.2.2 one of the formula-premises used to infer β_n is also not valid in $\mathbb{A}$. Hence, on the level $n+1$ there exists a formula β_{n+1} which is also non valid in $\mathbb{A}$ and the corresponding nodes are connected by an edge. Therefore if formula α is non valid in $\mathbb{A}$, then there exists a path (in the proof D of α) on which all formulas are not valid in $\mathbb{A}$. In particular, the formula assigned to the leaf is not valid. This contradicts either lemma 4.2.1 or the assumption of the theorem. For the formula assigned to the leaf is either a tautology (by lemma 4.2.1) or is valid in $\mathbb{A}$ (by the assumption). Since we did not assume anything about the model for Z, the formula α is therefore valid in any model for Z. ■

We can observe the following corollary of the above theorem.

Corollary 4.2.4 *The system $AL(\Pi)$ is consistent.*

There is no formula α such that both α and its negation $\neg\alpha$ have formal proofs in the system. If such a formula existed, then, by the above theorem, both α and $\neg\alpha$ would be valid in any data structure. This is clearly impossible.

Theorem 4.2.3 can be expressed in the following way: if a formula α is a syntactical consequence of a set Z of formulas, then it is also a semantic consequence of the set Z.

The system $AL(\Pi)$ does not admit a proof of a falsifiable (i.e. not valid) formula. This raises another question: does the system have the property that every valid formula has a proof? The following theorem, called completeness theorem, answers this question.

Theorem 4.2.5 *For every set Z of formulas and for every formula α $\alpha \in C(Z)$ iff $\alpha \in Cn(Z)$*

The detailed and difficult proof of this theorem is given elsewhere. Those readers interested in the method for proving the completeness theorem are advised to consult [40].

The following example shows how to apply the completeness theorem in order to show the existence of a proof. We could produce a formal proof, but the reasoning presented below is much shorter and easier to understand.

Example 4.2.1

Example 4.2.6 *Suppose that $V_{out}(K) \cap V(\alpha) = \emptyset$ for a formula α and a program K. Then the formula*

$$K\alpha \Leftrightarrow (\alpha \wedge Ktrue)$$

is a theorem of algorithmic logic.

Indeed, for every data structure $\mathbb{A}$ and for every valuation v,

$$\mathbb{A}, v \models K\alpha \text{iff} \mathbb{A}, v \models Ktrue \text{and} \mathbb{A}, K_{\mathbb{A}}(v) \models \alpha.$$

Since by assumption, the variables which can have the values changed as a result of performing the program K do not occur in the formula α, hence $\alpha_{\mathbb{A}}(K_{\mathbb{A}}(v)) = \alpha_{\mathbb{A}}(v)$.
As a consequence, for any data structure $\mathbb{A}$ we have $\mathbb{A} \models K_\alpha \Leftrightarrow (\alpha \wedge Ktrue)$. By the completeness theorem we obtain $\vdash K\alpha \Leftrightarrow (\alpha \wedge Ktrue)$. This result can also be presented as an inference rule

$$\frac{Ktrue, \alpha}{K\alpha} \quad \text{where} \quad V_{out}(K) \cap V(a) = \emptyset.$$

which is a convenient tool in proofs of properties of programs .

□

example 4.2.2

Example 4.2.7 *We shall prove that, for arbitrary programs K, M and for every formula α , such that $V(M\alpha) \cap V(K) = \emptyset$,*
$\vdash M(K\alpha) \Leftrightarrow (M\alpha \wedge Ktrue)$
and $\vdash K(M\alpha) \Longleftrightarrow M(K\alpha)$.

Proof. Applying rule $R2$ to formula $K\alpha \Longleftrightarrow (\alpha \wedge Ktrue)$ we obtain
$M(K\alpha) \iff (M\alpha \wedge M(Ktrue))$.

Since, by assumption, programs K and M do not have common variables, the formula $M(Ktrue) \iff (Ktrue \wedge Mtrue)$ is valid in every data structure, which by the completeness theorem concludes the proof of $M(K\alpha) \iff (M\alpha \wedge Ktrue)$. If in formula (4.4) we replace the formula by $(M\alpha)$, we get $\vdash K(M\alpha) \iff (M\alpha \wedge Ktrue)$. Applying axiom Ax1 to the last two statements, we obtain $\vdash M(K\alpha) \iff K(M\alpha)$ ■

Let us summarize the results of above considerations. A formula is a theorem of algorithmic logic if and only if it is a tautology. Moreover, the syntactic and the semantic consequence operations coincide. Thus the formal system of algorithmic logic $AL(\Pi)$ allows us to use both semantic or syntactic methods to check the validity of a given algorithmic property.

Unfortunately, as we observed earlier, formal proof's construction is not an automatic procees and is by no means, easy. The existence of the ω-rules is one of the important sources of difficulty. Let us examine the necessity of this rule. Is it possible to characterize completely the set of all valid formulas by means of a finitary axiomatization (that is, by means of rules with only a finite number of premisses)?

Lemma 4.2.8 *The ω-rules can not be avoided and replaced by any (finite or infinite) set of finitary rules.*

Proof. Let X be such a system. The following postulates are valid (by assumption):

(Z1) all inference rules of X are finite,

(Z2) for every set Z of formulas and every formula α , $Z \models \alpha$ iff $Z \overset{\vdash}{X} \alpha$ (i.e. the formula α has a proof from Z in X).

Let Z be the following set of formulas
$\{(x := 0)0 \leq x, ..., (x := 0)((x := succ(x))^i 0 \leq x), ...\}$
and let α be the following formula
$\neg(x := 0)(\textbf{while } 0 \leq x \textbf{ do } x := succ(x) \textbf{ odtrue})$.

Observe that $Z \models \alpha$. Thus by assumption (Z2) we have $Z \vdash \alpha$, ie. the formula α has a proof from the set Z in the system X. Although the set Z is infinite, the formal proof of α in system X is finite by assumption (Z1). Thus there exists a subset $Z_0 \subset Z$ which contains all formulas that appear in the proof of the formula α and such that $Z_0 \overset{\vdash}{X} \alpha$. We will show that this leads to contradiction.

Consider an arbitrary finite subset Z_I of the set Z. It is obviously determined by those natural numbers representing the number of iterations of the program $(x := succ(x))$ in the formulas of the set Z_I. Let

$Z_I = \{(x := 0)(x := succ(x))^i 0 \leq x\}_{i \in I}$

and let $\mathbb{A}$ be the data structure

$\mathbb{A} = \langle N, 0, succ, 0 \leq \rangle >$

where $succ_{\mathbb{A}}$ is the usual successor operation in N, 0 is the constant zero and $0 \leq_{\mathbb{A}}$ is the one-argument relation in N defined by

$(0 \leq)_{\mathbb{A}}(n) = true$ iff $n \in I$.

Thus for any valuation v, we have

$((x := 0)(x := succ(x))^i 0 \leq x)\mathbb{A}(v) = (0 \leq)_{\mathbb{A}} succ^i_{\mathbb{A}}(0) = (0 \leq)_{\mathbb{A}}(i)$.

Hence

$\mathbb{A}, v \models (x := 0)(x := succ(x))^i 0 \leq x$ iff $i \in I$

and therefore $\mathbb{A}$ is a model of the set Z_I. Since I is a finite subset of N, there exists a natural number j such that, $j \notin I$. We then have

$non\ A, v \models (x := 0)(x := succ(x))^j 0 \leq x$.

The program **begin** $x := 0$; **while** $0 \leq x$ **do** $x := succ(x)$ **od end** thus has a finite computation in the data structure $\mathbb{A}$, which implies $non\ \mathbb{A} \models \alpha$.

This demonstrates that, for any finite subset Z_I of Z, there exists a model for Z_I which is not a model for α . As a consequence $non\ Z_I \models \alpha$. This contradiction proves the impossibility of satisfying both assumptions (Z1) and (Z2) . ■ The ω-rules, although not very convenient themselves, enable us to prove many useful secondary inference rules, (cf. Def. 4.1.5), which allow us to prove properties of programs in a finitary way. Below we present an example of such a secondary rule.

example 4.2.3

Theorem 4.2.9 *The following schema is a sound inference rule*

$$\frac{K_1\mathbf{true}, K_2\mathbf{true}}{\mathbf{while}\ \gamma\ \mathbf{do} M\ \mathbf{od}\ \alpha \Leftrightarrow \mathbf{while}\ \gamma\ \mathbf{do}\ K_1; M; K_2\ \mathbf{od}\ \alpha}$$

where γ is any quantifier-free formula, α is any formula, and K_1, K_2, M are programs such that $V_{out}(K_1) \cap V(M\gamma \vee \alpha) = \emptyset$ and $V_{out}(K_2) \cap V(M\gamma \vee \alpha) = \emptyset$.

Proof. Let Z be any set of formulas. If $Z \vdash K_2 true$, then by the rule (4.4) we have

$Z \vdash (\gamma \iff K_2\gamma)$.

Applying rule R2 we have

$Z \vdash K_1(M\gamma) \iff K_1(M(K_2\gamma))$.

Similarly, if $Z \vdash K_1 true$, then by rule (4.4)

$Z \vdash M\gamma \iff K_1(M\gamma)$.

The above two formulas imply by axiom Ax1 that

$Z \vdash M\gamma \Leftrightarrow$ **begin** $K_1; M; K_2$ **end**γ .

An analogous argument allows us to obtain

$Z \vdash M(\neg\gamma \wedge \alpha) \Leftrightarrow \mathbf{begin}\ K1; M; K2\ \mathbf{end}(\neg\gamma \wedge \alpha)$.

Let us denote as β the formula $(\neg\gamma \wedge \alpha)$. By the axioms of the propositional calculus Ax1-Ax11, we can infer

$Z \vdash (\gamma \wedge M\beta) \vee \beta \Leftrightarrow (\gamma \wedge K_1(MK_2\beta)) \vee (\neg\gamma \wedge \beta)$

which by Ax20 is equivalent to

$Z \vdash \mathbf{if}\ \gamma\ \mathbf{then}\ M\ \mathbf{fi}\beta \Leftrightarrow \mathbf{if}\ \gamma\ \mathbf{then}\ K_1; M; K_2\ \mathbf{fi}\beta$.

Straightforward induction leads us to

$Z \vdash (\mathbf{if}\ \gamma\ \mathbf{then}\ M\ \mathbf{fi})^i\beta \Leftrightarrow (\mathbf{if}\ \gamma\mathbf{then}K_1; M; K_2\ \mathbf{fi})^i\beta$

for any natural number i.

Observe that, by our assumptions, $V_{out}(K1) \cap V(M^i\beta) = \emptyset$ and $V_{out}(K2) \cap V(M^i\beta) = \emptyset$. Thus, by property (4.3) and inference rule R3, we obtain

$Z \vdash \mathbf{while}\ \gamma\ \mathbf{do}\ M\ \mathbf{od}\alpha \Longleftrightarrow \mathbf{while}\ \gamma\ \mathbf{do}\ K_1; M; K_2\ \mathbf{od}\alpha$. ■ Several examples which make use of this rule can be found in the next chapter.

Example 4.2.4

Example 4.2.10 *Let Z be a set of algorithmic formulas and let γ and δ be open formulas such that $Z \vdash (\gamma \Rightarrow \delta)$.*
Let K, M_1, M_2 be arbitrary programs and α an arbitrary formula. Assume that $V_{out}(M1) \cap V(\delta) = \emptyset$. Then the following formula is provable from Z
$\mathbf{while}\ \gamma\ \mathbf{do}\ M_1; \mathbf{if}\ d\ \mathbf{then}\ K\ \mathbf{fi}; M_2\ \mathbf{od}\alpha \Leftrightarrow \mathbf{while}\ \gamma\ \mathbf{do}\ M_1; K; M_2\ \mathbf{od}\alpha$.

PROOF **Proof.** Let $\gamma, \delta, \alpha, K, M_1, M_2$ be as in the above assumptions. By the property proved in example 4.2.1 we have

$Z \vdash M_1\delta \Leftrightarrow (\delta \wedge M_1\mathbf{true})$

$Z \vdash M_1\neg\delta \Leftrightarrow (\neg\delta \wedge M_1\mathbf{true})$

$Z \vdash (\gamma \wedge \delta) \Leftrightarrow \gamma$ $\quad Z \vdash (\gamma \wedge \neg\delta) \Leftrightarrow \mathbf{false}$. Denote by M the following program $\mathbf{begin}\ M_1; \mathbf{if}\ d\ \mathbf{then}\ K\ \mathbf{fi}; M_2\mathbf{end}$ and by M' the program $\mathbf{begin}\ M_1; K; M_2\ \mathbf{end}$. Let moreover β denote an arbitrary formula. Then the formula

$\mathbf{if}\ \gamma\mathbf{then}\ M\ \mathbf{fi}\ \beta$ is equivalent to the subsequent following formulas

$(\gamma \wedge M\beta) \vee (\neg\gamma \wedge \beta)$

$(\gamma \wedge\ M_1(\delta \wedge K(M_2\beta)\ \vee (\neg\delta \wedge M_2\beta)) \vee\ (\neg\gamma \wedge \beta))$

$(\gamma \wedge \delta \wedge M_1(K(M_2(\beta))) \vee (\gamma \wedge \neg\delta \wedge M_1(M_2\beta))) \vee (\neg\gamma \wedge \beta)$

$(\gamma \wedge M'\beta) \vee (\neg\gamma \wedge \beta)$

$\mathbf{if}\ \gamma\ \mathbf{then}\ M'\mathbf{fi}\ \beta$.

Putting $\beta = (\neg\gamma \wedge \alpha)$, we have by straightforward induction

$Z \vdash (\mathbf{if}\ \gamma\mathbf{then}\ M\mathbf{fi})^i(\neg\gamma \wedge \alpha) \Leftrightarrow (\mathbf{if}\ \gamma\mathbf{then}\ M'\mathbf{fi})^i(\neg\gamma \wedge \alpha)$

for all $i \in N$. Finally, by property (4.3) and by inference rule R3, we have

$Z \vdash \mathbf{while}\ \gamma\ \mathbf{do}\ M\ \mathbf{od}\alpha \Leftrightarrow \mathbf{while}\ \gamma\ \mathbf{do}\ M'\mathbf{od}\alpha$.

The above example can be represented as the following sound secondary rule

$$\frac{\gamma \Rightarrow \delta}{\textbf{while } \gamma \textbf{ do } M_1; \textbf{if } \delta \textbf{ then } K \textbf{ fi}; M_2 \textbf{ od}\alpha \Leftrightarrow \textbf{while } \gamma \textbf{ do } M_1; K; M_2 \textbf{ od}\alpha}$$

under the assumption $V_{out}(M1) \cap V(\delta) = \emptyset$. ■

example 4.2.5

Example 4.2.11 *The following schemes are simple secondary rules whose proofs do not require ω-rules.*

$$\frac{\alpha \Rightarrow \neg\gamma}{\alpha \wedge \textbf{if } \gamma \textbf{ then } M \textbf{ fi}\beta \Leftrightarrow \alpha \wedge \beta}$$

$$\frac{\alpha \Rightarrow \neg\gamma}{\alpha \wedge \textbf{while } \gamma \textbf{ do } M \textbf{ od}\beta \Leftrightarrow \alpha \wedge \beta}$$

where γ is any open formula, α ,β are arbitrary formulas and M is a program.

Proof. If formula $(\alpha \Rightarrow \neg\gamma)$ has a proof, then by axioms Ax_2 and Ax_7, there are proofs of the following formulas
$((\alpha \wedge \gamma) \Leftrightarrow (\neg\gamma \wedge \gamma))$,
$((\alpha \wedge \beta) \Leftrightarrow (\neg\gamma \wedge \alpha \wedge \beta))$.
Thus
$\vdash (\alpha \wedge (\gamma \wedge M\delta \vee \neg\gamma \wedge \beta)) \Leftrightarrow (\alpha \wedge \beta)$
for any formula δ.
Now we shall apply this result twice. Setting, first time $\delta = \beta$ we obtain

$$\vdash (\alpha \wedge if\ \gamma then\ M\ fi\beta) \Leftrightarrow (\alpha \wedge \beta)$$

. Setting the second time $\delta = (while\ \gamma\ do\ M\ od\beta)$ we have by axioms Ax_{20} and Ax_{21}

$$\vdash (\alpha \wedge while\ \gamma do\ M\ odGbeta)\ \Leftrightarrow\ (\alpha \wedge \beta).$$

■

The following theorem is of importance, since it leads to a better understanding of the proof concept.

theorem 4.2.5

Theorem 4.2.12 *For every formulas α , β and for every set Z of formulas,*

$$Z \cup \{\alpha\} \vdash \beta \text{ iff } Z \vdash ((\forall x)\alpha(x) \Rightarrow \beta)$$

where x denotes the set of all variables which are free in the formula α .

Proof.

Let **A** be a model for the set Z, and suppose that for some valuation v_0
$\mathbb{A}, v_0 \models (\forall x)\alpha(x)$.
Since the value of this formula does not dependent on the values of the variables x, we have for any valuation v,
$\mathbb{A}, v \models (\forall x)\alpha(x)$.
This implies that **A** is a model for the set $Z \cup \{\alpha\}$.
If we assume that $\beta \in C(Z \cup \{\alpha\})$ then by the above reasoning and by the completeness theorem 4.2.4 we obtain
$\mathbb{A} \models \beta$.
In particular, $\mathbb{A}, v_0 \models \beta$.
Hence, we have proved that, for an arbitrary valuation v, $\mathbb{A}, v \models ((\forall x)\alpha(x) \Rightarrow \beta)$.
Consequently, every model for the set Z is also a model for the formula $((\forall x)\alpha(x) \Rightarrow \beta)$, i.e. $Z \models ((\forall x)\alpha(x) \Rightarrow \beta)$.
By the completeness theorem, we have $Z \vdash (\forall x)\alpha(x) \Rightarrow \beta$. ■
As an immediate corollary of the above, we obtain the following theorem on deduction.

theorem 4.2.6

Theorem 4.2.13 *For any formula β , any closed formula α and any set Z of formulas,*

$$Z \vdash (\alpha \Rightarrow \beta) \quad \text{iff} \quad Z \cup \{\alpha\} \vdash \beta.$$

□

Chapter 5

Other Logics of Programs

5.1 Floyd's descriptions of programs

In 1967 R.W.Floyd [19] introduced the notion of a description of a program in order to analyse the meaning of programs. By a *description* of a program's diagram (c.f. chapter 3) we mean a mapping which to every edge of the diagram associates a formula of the first-order logic. If a description has the property: for every edge of the diagram the formula associated to it, is satisfied by a memory state (a snapshot) when a computation passes along the edge, then the description is called *admissible*. Admissible descriptions are helpful in verification of partial correctness of programs. Namely, if a program possesses an admissible description, then it is partially correct w.r.t. the formulas associated with the initial edge and the terminal edge of the program's diagram. It turned out, that there exists a simple enough, syntactical condition (verification condition) satisfactory for a description to be admissible. The theory proposed by R.W.Floyd caused big interest of the computer science community and of a number of programmers, for it was one of first mathematical theories of the semantics of programs. In the paper [22] one can find the following result: a description is acceptable in a theory $\mathcal{T}$ iff it is admissible in all models of the theory $\mathcal{T}$

In the present section we shall present the concepts of Floyd in a slightly modified form. We used the result of L.Banachowski [6] and his suggestion to consider a structured programs. Besides the definition of a description, we shall use the term an annotated program, which denotes an linear expression corresponding to a graphical description.
Let us consider an example of division of integers.

Example 5.1.1 *(well known example)*

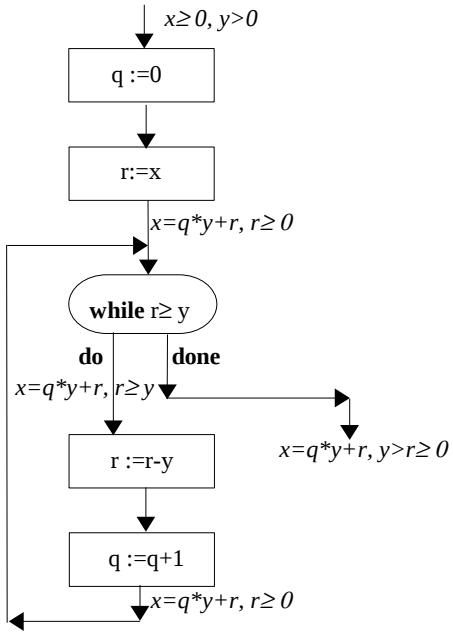


Figure 5.1: A description of a program 5.1.1

$q := 0;$
$r := x;$
while $r \geq y$
do
$r := r - y;$
$q := q + 1$
done

One can remark that during every execution of the program the following property holds: *whenever during the computation we pass along an edge then the actual state of memory satisfies the formula associated to the edge.* In particular, when the program ends its computation, the formula associated to the outgoing edge is satisfied by the results of the program. The formula of our example expresses the following property: the integer number q is the quotient of the integers x and y, and the number r is the remainder.

Definition 5.1.2 *By an* annotated program *we shall understand an expression defined by induction w.r.t. length of a program in the following way:*

a) Any expression of the form $\{\alpha\}\ [x := \tau]\ \{\beta\}$ *is an annotated program.*

Let M_1' *and* M_2' *be annotated versions of programs* M_1 *and* M_2.

b) the expression $\{\alpha\}$**if** γ **then** M_1' **else** M_2' **fi** $\{\beta\}$ *is the annotated version of the program* **if** γ **then** M_1**else** M_2 **fi**,

c) the expression $\{\alpha\}$ **while** γ **do** M_1' **od** $\{\beta\}$ *is the annotated version of the program* **while** γ **do** M_1 **od**,

d) the expression $\{\alpha\}$ **begin** $M_1'; M_2'$ **end** $\{\beta\}$ *is the annotated version of the program* **begin** $M_1; M_2$ **end**.

The formulas α *and* β *will be called respectively the pre-condition (or initial condition) and the post-condition (or terminal condition) of the description (also of annotated version) of the program.* ■

It is convenient to illustrate the concept of the description of program graphically. A description of an atomic program, i.e. an assignment instruction looks like on the Fig. 5.2. Let M1' and M2' be two annotated programs, c.f.

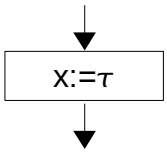


Figure 5.2: Description of assignment instruction

Fig. 5.3 Then the annotated programs of more complicated structure are as

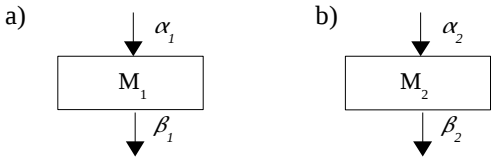


Figure 5.3: Two annotated programs

presented in the Fig. 5.4

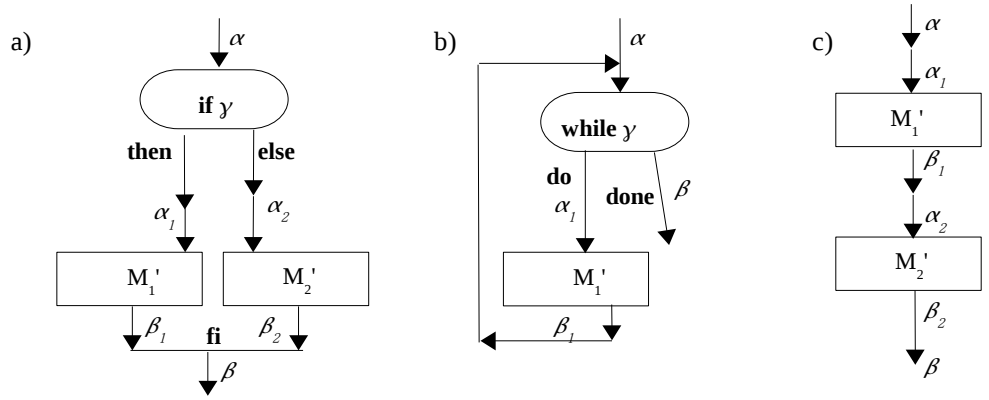


Figure 5.4: Descriptions of a) conditional instruction, b) iterative instruction, c) composed instruction

Definition 5.1.3 Verification condition *of an annotated program M' is a formula $VC(M')$ defined by induction in the following way:*

a) If M' is of the form $\{\alpha\}s\{\beta\}$ where s is an assignment instruction and α and β are any formulas then

$$VC(M') \stackrel{df}{=} (\alpha \Rightarrow s\beta).$$

Let M_1' and M_2' be annotated versions of programs $M, 1$ and M_2, let α_i be the pre-condition of the annotated program M_i', let β_i be the post-condition of the annotated program M_i' $\{i = 1, 2\}$.

*b) if M' is of the form $\{\alpha\}$**if** γ **then** M_1' **else** M_2' **fi** $\{\beta\}$ then*

$$VC(M') \stackrel{df}{=} VC(M_1') \wedge VC(M_2') \wedge ((\alpha \wedge \gamma) \Rightarrow \alpha_1) \wedge ((\alpha \wedge \neg\gamma) \Rightarrow \alpha_2) \wedge ((\beta_1 \vee \beta_2) \Rightarrow \beta).$$

*c) if M' is of the form $\{\alpha\}$**begin** $M_1'; M_2'$**end**$\{\beta\}$ then*

$$VC(M') \stackrel{df}{=} VC(M1') \wedge VC(M2') \wedge (\alpha \Rightarrow \alpha_1) \wedge (\beta_1 \Rightarrow \alpha_2) \wedge (\beta_2 \Rightarrow \beta).$$

*d) if M' is of the form $\{\alpha\}$**while** γ **do** $M1'$ **od**$\{\beta\}$ then*

$$VC(M') \stackrel{df}{=} VC(M_1') \wedge (((\alpha \wedge \beta_1) \wedge \gamma) \Rightarrow \alpha_1) \wedge (((\alpha \vee \beta_1) \wedge \Rightarrow \neg\gamma) \Rightarrow \beta).$$

■

Let us consider the following program M

Example 5.1.4 *(a program M)*
begin
while$(z-y)$¿0
do
$i := i+1;$
$z := z-y;$
$y := y+2$
od;
if$z = y$**then**$y := 0$**else**$y := z$**fi**
end

and its annotated version

Example 5.1.5 *(an annotated version of the program M)*
$\alpha_1:\ \{y = 1 \wedge z = x \wedge x > 0 \wedge i = 0\}$
begin
$\alpha_2:\ \{y = 2i+1 \wedge z = x - i^2 \wedge x > 0 \wedge i \geq 0\}$
while $(z-y) > 0$
do
$\alpha_3:\ \{z > y \wedge z = x - i^2 \wedge y = 2i+1 \wedge i \geq 0\}$
$i := i+1;$
$\alpha_4:\ \{z > y \wedge z = x - (i-1)^2 \wedge y = 2i-1 \wedge i \geq 0 \wedge x \geq 0\}$
$z := z-y;$
$\alpha_5:\ \{z > 0 \wedge z = x - (i-1)^2 - (2i-1) \wedge y = 2i-1 \wedge i \geq 0 \wedge x \geq 0\}$
$y := y+2$
$\alpha_6:\ \{z > 0 \wedge z = x - \Sigma_{j=1}^{i}(2j-1)(i-1)^2 - 2i - 1 \wedge y = 2i-1 \wedge i \geq 0 \wedge x \geq 0\}$
od;
$\alpha_7:\ \{z \leq y \wedge y = 2i+1 \wedge z = x - i^2 \wedge x > 0 \wedge i \geq 0\}$
$\alpha_8:\ \{x - i^2 \leq 2i+1 \wedge z = x - i^2 \wedge y = 2i+1\}$
if$z = y$
then
$\alpha_9:\ \{x - i^2 = 2i+1 \wedge z = x - i^2\}$
$y := 0$
$\alpha_{10}:\ \{y = 0 \wedge x = (i+1)^2\}$
else
$\alpha_{11}:\ \{i^2 \leq x < (i+1)^2 \wedge z = x = i^2\}$
$y := z$

α_{12} : $\{y = x - i^2 \wedge i^2 \leq x < (i+1)^2\}$
fi
α_{13} : $\{y = x - \lfloor\sqrt{x}\ \rfloor^2\}$
end
α_{14} : $\{y = x - \lfloor\sqrt{x}\ \rfloor^2\}$

and its description in graphical form (see Fig.8.7)
Fig.8.7

Definition 5.1.6 *A verification condition $VC(M')$ of an annotated program M' is* admissible *in a data structure $\mathbb{A}$ iff it is a formula valid in $\mathbb{A}$.* ■

Example 5.1.7 *The verification condition for the above given annotated program is the following formula:* $VC(M) = (\alpha_1 \Rightarrow \alpha_2) \wedge (\alpha_7 \Rightarrow \alpha_8) \wedge (\alpha_{13} \Rightarrow \alpha_{14}) \wedge (\alpha_5 \Rightarrow [y := y+2]\alpha_6) \wedge (\alpha_3 \Rightarrow [i := i+1]\alpha_4) \wedge (\alpha_4 \Rightarrow [z := z-y]\alpha_5) \wedge (((\alpha_2 \vee \alpha_6) \wedge z - y > 0) \Rightarrow \alpha_3) \wedge (((\alpha_2 \vee \alpha_6) \wedge z - y > 0) \Rightarrow \alpha_7) \wedge (\alpha_9 \Rightarrow [y := 0]\alpha_{10}) \wedge (\alpha - 11 \Rightarrow [y := z]\alpha_{12}) \wedge ((\alpha_8 \wedge z = y) \Rightarrow \alpha_9) \wedge ((\alpha_8 \wedge z \neq= y) \Rightarrow \alpha_{11}) \wedge ((\alpha_{10} \vee \alpha_{12}) \Rightarrow \alpha_{13})$. ■

Let us note, that the formula $VC(M)$ is a conjunction of thirteen implications, all of them have a simple structure. Every of these formulas describe certain node in the program's diagram. To every test node (a node of if or while type) correspond two implications (one may say, they describe it), one implication is associated with a node containing an assignment. Every implication expresses a simple semantically property of computations of program M: If a computation of program M reached a given node c and if the current configuration (a state) of the computation satisfies the condition given in the predecessor of the implication - this condition is an disjunction of the conditions associated to the edges leading to the node c - then after execution of the instruction (or test) contained in the node c, the new state of the computation satisfies the formula associated to the edge leaving the node c. In the case where c is a test node and two edges leave it, we have two implications. The above sentence can be repeated, appropriately modified, for every edge leaving a test node.

Lemma 5.1.8 *Let $\mathbb{A}$ be a data structure. If a verification condition $VC(M')$ of an annotated program M' is admissible in $\mathbb{A}$, then the program M is partially correct w.r.t. the formula α - pre-condition in the annotated program M' and the formula β - post-condition in M'.*
$\mathbb{A} \vDash VC(M')$ implies $\mathbb{A} \vDash ((\alpha \wedge Mtrue) \Rightarrow M\beta)$ i.e. the verification condition $VC(M')$ is valid in $\mathbb{A}$ iff the formula expressing the partial correctness

of the program M w.r.t. the pre- and post- conditions of M', is valid in $\mathbb{A}$. ■

Lemma 5.1.9 *The verification condition $VC(M')$ of an annotated program M' is admissible in the data structure $\mathbb{A}$ iff the formula $(aM \Rightarrow b)$ is valid in $\mathbb{A}$, where αM denotes (c.f 3.x.y) the strongest consequent of condition α w.r.t. the program M.* ■

Definition 5.1.10 *A verification condition $VC(M')$ of an annotated program M is* acceptable *in a theory $\mathcal{T} = \langle \mathcal{L}, \mathcal{C}, \mathcal{A} \rangle$ iff it is a theorem of the theory $\mathcal{T}$.*

Lemma 5.1.11 *For a given annotated program M its verification condition is acceptable in a theory $\mathcal{T}$ iff it is admissible in every model of the theory $\mathcal{T}$.* ■

The above lemma is in fact, a rather simple corollary from the completeness theorem (c.f. 4.x.y). The lemma has, however, deeper consequences, for it enables verification of partial correctness of a program through a proof of a number of implications, where each of them is easy to check.

A verification condition is a conjunction of certain number of implications. Every implication has a simple structure. Hence, most of them are easily provable or disprovable (by means of a counter-example). This feature caused certain popularity of Floyd's method. Two years later, C.A.R.Hoare [27] presented an axiomatic system for proving partial correctness of programs.

5.2 Hoare's logic of partial correctness

In 1969 C.A.R. Hoare [27] published a system of inference rules for reasoning about partial correctness of programs. The expressions of the language considered are: programs, formulas of first-order logic and triples of the form $\{\alpha\}\mathrm{M}\{\beta\}$ where α and β are first-order formulas and M is a program. The triple $\{\alpha\}\mathrm{M}\{\beta\}$ is a logical formula and expresses the property of program M being partially correct w.r.t. a precondition α and a post condition β. It is interpreted in the following way: for a given state v value of the triple is true iff either the precondition α is not satisfied by the state v or program M does not terminate a computation starting from v or program M terminates and the final state of the computation satisfies the post condition β. The reader can remark that the triple $\{\alpha\}\mathrm{M}\{\beta\}$ expresses the same property as the algorithmic formula $(\alpha \wedge \mathrm{Mtrue}) \Rightarrow \beta)$. From this remark and from the completeness property of deductive system of AL it follows that the Hoare's

calculus is included in AL. However, we hope that a presentation of the more detailed reasoning on that, may be instructive for the reader.
Let us begin from the presentation of the formal construction of Hoare's system.
AXIOMS

Axiom of assignment instruction

H1 $\{\alpha(x/\tau)\}\ [x := \tau]\ \{alpha(x)\}$

Axiom of instruction abort

H2 $\{alpha\}\ [\mathbf{abort}]\ \ \{\mathbf{false}\}$

Axiom of instruction leave

H3 $\{alpha\}\ [\mathbf{leave}]\ \{alpha\}$

Inference Rules

Rule of composed instruction

H4 $$\frac{\{\alpha\}[K_1]\{\alpha'\}, \{\alpha'\}[K_2]\{\beta\}}{\{\alpha\}[\mathbf{begin}\ K_1; K_2\ \mathbf{end}]\{\beta\}}$$

Rule of branching instruction

H5 $$\frac{\{\alpha \wedge \gamma\}[K_1]\{\beta\}, \{\alpha \wedge \neg\gamma\}[K_2]\{\beta\})}{\{\alpha\}[\mathbf{if}\ \gamma\ \mathbf{then}\ K_1\ \mathbf{else}\ K_2\ \mathbf{fi}]\{\beta\}}$$

Rule of iteration instruction

H6 $$\frac{\{\alpha \wedge \gamma\}[K]\{\alpha\}}{\{\alpha\}[\mathbf{while}\ \gamma\ \mathbf{do}\ K\ \mathbf{od}]\{\neg\gamma \wedge \alpha\}}$$

Rule of consequence

H7 $$\frac{\{\alpha' \Rightarrow \alpha\}, \{\alpha\}[K]\{\beta\}, \{\beta \Rightarrow \beta'\}}{\{\alpha'\}[K]\{\beta'\}}$$

We are going now to show that the axioms and rules H1 - H7 can be deduced with the help of algorithmic logic. Hence, every reasoning which makes use of Hoare's rules is also a correct reasoning in algorithmic logic.

Proofs of soundness of the Hoare's rules in AL.

It is known, c.f. Alagic and Arbib [2], that the rules H1 - H7 are semantically sound, i.e. from the validity of premises it follows the validity of conclusions, in every rule. Making use of the completeness theorem one can generally assert that the axioms and rules presented above can be formally proved in AL. We are going to construct and present these proofs.

PROOF OF THE AXIOM H1

Axiom H1 is an equivalent of the algorithmic formula $\alpha(x/\tau) \wedge [x := \tau]\mathbf{true} \Rightarrow [x := \tau]\alpha$, which follows directly from axiomAx_{18} of algorithmic logic. This axiom is stronger than the formula H1.

PROOF OF THE RULE H4

We are going to prove that from the following two premises $(\alpha_1 \wedge K_1\mathbf{true} \Rightarrow K_1\alpha_2)$, $(\alpha_2 \wedge K_2\mathbf{true} \Rightarrow K_2\alpha_3)$ one can deduce the formula
$\alpha_1 \wedge \mathbf{begin}\ K_1; K_2\ \mathbf{endtrue} \Rightarrow \mathbf{begin}\ K_1; K_2\ \mathbf{end}\alpha$.
Indeed, applying the rule R_2 of algorithmic logic to the second premise, we obtain $(K_1 \wedge K_1(K_2\mathbf{true}) \Rightarrow K_1(K_2\alpha))$. Using the first premise $(\alpha_1 \wedge K_1\mathbf{true} \Rightarrow K_1\alpha_2)$and applying laws of propositional calculus we infer $\alpha_1 \wedge K_1(K_2\mathbf{true}) \Rightarrow K_1(K_2\alpha)$. By the axiom Ax_{19} we obtain the needed formula.

PROOF OF THE RULE H7

We have to prove that the formula $\alpha' \wedge K\mathbf{true} \Rightarrow K\beta'$ has a proof from the three premises mentioned in the rule $H7$. From the premise $\alpha \Rightarrow \alpha$ we deduce the formula $\alpha' \wedge K\mathbf{true} \Rightarrow \alpha \wedge K\mathbf{true}$. Applying the rule R_2 we prove the formula $K\beta \Rightarrow \beta'$. The formula $\alpha \wedge K\mathbf{true} \Rightarrow K\beta$ is our second premise. Now, applying the axiom Ax_1 twice we obtain $\alpha' \wedge K\mathbf{true} \Rightarrow K\beta$, and next, the formula $\alpha' \wedge K\mathbf{true} \Rightarrow K\beta'$.

PROOF OF THE RULE H5

From the first premise we infer the formula

$$(\alpha \wedge \gamma) \wedge K\mathbf{true} \Rightarrow ((K\beta) \wedge \gamma)$$

From the second premise we obtain

$$((\alpha \wedge \neg\gamma) \wedge M\mathbf{true}) \Rightarrow ((M\beta) \wedge \neg\gamma.$$

Now, we apply the following law of propositional calculus

$$((\delta \rightarrow \sigma) \wedge (\delta' \Rightarrow \sigma')) \Rightarrow ((\delta \vee \delta') \Rightarrow (\sigma \vee \sigma'))$$

and obtain the formula

$$((\alpha \wedge \gamma) \wedge K\mathbf{true}) \vee ((\alpha \wedge \neg\gamma) \wedge M\mathbf{true}) \Rightarrow (\gamma \wedge K\beta \vee \neg\gamma \wedge M\beta)$$

From this formula in a few easy steps we deduce the formula

$$\alpha \wedge \mathbf{if}\ \gamma\ \mathbf{then}\ K\ \mathbf{else}\ M\ \mathbf{fitrue} \Rightarrow \mathbf{if}\ \gamma\ \mathbf{then}\ K\ \mathbf{else}\ M\ \mathbf{fi}\beta$$

which ends the proof the rule R_5.

PROOF OF THE RULE H6

This rule may be proved on many ways. Dijkstra proves it in his book[]. We gave another proof of this secondary but very useful rule in the example 4.1.12.

Warning

After M. O'Donnell we would like to call the attention of the reader that it is somewhat easy to turn this system into an inconsistent one.

On the relative completeness of Hoare's logic

The system composed of axioms and rule $H1 - H7$ is an incomplete deduction system. There are properties valid but not provable. In 1974 S. Cook proposed to extend the system by adding to it all formulas of the first order language valid in the data structure $\mathbb{A}$.

5.3 Dijkstra's calculus of weakest preconditions

In 1974 E.W.Dijkstra [17] presented a calculus of so called predicate transformers. In his considerations the author of [17] starts from a remark that programs can be conceived as transformers of conditions i.e. predicates satisfied by states of computations. In particular he studied the weakest precondition. In a book[18] which appeared a year later he presented his opinions and several examples of semantic analyses of properties of programs, in which the axioms of weakest precondition are used. In this chapter we present the Dijkstra's calculus adapted to the language of deterministic, iterative programs c.f. chapter 2. The informal description of the weakest precondition in [18] reads as follows: *a condition which characterises the set of all initial states, such that the beginning of computations in any of the states will certainly lead to a successful termination in a state satisfying given terminal condition, is called the weakest precondition for a given post condition.* In the papers of Dijkstra there is no other definition of the notion. Instead, the author gives the properties which are to be satisfied by the weakest precondition. We are calling the attention of the reader to the definition of this notion we gave in 3.3.5. Accordingly, we shall replace the Dijkstra's formulas $wp(\alpha, \mathrm{K})$ by algorithmic formula $\mathrm{K}\alpha$. After the translation to the language of algorithmic logic the properties of weakest precondition obtain the following form.

Pr_1) $M\mathbf{false} \Leftrightarrow \mathbf{false}$

Pr_2) $\dfrac{(\alpha \Rightarrow \beta)}{(M\alpha \Rightarrow M\beta)}$

Pr_3) $M(\alpha \wedge \beta) \Leftrightarrow (M\alpha \wedge M\beta)$

Pr_4) $M(\alpha \vee \beta) \Leftrightarrow (M\alpha \vee M\beta)$

Pr_5) In the book [18] this property is formulated as follows: *for any program M and for every infinite sequence of formulas $\alpha_0, \alpha_1, \alpha_2, ...$, such that for all states and for every number $r \geq 0$ the following implications hold*

$$(\alpha_r \Rightarrow \alpha_{r+1}),$$

then for every state, the following equivalence holds

$$M((\exists_{r\geq 0})\alpha_r) \Leftrightarrow ((\exists_{s\geq 0})M\alpha_s).$$

Here we encounter certain problem when we wish to express this property in the language of algorithmic logic. The difficulties come from the fact, that the author unconsciously made the wording easier for himself, confusing the notions of quantifier and of infinite disjunction. In the language of algorithmic logic there is no infinite disjunction of formulas. Although the notation $(\exists_{r\geq 0}\ \alpha_r)$ admitted by Dijkstra is confusing, its semantic meaning is clear: the whole formula is satisfied if there exists an $r \geq 0$, such that the formula α_r is satisfied. The r-th formula in the sequence $\alpha_1, \alpha_2, ...$ Hence, one should assert that we are doing with infinite disjunction and write

$$M(\bigvee_{r\geq 0} \alpha_r) \Leftrightarrow (\bigvee_{s\geq 0} M\alpha_s).$$

We can remark, moreover that the index r need not to occur in any of the formulas of the sequence, it is just a number of a formula. In the language of first-order logic a correct formula of similar form is $(\exists_{r\geq 0}\alpha(r))$, but it has completely different meaning and has no connection with sequence of formulas we were talking about in the premise of the property Pr_5.

This leads to the conclusion, that property Pr_5 is a rule of inference with the infinitely many premises. It should read as follows

$$\frac{(\alpha_r \Rightarrow \alpha_{r+1})_{r\in N}}{M(\bigvee_{r\geq 0} \alpha_r) \Leftrightarrow (\bigvee_{s\geq 0} M\alpha_s)}$$

in other words if for every natural number r, the implication $(\alpha_r \Rightarrow \alpha_{r+1})$ is valid then the equivalence $M(\bigvee_{r\geq 0} \alpha_r) \Leftrightarrow (\bigvee_{s\geq 0} M\alpha_s)$ is valid also.

The author discusses in turn program connectives appearing in the programming language and gives further properties of the weakest precondition. He views them as axiomatic definitions of meaning of corresponding programming constructions of the language. Without loss of generality we

can replace the language of guarded commands introduced by the author by the language of iterative, deterministic commands. As atomic programs we shall have the assignment instructions, and two instructions **abort** and **do-nothing**. One could define the instructions abort and do-nothing by the more complicated ones. We shall keep however the notation as useful in our future considerations.

Apart of above enumerated, five properties of the weakest precondition, the author gives six additional schemes of axioms. The goal which they are to reach is to define the semantics of the programming connectives which appear in the programming language.

A1) $\mathbf{do-nothing}\ \alpha \Leftrightarrow \alpha$

A2) $\mathbf{abort}\ \alpha \Leftrightarrow false$

A3) $[x := \tau]\ \alpha \Leftrightarrow \alpha(x/\tau)$

A4) $\mathbf{begin}\ M_1; M_2\ \mathbf{end}\ \alpha \Leftrightarrow M_1(M_2\alpha)$

A5) $\mathbf{if}\ \gamma\ \mathbf{then}\ M_1\ \mathbf{else}\ M_2\ \mathbf{fi}\ \alpha \Leftrightarrow ((\gamma \wedge M_1\alpha) \vee (\neg\gamma \wedge M_2\alpha))$

A6) for the iteration instruction Dijkstra gives the following definition of the weakest precondition

$$\mathbf{while}\ \gamma\ \mathbf{do}\ M\ \mathbf{done}\ \alpha \Leftrightarrow (\exists_{k\geq 0})H_k(\alpha)$$

where the formulas H_k are defined by induction, as follows:

$$H_0(\alpha) \overset{df}{\Leftrightarrow} \alpha \wedge \neg\gamma \tag{5.1}$$

$$H_{k+1}(\alpha) \overset{df}{\Leftrightarrow} (\mathbf{if}\ \gamma\ \mathbf{then}\ M\ \mathbf{fi})H_k(\alpha) \vee H_0(\alpha) \tag{5.2}$$

Let us closer examine these formulas. Formula H_0 is beyond suspicion. Let us compute the formula H_1, applying A5) and simple transformations of predicate calculus, we see that

$$\begin{aligned} H_1(\alpha) &\Leftrightarrow (\mathbf{if}\ \gamma\ \mathbf{then}\ M\ \mathbf{fi})(\alpha \wedge \neg\gamma) \vee (\alpha \wedge \neg\gamma) \\ &\Leftrightarrow ((\gamma \wedge M(\alpha \wedge \neg\gamma) \vee \neg\gamma \wedge (\alpha \wedge \neg\gamma)) \vee (\alpha \wedge \neg\gamma) \\ &\Leftrightarrow ((\gamma \wedge M\neg\gamma \wedge M\alpha) \vee (\neg\gamma \wedge \alpha)) \end{aligned}$$

In general, the formula H_k is expressed as more and more lengthy disjunction of conjunctions which also grow in length.

$$\begin{aligned} H_k(\alpha) &\Leftrightarrow ((\neg\gamma \wedge \alpha) \vee (\gamma \wedge M\neg\gamma \wedge M\alpha) \vee \\ &\quad \ldots (\gamma \wedge M\gamma \wedge MM\gamma \wedge \cdots \wedge M^{k-1}\gamma \wedge M^k\neg\gamma \wedge M^k\alpha)) \\ &\Leftrightarrow (\mathbf{if}\ \gamma\ \mathbf{then}\ K\ \mathbf{fi})^k(\neg\gamma \wedge \alpha) \end{aligned}$$

.

Remark 5.3.1 *The subformula $H_0(\alpha)$ in the above definition 5.2 is redundant and may be ommitted*

$$H_{k+1}(\alpha) \overset{df}{\Leftrightarrow} (\textbf{if } \gamma \textbf{ then } M \textbf{ fi}) H_k(\alpha). \qquad \blacksquare$$

Let us remark, first of all, that every of these formulas has a different structure and that the formulas H_k do not contain the variable k at all. The usage of quantifier notation $(\exists_{k\geq 0} H_k(\alpha))$ should be treated as an informal denotation of the infinite disjunction of formulas H_k, moreover, that every such formula has a different structure. Let us recall, in the first-order predicate calculus the expression of the form $(\exists x)\alpha(x)$ is a formula of the language if $\alpha(x)$ is a formula. Can we apply this simple grammatical rule to the expression $(\exists_{k\geq 0} H_k(\alpha))$? No, it is not a formula. Here the formulas and their denotations are mixed. The expression H_k belongs to the metalanguage. Putting our remark in other words: the formula $(\exists x)\alpha(x)$ has a value equal to the value of the least upper bound of values of formulas $\alpha(x/\tau)$ where τ is any term (or if you prefer an arithmetic expression). Every of these formulas has the same logical structure, the same number of logical connectives and quantifiers in it, however they can differ in length since the terms t can be of different structure. In the case of formulas H_k their logical structure changes with the growth of k. Therefore, application of quantifier notation is not allowed in this case. And how to do this when the variable k does not occur in the formulas H_k?

We would like to call the reader's attention to the fact that the semantic meaning of the Dijkstra's axiom A_6 is faithfully expressed by the following formula

$$\textbf{while } \gamma \textbf{ do } K \textbf{ done } \alpha \Leftrightarrow \bigcup \textbf{if } \gamma \textbf{ then } K \textbf{ fi } (\alpha \wedge \neg\gamma). \qquad (5.3)$$

Validity of this formula has been shown in lemma 3.4.2. By the completeness theorem we know that the formula has a proof from the axioms of algorithmic logic.

Proof. In the proof of 5.3 we use the axiom Ax_{21} and rules R_3 and R_4 of algorithmic logic. It follows from the axiom Ax_{21} that for every natural number i

$(\textbf{if } \gamma \textbf{ then } \mathrm{K} \textbf{ fi})^i(\neg\gamma \wedge \alpha) \Longrightarrow \textbf{while } \gamma \textbf{ do } \mathrm{K} \textbf{ done } \alpha.$

A proof of this fact we gave in example 4.1.5. Applying the rule R_4 we derive the implication

$\bigcup \textbf{if } \gamma \textbf{ then } \mathrm{K} \textbf{ fi } (\alpha \wedge \neg\gamma) \Longrightarrow \textbf{while } \gamma \textbf{ do } \mathrm{K} \textbf{ done } \alpha.$

The reverse implication is also provable. For every natural number i the

following formula is a tautology
(**if** γ **then** K **fi**)$^i(\neg\gamma \wedge \alpha) \Longrightarrow$ (**if** γ **then** K **fi**)$^i(\neg\gamma \wedge \alpha)$.
By axiom Ax_{22} and by propositional calculus we infer that for every natural number i the following implication is a tautology
(**if** γ **then** K **fi**)$^i(\neg\gamma \wedge \alpha) \Longrightarrow \bigcup$ **if** γ **then** K **fi** $(\neg\gamma \wedge \alpha)$
Now, we can apply the rule R_3 and obtain
while γ **do** K **done** $\alpha \Longrightarrow \bigcup$ **if** γ **then** K **fi** $(\neg\gamma \wedge \alpha)$. ■
It is easy to remark that the formulas Pr_3, Pr_4, A_3, A_4, A_5 are known already axioms of AL. One can ask whether the Dijkstra's formalism and algorithmic logic are equivalent. It turns out that all properties listed by Dijkstra can be proved from axioms of AL. Moreover the property Pr5 of continuity is provable in an extension of AL which admits the infinite disjunction and conjunctions. Without any additional assumptions we prove:

$M(\bigvee_{r\geq 0} \alpha_r) \Leftrightarrow (\bigvee_{s\geq 0} M\alpha_s)$

Assume that for a certain valuation v the formula $M(\bigvee_{r\geq 0} \alpha_r)$ is satisfied by v. Then by the definition of semantics of algorithmic formulas the formula $\bigvee_{r\geq 0} \alpha_r$ ar is satisfied by the valuation $M_{\mathbb{A}}(v)$ - the result of program M. But the formula $\bigvee_{r\geq 0} \alpha_r$ is satisfied if for a certain $s \geq 0$ the formula α_s is satisfied. We have that for a certain s the valuation $M_{\mathbb{A}}(v)$ satisfies $\bigvee_{s\geq 0} \alpha_s$. Hence the valuation v satisfies $\bigvee_{s\geq 0} M\alpha_s$. The other implication is provable in a similar way.

Our next remarks concern the property Pr_1 called by Dijkstra the law of excluded miracle. Since the implication false $\Longrightarrow$M false is a tautology, what remains to be proved is (Mfalse)=>false. Let us recall, that $(\alpha \wedge \neg\alpha) \Leftrightarrow$ false. Note, that for every program M and every formula a, holds

M $(\alpha \wedge \neg\alpha) \Leftrightarrow$ Ma $\wedge$ M$\neg\alpha$ (axiom Ax14).

By another axiom M$\neg\alpha \Longrightarrow \neg$Ma. Hence

M $(\alpha \wedge \neg\alpha) \Longrightarrow$ Ma $\wedge \neg$Ma

Applying the axiom $(\alpha \wedge \neg\alpha) \Longrightarrow \beta$ we derive the implication M false $\Longrightarrow$ false what finishes the formal proof of the law of excluded miracle.

Let us return to the properties Pr_2 and Pr_5, they are inference rules. The property Pr_5 of continuity has infinitely many premises. We have shown that the premises are not necessary for the distributivity of program over infinite disjunction. However the infinite disjunction itself must be characterised in some way. It is not difficult to guess that for this, one needs a rule with infinitely many premises. In an extended algorithmic logic one can derive the

equivalence A_6 from axiom $Ax_2 1$ and rule R_3. We do not think however that such extension is needed. It seems redundant. In the light of our previous considerations, we see that semantic properties of programs can be expressed and studied in a language of finite expressions.

Another question arises: *is the statement of Dijkstra that properties Pr_1 - Pr_5 and axioms A_1 - A_6 define the semantics well argued?* D.Harel [24] has studied the question and came to the conclusion that among many possible strategies of visiting trees of non-deterministic computations only one method of visiting satisfies the axioms of Dijkstra, and therefore, concludes Harel, the axioms of Dijkstra define the semantics of non-deterministic computations. For us the problem was not definitely solved, for where the trees of computations are coming from? This was a motivation for the research, the results of which are presented in section 4.4. The results of 4.4 present the stronger consequences of admitting algorithmic axioms. The meaning of the programming connectives: composition, branching, iteration and also of atomic instructions (assignment) is uniquely determined by the requirement that a realisation of the language satisfied the axioms of AL. In the present moment we do not know whether a similar result can be proved for non-deterministic computations.