Nondeterministic programming within the frame of first order classical logic, Part 1

By T. GERGELY and L. ÚRY

1. Introduction

1.1 Nondeterminism in computer science

In computer practice a lot of phenomena have arisen that deviate from the deterministic attitude forming the base of traditional programming. These nondeterministic phenomena may be due to varying reasons.

Considering the reasons three main types of nondeterminism can be distinguished. The first type of nondeterminism is quite independent from the will of programmers and its causes are hidden in the construction and functioning of computers. Due to this nondeterminism almost every program has some uncertainty while execution. These could be caused by power cut, current trouble, machine break-down or by any other unforseeable reason. If the computer works in time-sharing mode the uncertainty further increases and the behaviour of a program will depend on the other programs executed alternately with it. Moreover it depends on the memory requirements of the programs, on the number of peripheries at disposal, etc. In computers allowing parallel computations further causes of uncertainty interfer, namely the speed difference of certain processes and communication, the noise level of the communication channel, the concurrency for resources etc. In interactive mode another type of uncertainty is caused by the randomness of interactions affecting the program under execution.

We may call *probability programming* the methods that consider the above uncertainties and its theory should be based on the usage of the tools of mathematical statistics and those of theory of probability. Random events occuring in program execution are handled by these tools. In this type of programming the commands do not have a uniquely defined result, only its distribution is known. Thus the running of a program can be described by using stochastic process e.g. by using either Markov or semi-Markov chains. One of the main aims of the theory of such type of programming is to minimalize the expectable number of failures.

The second type of nondeterminism is already connected with the programmer's will. The programmer's attitude is still deterministic, but he uses probabilistic methods containing well defined randomness in the solution of some tasks. A widespread method of this type is connected with the use of random number generator. The programming style using this method is deterministic and it also supposes the determinism of the computer, however, for the solution of certain tasks it uses one of the Monte-Carlo methods.

The third type of nondeterminism is connected with the *essential* and *logical* uncertainty enclosed in the solutions of tasks. It embodies two kinds of uncertainties. The first one occurs in such a situation of problem solving when there are more alternat ives selecting from which any, the result will be produced without difficulty. The second kind of uncertainty is connected with such a situation where only certain alternatives lead to correct results but, in advance, we do not know which one. A programming style which considers the above mentioned two kinds of uncertainties suggests a nondeterministic attitude in contrast with the deterministic one of the traditional style of programming. The main difference between these two kinds of attitudes is that the nondeterministic one considering different kinds of choices does not specify how to make them though the deterministic attitude does not leave the question how to make choices (if there are any) unspecified. Programming theory connected with the third type of nondeterminism is in the focus of our further investigations.

1.2 Some reasons of interest in nondeterministic programming

Recently nondeterministic programming has attracted more and more attention. It provides the programmers to concentrate on some important questions about deterministic programs without specifying details irrelevant to the questions to be analysed. Thus this programming attitude can be used to reason about deterministic programs.

The possibility to consider actions not describing their details makes the nondeterministic programming very useful in the field of Artificial Intelligence, e.g. in natural language understanding, in problem solving, in robot planning, etc. E.g. it suggests a fractional method of problem solving or robot-planning as follows. First a global algorithm — a "global plan" can be designed as a nondeterministic program, then, by analyzing this program and completing it with appropriate parts we get a concrete deterministic program, i.e. a complete detailed algorithm to solve the task.

One of the most significant reasons why nondeterministic programming becomes more and more important is that it plays a significant role in the elaboration of the theory of interactive and parallel programming. Most of its applications are connected with this area. See e.g. HOARE (1978), MILNER (1973), OWICKI and GRIES (1975), PLOTKIN (1976), etc.

In view of aboves it is quite natural that nondeterministic programming plays an increasing role in both the theory and practice of programming.

The aim of our investigation in the present work is the elaboration of a mathematical theory of nondeterministic programming which can handle both syntax and semantics by using mathematical tools and provides tools to speak about program properties and to prove them. The elaboration of this theory will be done by using the approach developed in GERGELY and ÚRY (1978).

1.3 Some words on our approach

The essence of the programming situation to be considered is that beside programming language such a new language arises that is suitable to describe the properties and the meaning of programs and our expectations towards programs in an unambiguous way. Thus this new language is a descriptive one in contrast with the programming language which serves to give instruction, i.e. commands. To assure unambiguity the descriptive language should have well defined and exact semantics beyond the syntax. The syntax should be suitable to describe program properties and with the power of proof to analyse whether certain features of programs correspond to the expectations given by the specification, i.e. to analyse the correctness of programs. The semantics of the descriptive language should provide unambiguous understanding of the meaning of programs, hence it has to be compatible with the semantics of programming language. There are two main possibilities to give exact semantics. The first is to characterize programs according to the question "*what* the program does?" the second is to do it according to the question "*how* the programs do it?"

In programming theories we find the following three approaches for the exact handling of semantics: *operational, functional* and *resultative.* The first aims to give a direct answer to both questions to *what* and *how.* The resultative or, in other words, axiomatic semantics neglects the question *how* and characterizes only the main properties of the change of data environment of the program produced while execution. Functional or, in other words, denotational semantics also gives the meaning of programs answering both questions, though it does it in an indirect way.

We wish to elaborate such a theory of nondeterministic programming which would be also a useful base for developing the mathematical theory of interactive and parallel programming. In order to understand the main features of interaction and parallelism in detail the execution processes themselves are to be considered. This permits to follow up the specific features connected with the mutual effects of the processes (e.g. communication, interaction). Thus operational semantics seems to be adequate to our aim. To have this type of semantics first the question *what* has to be answered by the characterization of changes in data caused by the execution of program and, secondly, the flow of programming processes in time has to be described to have an answer to the question *how.*

Thus the theory of programming to be developed has to have such a descriptive language that is capable of describing and characterizing both the data environment of programs and the time related to program execution. The first requirement is quite familiar with nearly every theory of programming, but not so is the time consideration, for programming theories do not consider time explicitly except for some of the most recent works.

Of course in the case of sequential programming the time aspects can be characterized through the change of data without considering time explicitly. However this approach is not applicable in those cases where time plays a primary and independent role as e.g. in interactive, real time and parallel programming. Therefore the programming theory to be developed here will contain tools that also provide explicit time consideration.

1.4 The role of classical first order logic in a theory of programming

To develop a mathematical theory of programming that corresponds to our aim the first problem is to introduce such a descriptive language that satisfies the above mentioned expectation concerning the characterization of time and data and it has to have exact and well defined semantics and appropriate tools to prove different statements about the program properties. In the case of sequential deterministic programming the first order classical language was quite satisfactory to be a descriptive language for the corresponding theory of programming as it was shown in GERGELY and ÚRY (1978) where the frame of classical first order mathematical logic was used to develop the theory of programming.

In the present work we show that the above mentioned logical frame is sufficient to develop the corresponding theory and the first order language can be used in the role of the descriptive one for the case of sequential nondeterministic programming. Why do we prefer the classical first order language? Because

— it has a well defined exact and transparent syntax and semantics;
— it has a special branch, the model theory with very strong mathematical methods to investigate semantics;
— it has a well developed proof theory that offers effective notion of proof and effective tools and methods for proving;
— it is currently used in the research practice so its use is fairly familiar;
— it is the simplest one of the languages of mathematical logic by which a programming language can be investigated since the propositional language is not suitable for this.

So it is justified to try to elaborate the mathematical basis of programming theory within the frame of classical mathematical logic that is the most highly developed branch of mathematical logic. This is encouraged by the fact that data environments of programs can be given without major restriction of generality by first order language.

In this work the mathematical foundations of programming theory, and the elaboration of the theory itself is done by strictly keeping to the frame of first order logic.

In the theory both date and time will be explicitly discussed by using first order language.

1.5 A short survey

Nondeterministic programming is mainly used in the area of Artificial Intelligence and in the investigation of parallel computation as it has already been mentioned. In connection with the first area MANNA (1970) introduces a nondeterministic programming language which is very similar to the language to be introduced here. It contains both kinds of choices, but it does not allow the description of time conditions.

Several works are devoted to the nondeterminism in connection with parallel computation. In the early work of ASHCROFT and MANNA (1970) parallelism has already been explained in terms of nondeterminism. Milner handles nondeterminism by using oracles. In the case of two computing processes executed parallelly an oracle is such an infinite sequence of 0 and 1 that be however far contains both elements 0 and 1. By this oracle the execution of two parallelly computed processes can be described so that 0 and 1 denote which process is at work. The theory using oracles is described in MILNER (1973) and (1978).

A very elegant mathematical theory of the same handling of nondeterminism is developed in PLOTKIN (1975). Developing the theory of programming both MILNER and PLOTKIN use denotational description of semantics of nondeterministic programs. EGLI (1975) also uses this type of the description of semantics.

An axiomatic definition of semantics is given in OWICKI and GRIES (1976).

The parallel programming containing nondeterminism suggests to introduce effective nondeterministic elements into the language. The very simple command *choice* S_1, S_2 has been replaced by the guarded commands introduced in DIJKSTRA (1975). In HOARE (1978) and FRANCEZ et al. (1978) input-output commands are added to the guarded ones. Analogical commands are also introduced in MILNER (1978). Not depending on the aboves we mention the work of HAREL and PRATT (1978) where the execution of programs is supposed to be ambiguous and in the descriptive language modalities are introduced in order to handle the ambiguity. Thus by this descriptive language such statements can be expressed that "there exists such a run...", "every run is such...". In this work the semantics is operational. An analoguous descriptive language is developed in MIRKOWSKA (1978) within the frame of algorithmic logic. Summarizing aboves we would like to emphasize that so far no work has been engaged in using tools to describe time conditions explicitly. The problem of completeness is discussed only in HAREL and PRATT (1978) and MIRKOWSKA (1978). The previous shows that the introduced descriptive language is complete with respect to arithmetics, in the latter it is proved that the nondeterministic algorithmic logic is ω-complete. We note that these two results are really equivalent.

1.6 What is new and the contents of the work

The theory of nondeterministic programming is developed strictly within the frame of first order classical logic. The semantics of nondeterministic programs is described in an operational way by using a special type of games. A descriptive language to describe program properties is introduced by using the classical first order language. This descriptive language allows to describe and to speak explicitly about both time and data.

Moreover the question of completeness is discussed and a complete calculus in the spirit of Floyd and Hoare is introduced.

The first part including the first three sections contains the conceptual and mathematical base providing exact tools to handle nondeterminism. Thus the next section is devoted to the main tool of our theory to a special type of games. Section 3 contains the main notions of classical first order mathematical logic and arithmetic and the representation of data and time in the frame of first order logic. Here first of all the description of time properties is discussed in details. In Section 4 the basic notions and properties of games are introduced within the frame of first order logic. A very simple but powerful enough nondeterministic programming language is introduced in Section 5. Its semantics is given by using associated games. With respect to nondeterministic programs many different questions can arise. Some of them are given in Section 6. Here we immediately show that an adequate descriptive language is needed to answer the questions for each one. Selecting two questions in connection with partial and total correctness we give the appropriate descriptive language and show that it is complete. In Section 7 we introduce a calculus which is analoguous to that of introduced by Floyd and Hoare for the sequential deterministic programming (see details in Gergely and Úry (1978)). In Section 8 we illustrate the use of the calculus by some examples. Present paper consists of two parts. The first one contains the first three sections.

1.7 Basic conventions

We use basic notations and concepts of the naive set theory in the usual fashion. The notation $\{x|\varphi(x)\}$ denotes the set of all x such that $\varphi(x)$. Both inclusion and proper inclusion are denoted by the same symbol $\subset$. The empty set is denoted by $\emptyset$. In the case of natural numbers for ordered finite set we use intervals defined by $[i, j]\overset{\mathrm{d}}{=}\{k\,|\,i\leqq k\leqq j\}$. The *domain* and *range* of a function f are denoted by do f and rg f respectively. $f\colon A\to B$ denotes that f is a function for which do $f=A$ and rg $f=B$. A function $f\colon A\to B$ is *injective* if for any $a, b\in A$ if $f(a)=f(b)$ then $a=b$. It is called *bijective* if it is injective and $f(A)=B$.

The symbol $\langle s_i\rangle_{i\in I}$ denotes a function f with domain I such that $f(i)=s_i$ for all $i\in I$. Such a function f is called *sequence*.

For a non-empty set A let A^+ denote the set of all finite non-empty sequences formed from the elements of A. ${}^AB$ denotes the set of all functions from A to B. ω is the least infinite ordinal. $|A|$ denotes the cardinality of the set A. Moreover for informal logic we use "iff" for "if and only if" and w.r.t. for "with respect to".

The end of significant units like proofs, definitions etc. is marked by the symbol $\square$.

2. Games

2.1 More about nondeterminism

As we have seen the uncertainty in nondeterministic programming is caused by two kinds of choices. The first one: from the alternatives one chooses such a possible step of a task solution that leads to the result. The other one is when each one of the alternative steps may be chosen and the result thus can be reached.

Having a nondeterministic program its execution can be so imagined that there is someone who represents the interests of the program, say Mr. A. He is the one who makes the first kind of choices. In opposition there is someone else, say Mr. B, representing the circumstances influencing the program execution. He is the one who makes the other kind of choices without being influenced by the interests of the program.

Thus we have a situation analoguous to a game situation where two players A and B are playing. Player A has to choose so that whatever B chooses the course of the game should favour A. This analogy suggests the games to be a useful and easily handable tool for our investigation. The games are very close to our intuition because they are widespread. At the same time they clearly represent essential nondeterminisms e.g. that of due to the uncertainty of players in the moves of one another. This uncertainty is quite analoguous to that of the nondeterministic programming. Thus we use an appropriate type of games as the main tool in our theory of nondeterministic programming.

Now let us consider what type of game is adequate to represent the nondeterminism of nondeterministic programming.

According to the aboves we say that the rules define the circumstances within the frame of which a game can be played, i.e. they define the *game-frame.* A game-frame still does not possess goals for the players though in the type of games used here the goals are well-defined for both players. E.g. such a goal can be either to win, or not to lose. Having goals players aspire to win or not to lose within a given game-frame. Games considered here are antagonistic in the sense that players try to achieve in fact two opposite goals. Beside the goals such rules are to be introduced that specify the *conditions* by which one of the players wins, or loses or the play is draw. These conditions can be defined by the appropriate set of those situations (or states) that provide the winning (or not losing) of one of the players. In this case to achieve his goal the player is to reach the winning situations, i.e. he has to make moves providing the appropriate states. The improvement of the positions of one of the players at the same time is spoiling the positions of the other one. Another possibility is to give a rule specifying a payment function which renders a payment to each possible state. This payment is to be received by one of the players and is payed by the other one and its sum depends on the situation. Moreover the gain of one of the players and the loss of the other one is equal but opposite in sign. Now the winning of each player means to receive the greatest payment. Thus the goal of a player is to maximize the payment he receives.

If we add rules describing the winning conditions to the game-frame we get the corresponding *game.* It is obvious that with respect to a given game-frame a lot of games can be defined, which only differ in the winning conditions.

2.2 Basic notions of games

Let us consider such a type of game that presumes two players and possesses well-defined rules for each of them. A game presupposes a sequence of moves, each of which is an occassion for a choice between certain alternatives.

The rules of the game specify for each move which player does it and what his alternatives are. These rules are finitely describable, and are to be known by each player. At each move, the player precisely knows what his alternatives are and his choice will become immediately known to the other player. Moreover each player precisely knows what moves, i.e. what choices have been made previously. Thus the players have full information about what has happened in the game so far and what else can ever happen during the course of it. For the latter the rules are to specify that no choice can be made by chance (e.g. by a dice). This means that each move is deterministic in such a sense that the situation formed after having the moves is foreseen in a unique way. A *course* of game contains a complete sequence of choices (moves) made by the players.

Thus the type of games used here consists of the rules that define the circumstances of playing and of the goals and rules defining the winning conditions.

To illustrate the abovesaids let us consider the following version of the well-known game NIM. First of all let us see its frame. There is a single pile of chips containing e.g. 21 chips and there are two players A and B. The two players take turns one after the other picking up chips from the pile. At each move, a player must take at least one chip and at most three ones. This is the game-frame. If we fix the winning circumstances then we get the game of the game-frame. Let us suppose that the player who picks up the last chip loses and thus we have got a game. Note that this kind of NIM game is often called Last One Loses. Of course we can define an opposite game with respect to the same frame, namely we add the following rule: the player picking up the last chip wins. By adding another winning condition we get a new game. There are a lot of other possibilities.

A game-frame graphically can be represented by a tree. The nodes of the tree correspond to the situations involved in the game. The arcs emanating from a given node are the alternatives associated with the corresponding move. A tree representing all the possible moves of both players and all the possible corresponding situations is called a *game-tree* or an *and/or-tree.* A *path* of the game tree represents a play of the corresponding game-frame. The winning condition can be represented by a set of paths per players leading to winning. Thus a tree represents a game-frame. Marking out the winning paths of both players we get the tree representation of a game of the given game-frame.

Note that the representability of a game by a tree means that the game is of full information, i.e. the players have full information about the course of the game because each node of the tree includes the history of its acces since each node, except the root, has exactly one predecessive node.

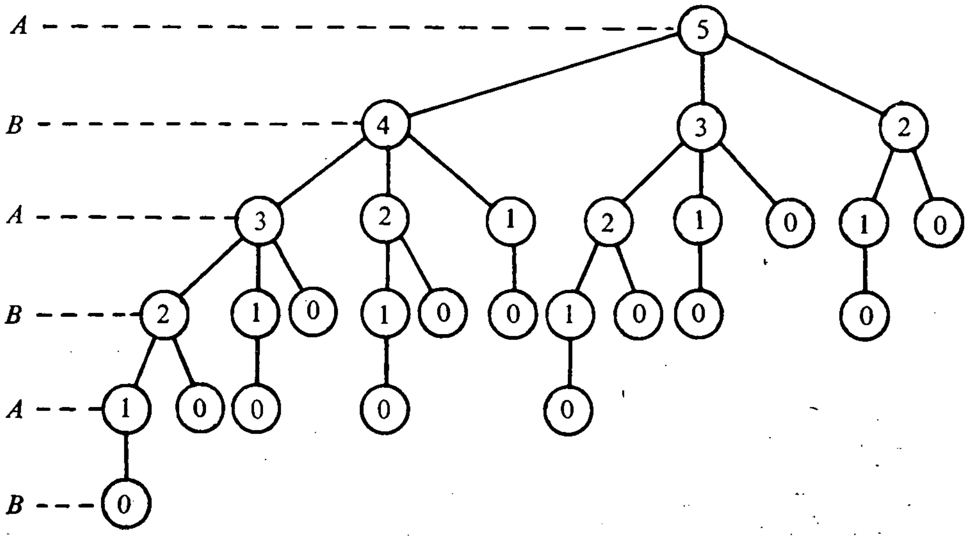

Fig. 1

To illustrate the abovesaids let us see the game-tree (Fig. 1) of the game Last One Loses for the case when the players begin with five chips in the pile. The nodes are labelled by the number of the remained chips in the pile. At alternate levels of depth in the tree, alternate players choose which move to make. To be definite we suppose that player A moves first. Each depth level is labelled by the name of the player who has the next choice at that level.

Since the so far mentioned type of game is a basic means of the theory of programming to be developed and since we wish to execute the investigation within a mathematical frame it is necessary to provide the mathematical definition of the basic notions of the games. To introduce games as mathematical objects we use their tree representation.

Let N be the set of natural numbers and let N^* denote the set of all finite sequences consisting of the elements of N. Λ denotes the empty sequence.

Let us take the following functions:

$$\begin{aligned} &pair\colon && N^* \times N \to N^* \\ &left\colon && N^* \setminus \{\Lambda\} \to N^* \\ &right\colon && N^* \setminus \{\Lambda\} \to N \end{aligned}$$

$$length(v) = \begin{cases} 0 & \text{if } v = \Lambda \\ length(left(v)) + 1 & \text{otherwise} \end{cases}$$

for any $v \in N^*$.

Intuitively speaking by the use of the function *pair* we can construct a new sequence if we add a natural number to a given sequence from the right side. The functions *left* and *right* provide the decomposition of sequences.

Definition 2.1. A set $V \subset N^*$ is said to be a *tree* iff the following properties hold:

(i) $\Lambda \in V$,

(ii) if $v \in V$ and $v \neq \Lambda$ then $left(v) \in V$. □

Example 2.2. Let us consider the following tree in graphical representation shown in Fig. 2.

According to our definition this can be represented as the following tree: $\{0, 01, 014, 0148, 015, 02, 026, 0269, 03, 037\}$, where 0 stands for Λ.

The graphical representation of this tree is shown in Fig. 3. □

Let $v, w \in V$. If $left\ (w) = v$ then w is a *successor* of v and v is a *predecessor* of w in the tree V.

The set of all successors of a node v in V is

$$S_V(v) \overset{d}{=} \{w \in V \mid left(w) = v\}.$$

Let $W \subset V$ be such that it satisfies the conditions (i) and (ii) of 2.1. Then W is a *subtree* of V.

Definition 2.3. A subtree $W \subset V$ is said to be a *path* of V iff the function *left*: $W \setminus \{\Lambda\} \to W$ is an injection.

Definition 2.4. Let V be a tree and $C \subset V$. The pair (V, C) is said to be a *game-frame*. □

The above defined game-frame provides frame for games of two players with full information. Let Mr. A and Mr. B be the players. The set C indicates those nodes of the tree V in which player A makes moves. Thus a (V, C) game-frame provides the following course. The starting point $v_0 = \Lambda$, for an arbitrary node

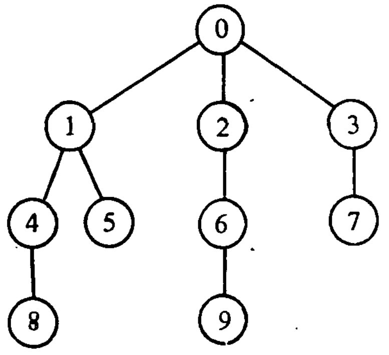

Fig. 2

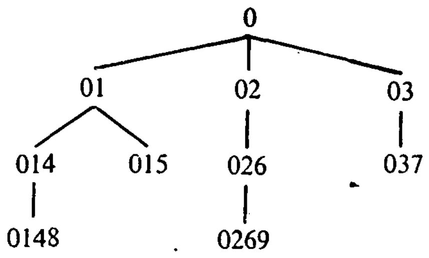

Fig. 3

v_n if $v_n \in C$ then it is A's turn and he can choose one of the alternatives and the course is driven into the corresponding node of $S_V(v_n)$. If $v_n \notin C$ then it is B's turn and the move is analogous to the above situation.

A course in the game-frame (V, C) is a path. Along a course there are the following possibilities:

(i) the course is finite, i.e. neither A nor B can move further because the corresponding set of successive nodes is empty;

(ii) the course is infinite, i.e. A and B can move further in every node.

Thus a game course of the game-frame (V, C) is a finite or infinite path in the tree V.

To have a game in the frame of a given (V, C) a winning condition is needed. According to the aboves the winning condition can be given as a set of those paths in V along which the player, say A, wins. Thus let Γ_A and Γ_B be sets such that $\Gamma_A \cap \Gamma_B = \emptyset$. We say that $\Gamma_A(\Gamma_B)$ is the set of winning paths in V of the player $A(B)$ if it contains those paths along which player $A(B)$ wins. If the course of the game provides such a path that belongs neither to Γ_A nor to Γ_B we have a play which is draw. We note that there is a lot of different possibilities to give the sets Γ_A and Γ_B. For example it may be the case when player A aspires not to win as well as not to lose. This means that A wishes to prevent the winning of player B. In such cases it is quite enough to give the set Γ_B. The set Γ_A is unnecessary because any path that does not belong to Γ_B is satisfactory to player A.

Definition 2.5. Let (V, C) be a game-frame. Moreover, let Γ_A and Γ_B be the set of all winning paths of the players A and B respectively. The quadruple $\mathfrak{A} = (V, C, \Gamma_A, \Gamma_B)$ is said to be a *game* of the game-frame (V, C). □

A player can move in such a way that he decides in advance which alternative he chooses in each possible situation. This means that the player uses a special set of rules that tells him what choices he should make for all situations that might arise during the course of a game. This set of rules is called a strategy which is definable by mathematical tools.

Definition 2.6. Let (V, C) be a game-frame. A function *str* defined on C is a strategy of player A for the game-frame (V, C) iff $str\,(v)\in S_V(v)$ for every $v\in C$. $\square$

The other player's strategy can be similarly defined, but we do not need it.

The function *str* gives the successor for each $v\in C$ and it seems that this depends only on v. However, remember that each node v includes its prehistory.

The strategy of player A defines what move he has to make when he achieves a situation v where his turn is the next. From the above definition follows that a strategy provides the moves in each possible statement many of which do not appear during a game because the player never reaches them if he plays according to the given strategy.

So it is quite natural to define the strategy in a less redundant way, namely considering only the subtree that can be potentially arisen by using the strategy.

Definition 2.7. Let *str* be a strategy of player A for the game-frame (V, C). A subtree $R_{str}\subset V$ is generated by the strategy *str* iff it has the following properties:

(i) if $v\in C\cap R$ then $S_{R_{str}}(v)=\{str(v)\}$ i.e. v has exactly one successor in R_{str} that is picked up by *str*,

(ii) if $v\in R\setminus C$ then $S_{R_{str}}(v)=S_V(v)$.

This subtree R_{str} is unique. $\square$

Thus if player A makes moves in accordance with his strategy *str*, then during a course of the game-frame (V, C) any of the paths of R_{str} can be realized. However since the moves of A are determined by *str*, player B can choose any of his alternatives. Thus B can realize any of the paths of R_{str}. According to the aboves it is quite natural to define a strategy of the player A for a game-frame (V, C) by means of an appropriate subtree of V.

Definition 2.8. Let (V, C) be a game-frame. A subtree $R\subset V$ is said to be a *run* of the game-frame iff the following properties hold:

(i) if $v\in C\cap R$ then there is a unique successor w of v in R. (I.e. there is a unique $w\in R$ such that *left* $(w)=v$.)

(ii) if $v\in R\setminus C$ then $S_R(v)=S_V(v)$. $\square$

It is obvious that for any run R there exists a strategy *str* of player A such that

$$R_{str} = R.$$

Note that for a given run R the appropriate function *str* is not unique because while defining it we consider only its subdomaine $R\cap C$ and its values on $C\setminus R$ can be arbitrary. So far the strategy has been introduced for a game-frame (V, C). Considering winning conditions, i.e. a game $(V, C, \Gamma_A, \Gamma_B)$, we can speak about winning strategy or not losing strategy. A strategy of player A is winning (not losing) iff moving accordingly the course of game realizes only paths belonging to Γ_A (not belonging to Γ_B). I.e. if $\pi \subset R$ then $\pi \in \Gamma_A$ ($\pi \notin \Gamma_B$).

For the illustration of the so far introduced notions let us see the following

Example 2.9. Let us consider the game Last One Loses with game-tree given in Fig. 1. In this case there is one pile of five chips and players have the alternatives to pick up from one to three chips at a move. The frame of this game is the pair (V, C), where

$$V = \{5, 54, 543, 5432, 54321, 543210, 54320, 5431, 54310, 5430, 542, 5421, 54210,$$
$$5420, 541, 5410, 53, 532, 5321, 53210, 5320, 531, 5310, 530, 52, 521, 5210, 520\},$$
$$C = \{5, 543, 542, 541, 532, 531, 530, 521, 520, 54321, 54320, 54310, 54210, 53210\}.$$

The winning condition of the game Last one Loses is as follows:

$$\begin{aligned}\Gamma_A = \{&(5, 54, 543, 5432, 54321, 543210),\\ &(5, 54, 543, 5430),\\ &(5, 54, 542, 5420),\\ &(5, 54, 541, 5410),\\ &(5, 53, 532, 5320),\\ &(5, 53, 531, 5310),\\ &(5, 52, 521, 5210)\}.\end{aligned}$$

Γ_B consists of all paths not belonging to Γ_A.

Let us consider the following run R:

$$R = \{5, 54, 543, 5430, 542, 5420, 541, 5410\}.$$

A corresponding winning strategy *str* of player A is the following:

v	5	543	542	541	54321	532	531	521
$str(v)$	54	5430	5420	5410	543210	5320	5310	5210

3. Logic and arithmetic

3.1 Logic

We intend to develop the theory of nondeterministic programming within the frame of classical first order mathematical logic. To be able to do so we recall the basic notions and definitions that we need to reach our aim.

Definition 3.1. A similarity type ϑ is a pair of functions $(\vartheta_R, \vartheta_F)$ such that rg $\vartheta_F \subset \omega$, rg $\vartheta_R \subset \omega \setminus \{0\}$, do $\vartheta_F \cap$ do $\vartheta_R = \emptyset$ and $|\text{do } \vartheta_R| \cdot |\text{do } \vartheta_F| \leq \omega$. The elements of do ϑ_R and do ϑ_F are called relation and function symbols respectively. ϑ_R and

ϑ_F give the arity of symbols. The 0-ary function symbols are called constant ones. □

For the following we fix a similarity type ϑ for which $= \in \text{do}\, \vartheta_R$ and $\vartheta_R(=)=2$.

Definition 3.2. A ϑ-type model $\mathfrak{A}$ is a function on $\text{do}\, \vartheta_R \cup \text{do}\, \vartheta_F \cup \{\emptyset\}$ such that
(i) $\mathfrak{A}(\emptyset)=A$ is a nonempty set, which is called the universe of the model,
(ii) $\mathfrak{A}(\varrho) \subset {}^{\vartheta_R(\varrho)}A$ for any $\varrho \in \text{do}\, \vartheta_R$,
(iii) $\mathfrak{A}(=)$ is the diagonal relation on ${}^2A$,
(iv) $\mathfrak{A}(f)$: ${}^{\vartheta_F(f)}A \to A$ for any $f \in \text{do}\, \vartheta_F$.
In a special case ${}^0A=\{\emptyset\}$ i.e. if $\vartheta_F(f)=0$ then $\mathfrak{A}(f)$ can be identified with an element of A. □

In general instead of $\mathfrak{A}(s)$ we write $s_{\mathfrak{A}}$ where $s \in \text{do}\, \vartheta_R \cup \text{do}\, \vartheta_F$. A ϑ-type model will always be denoted by a German capital and its universe by the corresponding Roman capital. M_ϑ denotes the class of all ϑ-type models.

Now we turn to the definition of the syntax.

Definition 3.3. Let V be any denumerable set. Let T_ϑ^V be the minimal set satisfying the following properties:
(i) $V \subset T_\vartheta^V$,
(ii) for any n and $f \in \vartheta_F^{-1}(n)$ if $\tau_1, \dots, \tau_n \in T_\vartheta^V$ then $f(\tau_1, \dots, \tau_n) \in T_\vartheta^V$.
The elements of T_ϑ^V are called terms.

Take $A_\vartheta^V = \{\varrho(\tau_1, \dots, \tau_n) \mid \varrho \in \vartheta^{-1}(n), n \in \omega, \tau_1, \dots, \tau_n \in T_\vartheta^V\}$. The elements of A_ϑ^V are called atomic formulas.

The set F_ϑ^V of ϑ-type formulas with variable symbols belonging to V is the minimal set satisfying the following properties:
(i) $A_\vartheta^V \subset F_\vartheta^V$,
(ii) if $\varphi, \psi \in F_\vartheta^V$ then $\varphi \wedge \psi \in F_\vartheta^V$,
(iii) if $\varphi \in F_\vartheta^V$ then $\neg\varphi \in F_\vartheta^V$,
(iv) if $\varphi \in F_\vartheta^V$ and $v \in V$ then $\exists v \varphi \in F_\vartheta^V$.
Let Q_ϑ^V be the minimal set satisfying the above conditions (i)—(iii). The elements of Q_ϑ^V are called quantifier free formulas. □

We use the following abbreviations
a) $\varphi \vee \psi$ for $\neg(\neg\varphi \wedge \neg\psi)$,
b) $\varphi \to \psi$ for $\neg(\neg\psi \wedge \varphi)$,
c) $\varphi \leftrightarrow \psi$ for $\neg(\neg\psi \wedge \varphi) \wedge \neg(\neg\varphi \wedge \psi)$,
d) $\forall v \varphi$ for $\neg\exists v \neg\varphi$,
where $v \in V$ and $\varphi, \psi \in F_\vartheta^V$.

For any $s \in T_\vartheta^V \cup F_\vartheta^V$ let $\text{Var}\, s$ denote the set of free variable symbols occuring in s.

For any $v \in V$, $\tau \in T_\vartheta^V$ and $\varphi \in F_\vartheta^V$ let $\varphi[\tau/v]$ be the formula obtained from φ by replacing every free occurrence of v in φ by τ so that there would not be a collision between the variable symbols of τ and the variable symbols of φ occuring with quantifiers.

Now we define the semantics of the first order language by defining a relation $\models_\vartheta \subset M_\vartheta \times F_\vartheta^V$.

Definition 3.4. Let $\mathfrak{A}\in M_\vartheta$. A valuation of V in $\mathfrak{A}$ is a function $q\colon V\to A$, i.e. a valuation is an element of ${}^VA$. Now we extend the valuation q to a function $\bar{q}\colon T_\vartheta^V\to A$ taking:

(i) $\bar{q}(v)=q(v)$ for every $v\in V$;

(ii) $\bar{q}(f(\tau_1, \dots, \tau_n))=f_{\mathfrak{A}}(\bar{q}(\tau_1), \dots, \bar{q}(\tau_n))$ for every $n\in\omega$, $f\in\vartheta_F^{-1}(n)$ and $\tau_1, \dots, \dots, \tau_n\in T_\vartheta^V$. □

Instead of $\bar{q}(\tau)$ we write $\tau[q]$. It is clear that $\tau[q]$ depends only on the values of $\operatorname{Var}\tau$. So sometimes we use the following notations:

(i) a variable symbol is often written underlined by a waved line to denote its value by a given valuation. E.g. if q is a given valuation then we write $\utilde{x}$ instead of $q(x)$;

(ii) let $\utilde{a}\in A$ denote an arbitrary finite sequence of elements from A. For any $\tau\in T_\vartheta^V$ supposing that $\utilde{a}$ contains at least as many elements as $\operatorname{Var}\tau$ we write $\tau[\utilde{a}]$ instead of $\tau[q]$.

The validity relation is defined by the following well known

Definition 3.5. Let $\mathfrak{A}\in M_\vartheta$ be arbitrary. Moreover, let $\mathfrak{A}\models_\vartheta\subset F_\vartheta^V\times{}^VA$ be the following relation:

(i) $\mathfrak{A}\models_\vartheta\varrho(\tau_1, \dots, \tau_n)[q]$ iff $(\tau_1[q], \dots, \tau_n[q])\in\varrho_{\mathfrak{A}}$ for any atomic formula;

(ii) $\mathfrak{A}\models_\vartheta(\varphi\vee\psi)[q]$ iff $\mathfrak{A}\models_\vartheta\varphi[q]$ and $\mathfrak{A}\models_\vartheta\psi[q]$;

(iii) $\mathfrak{A}\models_\vartheta(\neg\varphi)[q]$ iff $\mathfrak{A}\not\models\varphi[q]$;

(iv) $\mathfrak{A}\models_\vartheta\exists v\varphi[q]$ iff there is a valuation $q^*\colon V\to A$ such that $q^*{\restriction}_{V\setminus\{v\}}=q{\restriction}_{V\setminus\{v\}}$ and $\mathfrak{A}\models_\vartheta\varphi[q^*]$.

$\mathfrak{A}\models_\vartheta\varphi[q]$ means that the formula φ is valid in the model $\mathfrak{A}$ by the valuation q.
In the end $\mathfrak{A}\models_\vartheta\varphi$ iff for every valuation $q\in{}^VA$, $\mathfrak{A}\models_\vartheta\varphi[q]$. □

So the ϑ-type first order language $L_\vartheta=(F_\vartheta^V, M_\vartheta, \models_\vartheta)$ has been defined. If it does not cause ambiguity we write $\models$ instead of $\models_\vartheta$.

Now let $Ax\subset F_\vartheta^V$ be an arbitrary consistent set of formulas. Restricting M_ϑ to $Md(Ax)\overset{d}{=}\{\mathfrak{A}\in M_\vartheta|\mathfrak{A}\models Ax\}$ from the language L_ϑ we can define a new first order language $L_\vartheta^{Ax}=(F_\vartheta^V, Md(Ax), \models)$, which consists of the class of the models of Ax only. Further on in this study while an Ax is considered it is always supposed to be consistent without claiming this explicitly.

The notion of definiability plays a main role among the tools of our investigation. We recall that this notion is used in mathematical logic in two different senses. In the first one it is considered when and how new symbols with given properties can be added to a fixed language. This is the topic of the Definition Theory. For us, however, the other sense which is interested in knowing whether a function or a relation given in an arbitrary model can be expressed in a fixed language is more useful. We introduced the main definitions corresponding to this second approach.

Let us fix a language L_ϑ and let $\mathfrak{A}\in M_\vartheta$ be arbitrary.

Definition 3.6. A partial function $g\colon {}^nA\to A$ is said to be parametrically definable in $\mathfrak{A}$ iff there is a formula $\varphi\in F_\vartheta^V$ such that

(i) $\operatorname{Var}\varphi=\{x_1, \dots, x_n, y, a_1, \dots, a_m\}$;

(ii) There are $\underset{\sim}{a}_1, \dots, \underset{\sim}{a}_m \in A$ such that for any

$$\underset{\sim}{\vec{x}} \in A \quad \text{and} \quad \underset{\sim}{y} \in A, \ \mathfrak{A} \models \varphi[\underset{\sim}{\vec{x}}, \underset{\sim}{y}, \underset{\sim}{\vec{a}}] \quad \text{iff} \quad g(\underset{\sim}{\vec{x}}) = \underset{\sim}{y}.$$

Similarly, a relation $\varrho \subset {}^nA$ is said to be parametrically definable in $\mathfrak{A}$ iff there is a formula $\varphi \in F_\vartheta^V$ such that

(i) $\text{Var}\,\varphi = \{x_1, \dots, x_n, a_1, \dots, a_m\}$;

(ii) There are $\underset{\sim}{a}_1, \dots, \underset{\sim}{a}_m \in A$ such that for any $\underset{\sim}{\vec{x}} \in A$, $\mathfrak{A} \models \varphi[\underset{\sim}{x}, \underset{\sim}{\vec{a}}]$ iff $\underset{\sim}{\vec{x}} \in \varrho$.

A partial function g or a relation ϱ is definable iff the appropriate φ does not contain a_i's. □

We say that the above φ parametrically defines the partial function g or the relation ϱ in $\mathfrak{A}$.

Now we also fix an $Ax \subset F_\vartheta^V$. Let us suppose that for any $\mathfrak{A} \in Md(Ax)$ a function $g_\mathfrak{A}$: ${}^nA \to A$ (a relation $\varrho_\mathfrak{A} \subset {}^nA$) is given. Take $G = \{g_\mathfrak{A} | \mathfrak{A} \in Md(Ax)\}$ $(R = \{\varrho_\mathfrak{A} | \mathfrak{A} \in Md(Ax)\}$.

Definition 3.7. G (or R) is *parametrically definable* in Ax iff there is a formula φ which parametrically defines $g_\mathfrak{A}$ (or $\varrho_\mathfrak{A}$) in $\mathfrak{A}$ for every $\mathfrak{A} \in Md(Ax)$.

If the set $\{g_\mathfrak{A} | \mathfrak{A} \in Md(Ax)\}(\{\varrho_\mathfrak{A} | \mathfrak{A} \in Md(Ax)\})$ is parametrically definable and the definition is given by the formula φ then the function symbol g (the relation symbol ϱ) is said to be *universally definable in* $Md(Ax)$.

Example 3.8. Let $\varphi \in F_\vartheta^V$ be such that $\text{Var}\,\varphi = \{x_1, \dots, x_k, y\}$, and suppose that

$$Ax \models \forall x_1 \dots x_k \forall y \forall z(\varphi \land \varphi[z/y] \to y = z) \tag{1}$$

If so then in every model $\mathfrak{A}$ of Ax φ defines a partial function in the following way:

(i) $\underset{\sim}{\vec{x}} \in \text{do} f$ iff $Ax \models \exists y \varphi[\underset{\sim}{\vec{x}}]$,

(ii) $f(\underset{\sim}{\vec{x}}) = \underset{\sim}{y}$ iff $Ax \models \varphi[\underset{\sim}{\vec{x}}, \underset{\sim}{y}]$.

By (1) this definition is good and thus we use the following abbreviation:

$$\text{Parc}\,\varphi \overset{\text{d}}{=} \forall \vec{x} \forall y \forall z(\varphi \land \varphi[z/y] \to y = z). \quad \square$$

We say that the above φ parametrically defines G or R. If φ contains no a_i's we omit the adjective "parametrically".

Remark 3.9. If the above G is definiable in Ax and every $g_\mathfrak{A}$ is total then a new function symbol g "can be added" to ϑ_F of arity n with the following new axiom

$$Ax_g\colon \forall \vec{x} \forall y(y = g(\vec{x}) \leftrightarrow \varphi(\vec{x}, y))$$

where φ defines G. So we get a new language $L_\vartheta^{Ax'}$, where $Ax' = Ax \cup \{Ax_g\}$. The phrase "can be added" means that for any $\varphi \in F_\vartheta^V$, $Ax' \models \varphi$ iff $Ax \models \varphi$.

The details see in Section 2.9 of Mendelson (1964).

A similar fact holds for the above R.

3.2. Arithmetic

As known arithmetic plays an important role in computer science. It provides an unambiguous characterization of any formal language syntax. This permits the widespread use of computers since their functioning is based on natural number representation. While the numeric use of computers arithmetic plays an important role since the data form a structure satisfying the basic features of arithmetic. Arithmetic is also important to formalize our intuitive concept about discrete time connected with computer functioning. Thus in our investigation of programming theory arithmetic plays an important role. Namely, it provides formal tools to characterize sequences which prove to be useful in the study of program properties.

Let η be the type of arithmetic, i.e. do $\eta_R=\{=\}$, do $\eta_F=\{0, 1, +, \cdot\}$ and $\eta_F(0)=\eta_F(1)=0, \eta_F(+)=\eta_F(\cdot)=2$.

For the axiomatization of the arithmetic we choose the well-known Peano axioms:

$$A_1 \overset{d}{=} \neg(v+1 = 0)$$

$$A_2 \overset{d}{=} v+1 = w+1 \to v = w$$

$$A_3 \overset{d}{=} v+0 = v$$

$$A_4 \overset{d}{=} v+(w+1) = (v+w)+1$$

$$A_5 \overset{d}{=} v\cdot 0 = 0$$

$$A_6 \overset{d}{=} v(w+1) = (v\cdot w)+v$$

$$A_{7\varphi} \overset{d}{=} \varphi[0/v] \land \forall v(\varphi \to \varphi[v+1/v]) \to \forall v\varphi$$

Take $I=\{A_{7\varphi} | \varphi \in F_\eta^V$ and $v \in \text{Var}\, \varphi\}$. The set of Peano-axioms is

$$PA \overset{d}{=} \{A_i | 0 \leqq i \leqq 6\} \cup I.$$

For detailed analysis of PA see e.g. MENDELSON (1964).

As usually we use the following abbrevations

$$x \leqq y \quad \text{instead of} \quad \exists z(z+x=y),$$

$$x < y \quad \text{instead of} \quad x \leqq y \land \neg x=y.$$

We recall that for every infinite cardinal there are at least continuum number of non-isomorphic models of that cardinality of PA. For every $\mathfrak{A} \in Md(PA)$ its smallest submodel A_c satisfies PA and these submodels are isomorphic to each other and they are called standard models of PA. We would like to consider only standard models but unfortunately that is impossible at a first order language because there is no first order formula describing exactly the standard part of the models of PA. Thus if we are interested whether a first order formula is valid then we must consider not only standard models but nonstandard ones as well.

As usually $\mathfrak{N}$ denotes the fixed standard model of PA and $\mathbf{N}$ stands for its universe.

For handling of non-standard models see e.g. ROBINSON (1966).

3.3 The role of time in the theory of programming

As mentioned already in Introduction often not only the output result of a computing process is significant, but its temporal course too. Thus we would like to develop such a theory of nondeterministic programming that handles both data and time explicitly by the help of first order tools.

The representation of data within the frame of first order logic is straightforward; it can be done by the universe of the classical models. However relations and functions of the models correspond to data properties and to their possible changes respectively. Thus from the point of view of data computers are represented by the models of first order languages. Thus the previously mentioned representation neglects the explicit time representation. How to represent time is a question that should be looked at in details. But the functioning of computers is controlled by an "inner clock" so the change in data happens in time.

We assume that a change in data corresponds to a command which is executed for a timecycle of the machine. Let us denote the set of these disjoint time intervals by T. From theoretical point of view the time intervals of T can be considered as time moments supposing that the change takes place infinitely fast. We also assume that a machine works as long as it is needed i.e. as long as it is required by the program. This means that a machine itself can work infinitely long never stopping due to a break-down. However it stops only if it is required by the program and by this the program execution terminates. Let us consider the simplest case when there are only assignment statements.

The execution of a program on a machine is but the execution of assignment statements step by step i.e. iteratively. The transition of states of the machine representing the change in data is defined by the transition function. This function can be defined by induction on T as follows. In case we already know the state S_t of the machine at moment t then the state at the next moment $t+1$ can be defined by the state S_t using the concrete command that is to be executed in the moment t. To describe this by mathematical tools the closeness of the transition function under iteration (recursion) must be ensured.

Thus to represent time an arbitrary structure can be used which provides the starting moment, the generation of the next moment and the induction by succession. For example if we take a $\{(0, 0), (\,', 1)\}$-type structure $\mathfrak{T}=(T, 0, ')$ on which the induction works well then this can be used to represent time. Here T represents the set of time moments. Note that further on it will be also supposed that on the set of time moments T the usual addition and multiplication are also considered and the time moments are in order.

Thus to represent discrete time the use of the structure $\mathfrak{N}=(\mathbf{N}, 0, 1, +, \cdot)$ of natural numbers is obvious. However our main standpoint is to use classical first order language to describe models. Thus we cannot restrict ourselves to the standard model $\mathfrak{N}$ but any model $\mathfrak{T}$ of an appropriate first order axiomatization of $\mathfrak{N}$ is allowed. Consequently beside $\mathbf{N}$, which is very close to our intuition, very strange sets of time moments are allowed as well. Especially such sets T in which "infinitely large" (or non-standard) time moments also occur.

As usual the theories of programming developed so far use either implicitely or explicitely the set of natural numbers $\mathbf{N}$ to represent time.

So our assumption that the structure representing time has to satisfy only one condition, namely the axiom scheme of induction, seems not to be very close to our intuition. Thus let us go into a bit more details.

Our first notion is purely theoretical. If the iteration is the essence of programming then to represent time any such model can be chosen that provides to follow the changes done by iteration. So on this structure the induction must be allowed. Hence developing a theory of programming we have no reason to introduce further restriction for time (e.g. to suppose that time moments belong to N).

If we have a practical look at it then the situation seems to be totally different. Namely, in practice there exists no procedure containing non-standard number of steps. So the "infinitely large" time moment seems to be a fiction. However, if we consider the history of mathematics this opinion can be dissipated. That is, infinitesimal values play an important role in the history of mathematical analysis but their reason for the existence was only recently observed by A. Robinson and his followers.

Non-standard analysis is applied in computer science as well. It provides some very effective methods to solve differential and integral equations.

In order to develop a theory of programming being able to analyse the real situations of programming practice there is no reason to restrict it to the considered notions of "standard and real" machines and time. It is not our aim, of course, to investigate machines with non-standard time. Nevertheless if we have a theory of programming which can handle non-standard time as well then the execution of a program being correct within the frame of this theory will be correct in any machine with any type of time, especially in the machines with standard time.

Indeed well written and well used programs, in our opinion, can be executed in machines with arbitrary type of time, though programmers having developed the programs know absolutely nothing about this. This is so because programmers write down programs thinking in first order language though always imagining the standard time (i.e. the set N) to it. These impressions, fortunately are not embedded in the programs!

It may seem that the first order language is not sufficient to think about programs for it might provide far too many restrictions. However we have proved in Gergely and Úry (1978) that within the frame of classical first order logic for the sequential and deterministic programming a theory of programming of unified attitude can be developed and this frame fully satisfies the solution of the tasks of a programming theory. Present work shows that this frame is completely satisfactory for developing a theory of non-deterministic programming as well.

Now we give the mathematical description of the programming situation. Let ϑ be an arbitrary similarity type containing the type η of arithmetic ($\eta\subset\vartheta$). Intuitively ϑ provides the *name* of those relations and functions which have to be understood by the computer. The properties of relations and functions are described by the set of formulas $Ax\subset F_{\vartheta}^{Y}$. The set of axioms Ax expresses the expectations with respect to data and "hardware". Intuitively we always suppose that $PA\subset Ax$ i.e. the computer "understands" the arithmetic in the form of Peano axiomatization.

According to the above saids it is clear that at least two sets are needed to characterize a computer: — the set A of possible data and the set T of possible time moments. We intend to speak of both time and data in a first order language. Since time and data are of different entities it is advisable to distinguish their languages while describing a computer. This could be done by the use of a two sorted first order language, where the first sort corresponds to time and the second one to data. If different data types were allowed then we need a many sorted first order language. The mixed sorted functions and relations describe the connection between time and data.

For the sake of simplicity let us stick to the frame of the classical (one sorted) first order language and to describe time we use the same language as for data representation with the only difference that a new unary relation symbol ζ is introduced. By this we supply our models with an inner time supposing that time can be modelled by data. Of course not each data type can be satisfactory for this aim, e.g. the Boolean data type is not.

Already in this approach we seem to meet the advantages provided by the explicit handling of time. Therefore, it was not necessary to introduce and use the many-sorted first order language.

To describe time we introduce a new unary relation symbol $\zeta \notin$ do ϑ and let us add ζ to the type ϑ. So, we have $\vartheta^* = \vartheta \cup \{(\zeta, 1)\}$. Expectations with respect to time beyond data would be given by a set of axioms $Ax^*(Ax \subset Ax^* \subset F^V_{\vartheta^*})$. Of course the set Ax^* is larger than the set of axioms Ax expressing the expectations with respect to data. To formalize the minimal properties expected from time we introduce the following notations $\zeta^*(x) \overset{d}{=} \exists t\,(x \leqq t \wedge \zeta(t))$ (where the relation symbol $\leqq$ is the ordering used in PA),

$$B_0 \overset{d}{=} \zeta^*(0),$$

$$B_1 \overset{d}{=} \zeta^*(x) \to \zeta^*(x+1).$$

The fulfilment of the formulas B_0 and B_1 provides that the set of time moments is not empty. The induction under ζ^* can be formalized as follows:

$$B_{2\varphi} \overset{d}{=} [\varphi(0) \wedge \forall x(\zeta^*(x) \wedge \varphi(x) \to \varphi(x+1))] \to \forall x(\zeta^*(x) \to \varphi(x)).$$

B_1 and $B'_{2\varphi}$s provide the closing under addition and multiplication.

According to the abovesaids with respect to time we always suppose that ζ^* satisfies PA^*, where

$$PA^* = \{B_0, B_1\} \cup \{B_{2\varphi} | \varphi \in F^V_{\vartheta^*}, x \in \mathrm{Var}\,\varphi\}.$$

Definition 3.10. A set of formulas $Ax^* \subset F^V_{\vartheta^*}$ is said to be a *ϑ-type system* if $PA^* \subset Ax^*$. $\mathfrak{A}$ is *a model with inner time* $T_{\mathfrak{A}}$ of the system Ax^* if $\mathfrak{A} \models Ax^*$ and $T_{\mathfrak{A}} = \{a \in A \,|\, \mathfrak{A} \models \zeta^*[a]\}$. □

Examples 3.11. (i) Let $Ax = PA$ and $Ax^* = PA \cup \{\forall x\, \zeta(x)\} \cup PA^*$. In this case any model $\mathfrak{A} \in Md(PA)$ will be the model of Ax^* if ζ is interpreted by the universe A itself. The model $\mathfrak{A}'$ provided by such a way will be evidently a model of Ax^* with inner time A.

(ii) Let $\tau_1 = \tau_2$ be a Diophantine equation which has no solution in **N**, but $PA \not\models \neg \tau_1 = \tau_2$. Let $Ax^* = PA \cup PA^* \cup \{\forall \vec{x}\, \zeta^*(\vec{x}) \to \neg \tau_1(\vec{x}) = \tau_2(\vec{x})\}$. Using the method of (i), from a model $\mathfrak{A} \in Md(PA)$ a model $\mathfrak{A}'$ can be arisen, which would be a model inner time of Ax^* if the equation $\tau_1 = \tau_2$ has no solution in A.

(iii) Let $Ax^* = PA \cup PA^*$, and let $\mathfrak{A} \in Md(PA)$ be arbitrary.

It is obvious that if ζ would be interpreted by the standard part of the model $\mathfrak{A}$ (i.e. by **N**) then the model $\mathfrak{A}'$ arisen by using the method of (i) would be a model of Ax^* with inner time **N**. □

Remarks 3.12. (i) In Definition 3.10 it would be satisfactory to claim that $Ax^* \vdash PA^*$. Thus in Definition 3.11 (i) $Ax^* = PA \cup \{\forall x \zeta(x)\}$ would be enough.

(ii) Intuitively speaking a ϑ-type system Ax^* provides the description of the hardware of a computer. It fixes those features that characterize the static (Ax) and the dynamics ($Ax^* \setminus Ax$) of the computer. Ax^* may have a lot of models which are usually different but it does not completely define a machine. However only those features of machines are interesting in our investigation that are true in every model of Ax^*. □

3.4 Recursive definiability

Let Ax^* be a ϑ-type system. A fairly often used method of implicit definition is the recursive one. In order to understand this situation in the case when Ax^* refers to time we need the following. Let ϱ be a new k-ary relation symbol not occuring in ϑ^* ($\varrho \notin$ do ϑ^*). Let $\varphi(\varrho)$ denote the inclusion $\varphi \in F^V_{\vartheta^* \cup \{(\varrho, k)\}}$ i.e. $\varphi(\varrho)$ is a formula of the syntax the type of which is the extention of ϑ^* with the k-ary relation symbol ϱ. Moreover we need a tool by which we can reduce a formula of $F^V_{\vartheta^* \cup \{(\varrho, k)\}}$ to a formula of $F^V_{\vartheta^*}$. For this the following type of substitution can be used.

Definition 3.13. Let $\varphi \in F^V_{\vartheta^* \cup \{(\varrho, k)\}}$ and let $\chi \in F^V_{\vartheta^*}$ with $\mathrm{Var}\,\chi = \{x_1, \dots, x_k\}$. Let $\varphi[\chi/\varrho]$ be defined by the following way:

(i) if ϱ does not occur in φ then $\varphi[\chi/\varrho] = \varphi$,

(ii) if $\varphi = \varrho\,(\tau_1, \dots, \tau_k)$ then $\varphi[\chi/\varrho] = \chi[\langle \tau_i/x_i \rangle_{i \in [1,k]}]$,

(iii) if $\varphi = \varphi_1 \langle\rangle \varphi_2$ where $\langle\rangle$ is either $\wedge$ or $\vee$ then $\varphi[\chi/\varrho] = \varphi_1[\chi/\varrho] \langle\rangle \varphi_2[\chi/\varrho]$,

(iv) if $\varphi = \neg\psi$ then $\varphi[\chi/\varrho] = \neg\psi[\chi/\varrho]$,

(v) if $\varphi = Qv\psi$ then $\varphi[\chi/\varrho] = Qv\psi[\chi/\varrho]$ where Q is either $\forall$ or $\exists$. □

We are interested whether the equation $\varrho \leftrightarrow \varphi(\varrho)$ has a solution in Ax^* i.e. whether a formula $\chi \in F^V_{\vartheta^*}$ exists such that

$$Ax^* \models \chi \leftrightarrow \varphi[\chi/\varrho].$$

In this case we say that χ is a solution of the recursive equation $\varrho \leftrightarrow \varphi(\varrho)$ in Ax^*.

Moreover for some types of formulas in $F^V_{\vartheta^* \cup \{(\varrho, k)\}}$ there exists a minimal solution of the above recursive equation.

Definition 3.14. A formula $\varphi \in F^V_{\vartheta^* \cup \{(\varrho, k)\}}$ with $\mathrm{Var}\,\varphi = \{x_1, \dots, x_k\}$ is a pedigree formula iff there is a formula $\psi \in F^V_{\vartheta^* \cup \{(\varrho, k)\}}$ such that

(i) $Ax^* \models_{\vartheta^*} \varphi \leftrightarrow \psi$;

(ii) ψ has the form $\psi = \psi_0 \vee \psi_1$, where ϱ does not occur in ψ_0 and the symbols $\neg$ and $\forall$ do not act on ϱ in ψ_1;

(iii) all occurences of ϱ in ψ contain only variable symbols;

(iv) bounded variable symbols of ψ are distinct from each other. □

The forthcoming theorem shows the recursive definition to be allowed, provided that we make only "positive" statements. This latter is contained in condition (ii) of the above definition of pedigree formula. It is needed in recursive definition to consider only already existing objects and not to speak about such that have not occured so far but may do so sometimes in the future. So for example we cannot say which objects should not belong to a recursively defined set. So the condition (ii) provides the constructive feature of the recursive definiability. The conditions (iii) and (iv) are merely technical. If a formula $\varphi \in F^V_{\vartheta^* \cup \{(\varrho,k)\}}$ satisfies conditions (i) and (ii) then it is already a pedigree formula, of course with another ψ as if φ satisfied conditions (iii) and (iv) as well.

Theorem 3.15. For any pedigree formula $\varphi \in F^V_{\vartheta^* \cup \{(\varrho,k)\}}$ there exists a formula $\chi_\varphi \in F^V_{\vartheta^*}$ such that

(i) $\mathrm{Var}\,\chi_\varphi = \mathrm{Var}\,\varphi$;

(ii) $Ax^* \models \chi_\varphi \leftrightarrow \varphi[\chi_\varphi/\varrho]$ i.e. χ_φ is a solution of the recursive equation $\varrho \leftrightarrow \varphi(\varrho)$;

(iii) if χ is any other solution of the recursive equation, i.e. it is a formula of $F^V_{\vartheta^*}$ such that $Ax^* \models \chi \leftrightarrow \varphi[\chi/\varrho]$ then $Ax^* \models \chi_\varphi \rightarrow \chi$, i.e. χ_φ is the minimal solution.

Sketch of the proof. It is similar to that of Theorem 3.4 in Gergely and Úry (1978). It uses the fact, that there is a formula $\psi = \psi_0 \vee \psi_1$ such that the properties (i)—(iv) of Definition 3.14 hold. By using the property (i) and the following fact: if $Ax^* \models \varphi_1 \leftrightarrow \varphi_2$ for any $\varphi_1, \varphi_2 \in F^V_{\vartheta^* \cup \{(\varrho,k)\}}$ then for any $\chi \in F^V_{\vartheta^*}$ with exactly k variable symbols:

$$Ax^* \models_{\vartheta^*} \varphi_1[\chi/\varrho] \leftrightarrow \varphi_2[\chi/\varrho]$$

to prove the theorem it is enough to construct such a formula χ_φ that $Ax^* \models \chi_\varphi \leftrightarrow \psi[\chi_\varphi/\varrho]$. The idea of the construction is that χ_φ is either ψ_0 or it builds up from ψ_0 applying ψ_1 ζ-many times. The building up of χ_φ can be done similarly to that of Gergely and Úry (1978). □

Abstract (to Part 1)

Nondeterministic programming play an increasing role in the theory of programming. This role is discussed in Section 1 together with the role of classical first order logic in developing a theory of programming. Two kinds of nondeterminism are considered: *any* and *every*. The uncertainty in programs that use both *any* and *every* is quite analogous to that of game situations. So in our theory games are the centre of interest. The basic constructions of games are introduced in Section 2. The theory will be built within the frame of classical first order logic. The basic notions and constructions needed to develop this theory are given in Section 3.

RESEARCH INSTITUTE FOR
APPLIED COMPUTER SCIENCE
CSALOGÁNY U. 30—32.
BUDAPEST, HUNGARY
H—1536

References

[1] Ashcroft, E. and Z. Manna, Formalization of properties of parallel programs, *Artificial Intelligence Memo* AIM—110, Stanford University, Stanford, 1970.

[2] Dijkstra, E. W., Guarded commands; nondeterminacy and formal derivation of programs, *Comm. ACM*, v. 18, 1975, № 8, pp. 453—457.

[3] Egli, H., A mathematical model for nondeterministic computations, Technological University, Zürich, 1975.
[4] Francez, N., C. A. R. Hoare and W. P. de Roever, Semantics of nondeterminism, concurrency and communications, *Mathematical Foundations of Computer Science*, Ed. J. Winkowski, Springer Verlag, Berlin, 1978.
[5] Gergely, T. and L. Úry, Mathematical theories of programming (manuscript), Budapest, 1978.
[6] Harel, D. and V. R. Pratt, Nondetermism in logics of programs, Report MIT/LCS/TM—98, 1978.
[7] Hoare, C. A. R., Communicating sequential processes, *Comm. ACM*, v. 21, 1978, № 8, pp. 666—677.
[8] Manna, Z., The correctness of nondeterministic programs, *Artificial Intelligence*, v. 1. 1970, № 1—2, pp. 1—26.
[9] Mendelson, E., *Introduction to mathematical logic*, Van Nostrand, N. Y., 1964.
[10] Milner, R., An approach to the semantics of parallel programs, *Proceedings of the Convegno di Informatica Teorica*, Instituto di Elaborazione delle Informazione, Pisa, 1973.
[11] Milner, R., Synthesis of communicating behaviour, *Mathematical Foundations of Computer Science*, Ed. J. Winkowski, Springer Verlag, Berlin, 1978.
[12] Mirkowska, G., Algorithmic logic with nondeterministic programs, *Proceedings of Colloquium Mathematical Logic in Programming*, North Holland, 1980 (under publication).
[13] Owicki, S. and D. Gries, An axiomatic proof technique for parallel programs I., *Acta Inform.*, v. 6, 1976, pp. 319—340.
[14] Plotkin, G. D., A powerdomain construction, *SIAM J. Comput.*, v. 5, 1976, № 3, pp. 452—487.
[15] Robinson, A., *Nonstandard analysis*, North Holland, Amsterdam, 1966.

(Received Sept. 7, 1979)