Acta Informatica 22, 311–331 (1985)

Proving Properties of Pascal Programs in MIZAR 2

Piotr Rudnicki* and Włodzimierz Drabent

Institute of Computer Science, Polish Academy of Sciences, P.O. Box 22, PL-00-901 Warszawa PKiN, Poland

Summary. In this paper we present the so called natural semantics for a subset of Pascal programming language. A set of sentences of first order predicate calculus defines the meaning of the Pascal language constructs. The meaning of a specific program is defined separately by another set of sentences which can be generated automatically. Both these sets together constitute axiomatics of a theory, called the theory of a specific program. The axiomatics is built in such a way that its logical consequences describe all the computational processes defined by the program. Proofs of properties for two small programs are discussed in detail. These properties and their proofs are recorded in the MIZAR 2 language – a computer formalization of predicate calculus. MIZAR 2 proof checker was used to verify the proofs.

1. Introduction

As a foundation we take the natural semantics of a subset of the Pascal programming language defined by a set of the first order logic sentences. We discuss a method for describing a program by another set of first order sentences. In particular we discuss a software system which has been developed to automatize this process. Combining these two sets of sentences with the properties of used data types one may try to prove, using the ordinary rules of logic, any program property, e.g. correctness and termination. This is done for two small sample programs.

The idea of defining the semantics of a programming language in first order logic is due to Burstall [1]. A modified version of this idea was also given by Winkowski [9]. The starting point of this approach is: from an abstract point of view each program defines a system

$$M = (S, R)$$

* Present address: Department of Computing Science, The University of Alberta, Edmonton, Alberta T6G 2H1, Canada

where S is a set of McCarthy state vectors ($\langle control \rangle$, $\langle data \rangle$) and $R \subseteq S \times S$ is a *transition* – or *next state* – relation. Burstall's idea was to characterize such a system by first order axioms, some of them reflecting the semantics of the programming language and others describing the program under consideration.

For instance, the meaning of the following instruction

$$1\text{: } \textbf{goto } 2$$

is generally defined by the Pascal semantics and can be expressed by the axiom

$$(\forall s, s')(sRs' \ \& \ control(s)=1 \ \Rightarrow \ control(s')=2 \ \& \ data(s)=data(s')).$$

If a program contains also an instruction labelled by 2 and the instruction is accessible from the point of jump then this fact is expressed by another axiom specific to the program, namely

$$(\forall s)(control(s)=1 \Rightarrow (\exists s')(sRs')).$$

More complicated cases will be discussed in the sequel.

Winkowski [9] characterizes the set of computations of M instead of M itself. A *computation* of M is a sequence of state vectors

$$s_0, s_1, s_2, \ldots$$

satisfying $s_0 R s_1$, $s_1 R s_2, \ldots$ that possibly terminates, i.e. its last state is not in the domain of R. Each initial subsequence

$$s_0, s_1, s_2, \ldots, s_k$$

of such a computation C is called a *history* of C. Given such a history h we define $state(h)$ as s_k and call it the *resulting state* of h. We will consider only deterministic systems with the transition relation being a function. If there exists s_{k+1} such that

$$s_0, s_1, s_2, \ldots, s_k, s_{k+1}$$

is also a history h' of C then we say that h' *follows* h and write $h' = next(h)$. Further we will in fact be concerned with a system equivalent to M of the form

$$M' = (S^*, next).$$

We will present a way of building a set of first order axioms characterizing the semantics of a program. The axioms will implicitly define any computation C of the program in terms of histories of C, their order represented by the function *next* and the resulting states of histories. For instance, the axiom describing

$$1\text{: } \textbf{goto } 2$$

is now

$$(\forall h)\ (control(state(h))=1 \Rightarrow control(state(next(h)))=2 \ \& \ data(state(next(h)))=data(state(h))).$$

Some effort has been made to automatize the process of deriving axioms which define the meaning of a program from the program text (Oryszczyszyn [5]). The idea was to develop a software system called DESCRIBER which produces axiomatic descriptions of programs written in a subset of Pascal. The descriptions are formulated in a language called MIZAR 2 (Trybulec [7]) which is a computer oriented formalization of the first order functional calculus, elements of set theory and natural deduction. The proofs for properties of programs may also be recorded in that language. The language MIZAR 2 has been implemented on ICL 1900 machines, i.e. there exists a processor of this language, which contains a module for checking proofs and inference steps recorded in MIZAR 2 input texts.

In Sect. 2 we shall explain the approach to the programming language semantics, the idea of the DESCRIBER and we shall illustrate (through an example already recorded in MIZAR 2) how this system works. In Sects. 3 and 4 we formulate and prove properties of programs after including specific data type theory into the theory giving the meaning of programs.

We hope that the approach may become useful in dealing with lower level properties of programs (like implementation of abstract data types, memory allocation, variable aliasing, non-structural flow of control). The natural semantics also makes possible proving properties beyond partial correctness, for instance termination.

2. Program Description Technique

2.1. The Role of the MIZAR 2 Language

The MIZAR 2 language (Trybulec [7]) serves to record reasonings conducted in the first order logic augmented by some notions of set theory. There exists a computer processor for the language which determines whether an input text complies with the MIZAR 2 syntax rules. It contains a checking module which checks whether the recorded inferences are consistent with the rules of logic.

In the experiments reported in this paper the MIZAR 2 language plays an essential role. In this language we will formulate:
- the set of sentences defining the meaning of the program,
- properties of the program to be proven,
- proofs of these properties.

The structure of the MIZAR 2 texts is similar to that of mathematical articles. In the first part of such a text, called environment or preliminaries, we display notions and facts assumed to be given. In the second part – called the text proper – we formulate and prove theorems (by hand).

In order to facilitate the reading of appendices containing the entire MIZAR 2 texts we will gradually introduce the notation of this language.

2.2. Approach to the Semantics

In this work we apply the *natural semantics* of the programming language, cf. Burstall [1], Winkowski [9]. The natural semantics originates from looking at

a program in terms of a compiler or even in terms of the machine code produced by a compiler. This approach reminds the operational semantics at least in the underlying intuition.

The notions of the *history* of computation and the *control point* (instruction location) are basic and were partially explained in Sect. 1. The other primitive notions are that of *value*, of *address* where the value is stored and of the *valuation of an address* in a history. An address is also treated as a value, i.e. a member of a certain set of addresses – elsewhere called a memory. The meaning of a variable is associated with an element of a set of addresses. The semantics of an instruction is understood as a function which (possibly) changes the contents of memory and determines the instruction to be performed next. Treatment of procedures and functions corresponds to their usual implementation – involving activation records, return addresses and static/dynamic links.

The meaning of Pascal statements is defined here in an axiomatic way. We adopt a set of axioms as the full characterization of the Pascal statements meaning. Despite the similar name this should not be confused with the Hoare's axiomatic semantics. Observe that we can formulate (and prove) not only partial correctness of a statement with respect to pre- and postcondition but also other program properties. This includes termination and those features of computational process which are inexpressible in terms of initial/final states. Such features are significant e.g. when adequacy of simulation programs is concerned, cf. [10].

The natural semantics of the Pascal programming language was presented by Byliński [2] and concerns the Pascal version as reported by Jensen and Wirth [3]. The method of obtaining a description (i.e. the set of axioms reflecting meaning) for a specific Pascal program is given by Oryszczyszyn [5]. The approach of [2, 5] to procedures and functions does not seem to be satisfactory and work on improving it is in progress. Here we are not going to report the whole of [2] and [5]. We limit the presentation only to those features of Pascal which are necessary to discuss examples in Sects. 3 and 4. We will present the semantics and the program description technique for the following Pascal notions:

- types: standard, pointer and record,
- variables: entire, referenced and field designator,
- expressions,
- statements: compound, assignment and **while.**

Please notice that we omit procedures and functions and will discuss only main programs.

2.3. *Description Apparatus and Semantics*

2.3.1. Histories and Control Points. These are the basic notions of the description technique; the underlying intuition was explained in the introduction. Control points are used in a program description and are introduced by the DESCRIBER according to the assumed semantics of statements (see below).

Roughly speaking every statement in a program is preceded by a control point, which is a counterpart of the instruction location in a computer.

Since each history of a program run has its resulting (and unique) state then instead of speaking about *control and data of the state* we will speak about *control and data of the history*. In MIZAR 2 we declare

type *HISTORY*

and every object of that type will be a history of a computation. For control points we use *natural* numbers (standard notion in MIZAR 2). In the description technique we introduce functional notation to refer to an history (i.e. its resulting state) components. For a *HISTORY H* by *CP*(*H*) we denote the control point of *H* and by *NH*(*H*) the next history of *H* (Pascal programs are deterministic). In MIZAR 2 the function symbols *CP* and *NH* are introduced in the following way:

for *H* **being** *HISTORY* **consider** *CP*(*H*) **being** *natural*;
for *H* **being** *HISTORY* **consider** *NH*(*H*) **being** *HISTORY*

The functions *CP* and *NH* obtain their definitions in the description of any specific program. We will use two standard control points named *START* and *FINISH* which obtain their definition in the description of a specific program. We also use a constant *FH* for denoting the initial (first) *HISTORY* of a computation with the property *CP*(*FH*)=*START*.

2.3.2. Types. Each Pascal type – standard or defined – is described as a nonempty set. That agrees with the approach of Jensen and Wirth [3] where it is said that each Pascal type shall be understood as a set of values. The name of the introduced set is the same as the name of a type which ocurred in a program, except for standard types. For instance the standard Pascal type *integer* will be described by a set *integers* – the set of integer values which is in MIZAR 2 a standard object. The technique of describing user defined types is explained in Sect. 4.

For a given set *A* in the description we consider a set of *pointers to* (*adresses of*) the elements of *A* and this set is denoted by *PTR*(*A*). For instance, a Pascal variable of type *A* is described as an element of *PTR*(*A*). Due to a certain MIZAR 2 restriction, for any set *A* we discern a unique *NIL*(*A*) being an element of *PTR*(*A*). Note that in Pascal the value **nil** is compatible with any pointer type.

2.3.3. Variables. To each program variable of type *A* we assign an element of *PTR*(*A*) as its meaning. Declared Pascal variables are described with the help of the *VAR* function having two arguments, the first being a natural number serves as a variable counter and the second defines the type of the variable. The arity and type of the function *VAR* are declared in MIZAR 2 as follows:

for *N* **being** *natural*, *A* **being** *nonempty*
consider *VAR*(*N*, *A*) **being** *element* **of** *PTR*(*A*)

For instance the declaration of variables

var x, y, z: *integer*

is described as

take $X = VAR(1, integers)$
take $Y = VAR(2, integers)$
take $Z = VAR(3, integers)$.

The introduced objects X, Y and Z are – according to the declaration of VAR – elements of $PTR(integers)$. We assume that if $X = VAR(N, A)$, $Y = VAR(M, B)$ and $N \langle\rangle M$ then $X \langle\rangle Y$. This means that two declared variables with different variable counters are different, i.e. they denote different addresses.

2.3.4. VALUATIONS (of addresses). Since we will consider different stages of the program run we introduce a *valuation* function VAL which for an element V of $PTR(A)$ and an $HISTORY$ H gives an *element of* A – a value stored under the address V in H. The addresses (elements of a certain $PTR(A)$) we obtain e.g. from the descriptions of declared variables. In MIZAR 2 the VAL function is introduced in this way:

definition
  let A **be** *nonempty*, H **be** $HISTORY$, C **be** *element* **of** A,
    V **be** *element* **of** $PTR(A)$;
  pred $C = VAL(V, H)$
end

Obviously what is defined above is only the "shape" of the VAL function and *not* the function itself. The function denoted by VAL is defined in any specific program description; actually this is not a one function but a family of functions since VAL contains a hidden parameter, namely that denoted by A above.

2.3.5. Expressions. Each Pascal expression has a type. Pascal expressions of type A we describe using the declared in MIZAR 2 type:

$$EXPRESSION \textbf{ of } A$$

where A is a set describing a certain Pascal type A.

If E is an $EXPRESSION$ **of** A then the value of E in an $HISTORY$ H we denote by $E.H$ which denotes an element of A.

If E is an $EXPRESSION$ **of** $PTR(A)$ then by $\uparrow E$ we mean the dereferenced expression E, i.e. an $EXPRESSION$ **of** A.

The following formula connects the expression valuation with the previously introduced function VAL:

$$(\uparrow E).H = VAL(E.H, H)$$

where E is an $EXPRESSION$ **of** $PTR(A)$ for some A.

Pascal expressions are built from variables and constants by means of operators. In the language Pascal the constant (or variable) – say C – outside of context is indistinguishable from the simple expression built from C. This is

not the case in our description technique. We will make a syntactic distinction between a constant or a variable and an expression constructed from it. If C is an *element* **of** A (a value of type A) by $.C$ we denote the *EXPRESSION* **of** A built from C. E.g. $+3$ is an *element* **of** integers while $.+3$ is an *EXPRESSION* **of** *integers*; X is an *element* **of** *PTR*(*integers*) and $.X$ is an *EXPRESSION* **of** *PTR*(*integers*).

The difference between the expressions on the left hand side and right hand side of an assignment statement is immediately mirrored in the description of these expressions. E.g. for the following assignment (for the declarations of x and y see above):

$$x := y$$

the left hand side expression is described as $.X$ since x denotes here a certain address where an integer value may be stored. Notice that $.X$ is an *EXPRESSION* **of** *PTR*(*integers*). The right hand side expression is described as $\uparrow.Y$ since in that context y denotes an integer value and $\uparrow.Y$ is an *EXPRESSION of integers*. The value of y at a certain stage of the program run, i.e. in an *HISTORY H*, is described by $(\uparrow.Y).H$.

We accept the obvious fact that expressions built of constants always have the same value, i.e. $(.C).H = C$.

In the description technique we will use expression forming binary operators $+$ and $*$ defined for *EXPRESSION* **of** *integers* and resulting in an *EXPRESSION* **of** *integers*.

The Pascal boolean expression is described as an *EXPRESSION* **of** *INTEGERS*. We use $+1$ for true and $+0$ for false. The only relational operator we will use later is that of nonequality and is described by the function *NE* which for *E1* and *E2* – each an *EXPRESSION* **of** *A*, for some *A* – results in an *EXPRESSION* **of** *integers*. We assume that:

$$E1.H \langle\rangle E2.H \textbf{ implies } NE(E1,E2).H = +1$$

and

$$E1.H = E2.H \textbf{ implies } NE(E1,E2).H = +0.$$

2.3.6. Statements – Description and Semantics. In this paragraph we show how Pascal statements are described to form axioms about a program. Any specific Pascal statement is described by a special predicate. The meaning of the predicate is formally defined and constitutes the assumed semantics of the statement. The meaning of the compound statement and the meaning of semicolon in the body of a Pascal program are expressed immediately in a description of the program obtained from the DESCRIBER.

To define semantics of a statement we have to characterize the transition from a history of a computation to its next history. Thus for any statement we have to define:

- the next control point,
- what has been changed by (or has remained unchanged after) the execution of the statement in the memory.

The Pascal assignment statement appearing at a control point of a program is described by the following predicate:

$$[LSE, RSE] \textbf{ is } ASSIGNMENT \textbf{ of } N$$

where

LSE – is a description of the left hand side expression,
RSE – is a description of the right hand side expression
N – is a natural number identifying the control point in a program where the assignment statement occurred.

For instance the assignment $x:=y$ occurring at the control point 3 of a program is described as:

$$[.X, \uparrow .Y] \textbf{ is } ASSIGNMENT \textbf{ of } 3$$

The meaning of the above predicate we define axiomatically and express in MIZAR 2 as follows (**st** for satisfying):

for A **being** *nonempty*, V **being** $(EXPRESSION$ **of** $PTR(A))$,
E **being** $(EXPRESSION$ **of** $A)$, K **being** *natural*,
H **being** $HISTORY$
st $[V,E]$ **is** $ASSIGNMENT$ **of** K & $CP(H)=K$
holds
$CP(NH(H))=K+1$ &
$VAL(V.H, NH(H))=E.H$ &
(**for** B **being** *nonempty*, V' **being** $EXPRESSION$ **of** $PTR(B)$
st $V'.H$ **is** $INDEPENDENT$ **of** $V.H$
holds $VAL(V'.H, NH(H))=VAL(V'.H, H)$).

In the axiom the first conjunct in the quantifier scope (i.e. after **holds**) defines the way of inserting control points after an assignment statement. We remind that in case of a specific program the control points are inserted by the DESCRIBER.

The second conjunct expresses what has been changed in the memory (*data* of the $HISTORY$ H) by the execution of the assignment statement. Thus, a part of the *data* of $NH(H)$ is defined.

The third conjunct defines what in the memory has remained unchanged. This fact is expressed with use of the predicate $INDEPENDENT$ defined for addresses. In general the predicate has a complex definition and we explain here only the case of declared variables. If $V=VAR(N,A)$, $V'=VAR(M,B)$ and $N\langle\ \rangle M$ then V **is** $INDEPENDENT$ **of** V'.

Remark. Since in this approach we discuss memory locations explicitly, we hope to treat properly also the cases of aliasing.
END of remark.

The **while** statement is described by the predicate $WHILE[K, EI, M]$ where:

K – a control point in a program where the **while** statement occurred,
EI – a description of the boolean expression guarding the **while** loop,

M – an additional control point inserted after the iterated statement to describe the return jump.

The diagram below indicates the places where the control points for a while statement are inserted:

$^{(K)}$ **while** ⟨expression⟩ **do** $^{(K+1)}$ ⟨statement⟩ $^{(M)}$.

Obviously there may be other control points inserted for the iterated statement but the way they are inserted depends only on the iterated statement – their numbers are from the interval $(K+1, M)$.

The semantics of the **while** statement is given in two axioms. In a separate axiom we define the control flow:

for H **being** *HISTORY*, K, M **being** *natural*,
EI **being** *EXPRESSION* **of** *integers*
st $WHILE\ [K, EI, M]$
holds
$(CP(H)=K$ **implies**
$((EI.H=+1$ **implies** $CP(NH(H))=K+1)$ &
$(EI.H=+0$ **implies** $CP(NH(H))=M+1))$ &
$(CP(H)=M$ **implies** $CP(NH(H))=K)$

The second axiom we assume for the **while** statement reflects the fact that no changes have taken place in the memory while passing through K and M control points:

for H **being** *HISTORY*, K, M **being** *natural*
EI **being** *EXPRESSION of integers*
st $WHILE\ [K, EI, M]$ & $(CP(H)=K$ **or** $CP(H)=M)$
holds
for A **being** *nonempty*, V **being** *element* **of** $PTR(A)$
holds $VAL(V, NH(H))=VAL(V, H)$

2.4. Example. Consider the following factorial computing program:

program *factorial* (*input*, *output*);
var x, y, z: *integer*;
begin *read* (x); $y:=1$; $z:=0$;
while $z<>x$ **do begin** $z:=z+1$; $y:=y*z$ **end**;
write (y);
end.

In the further discussion we omit problems related to the program heading and input/output statements. First, let us look how, according to the assumed axioms and our technique, the DESCRIBER inserts control points:

var x, y, z: *integer*;
begin
$^{(0)}$ $y:=1$; $^{(1)}$ $z:=0$;
$^{(2)}$ **while** $z<>x$ **do begin** $^{(3)}$ $z:=z+1$; $^{(4)}$ $y:=y*z$ $^{(5)}$ **end**;
$^{(6)}$ **end**.

Below we present a part of the above program description as obtained from the DESCRIBER.

take $X = VAR(1, integers)$;
take $Y = VAR(2, integers)$;
take $Z = VAR(3, integers)$;
$AXSTART$: $START = 0$;
$AX0$: $[.Y, .+1]$ **is** $ASSIGNMENT$ **of** 0;
$AX1$: $[.Z, .+0]$ **is** $ASSIGNMENT$ **of** 1;
$AX2$: $WHILE\ [2, NE(\uparrow .Z, \uparrow .X), 5]$;
$AX3$: $[.Z, (\uparrow .Z + .+1)]$ **is** $ASSIGNMENT$ **of** 3;
$AX4$: $[.Y, (\uparrow .Y * \uparrow .Z)]$ **is** $ASSIGNMENT$ **of** 4;
$AXFINISH$: FINISH $= 6$

The above description constitutes only a part of the axiomatics of our sample program. To obtain the full axiomatics and thus form an appropriate MIZAR 2 environment we have to add:

- declarations or definitions of the used notions,
- the axioms assumed about instruction meaning,
- axiomatically formed properties of used data types.

How this is done and how the desired properties of programs are proven is presented in Sects. 3 and 4.

One could argue at this point that our method consists in transforming Pascal programs into **goto** programs. That is true, the DESCRIBER actually simulates the work of a compiler (in our case of the Pascal 1900 compiler, cf. Welsh [8]) thus one may say we describe a Pascal program from the point of view of the code generated by the compiler. The second reason why the program description has this form is that Pascal, or at least its implementation, does not forbid destroying, by **goto** statements, all structure of the so called structural control statements.

3. Example of Factorial Computing Program

The program with inserted control points together with a part of its description obtained from the DESCRIBER was presented in the previous section. In that program there appeared only one Pascal (standard) type, namely *integer*. This type is described by MIZAR 2 (standard) type, namely *integers*. To go further with investigation of that program we have to enrich MIZAR 2 environment with some properties of integers which we expect to use later. Thus we add a scheme of induction and we declare the function $FACTORIAL$ axiomatically characterizing its properties. In MIZAR 2 it is expressed as follows:

scheme $INDUCTION$;
  pred P;
  for N **being** *element* **of** *integers* **st** $+0 <= N$ **holds** $P[N]$
    since
      $COND1$: $P[+0]$;
      $COND2$: **for** N **being** *element* **of** *integers* **st** $+0 <= N$ & $P[N]$
        holds $P[N + (+1)]$
end;

for N **being** *element* **of** *integers* **consider** $FACTORIAL(N)$ **being** *element* **of** *integers*;
AR1: $FACTORIAL(+0)=1$;
AR4: **for** N **being** *element* **of** *integers* **st** $+0<=N$ **holds**
$FACTORIAL(N)*(N+(+1))=FACTORIAL(N+(+1))$

Actually, the environment must also be augmented by some other arithmetical facts (characteristic for addition and multiplication) as MIZAR 2 knows nothing about arithmetics; they are not quoted here.

At this moment we have completed the environment part of the MIZAR 2 text and we are ready to formulate and prove certain of its properties. For the sake of simplicity we omit the problem of run time errors assuming that we have integer numbers implemented with no restriction to their range. We prove the following property of our program:

$+0<=VAL(X,FH)$ **implies**
(**ex** H **being** $HISTORY$ **st**
$CP(H)=FINISH$ & $VAL(Y,H)=FACTORIAL(VAL(X,FH))$)

This formula expresses the fact that for each run of our program if the value of x in the first (initial) history – where each program run starts – is greater or equal to 0 then there exists a history H such that its control point is $FINISH$ and the value of y is equal to the factorial of the initial value of x. In the other words we prove for that program the stop property and its input-output characteristics. Observe that we can also formulate (and prove) formulas involving histories at more than two control points. It allows us to deal with properties of the whole computational process, not only its initial and final states.

The above properties of the program are proved with the use of the following lemma:

for N **being** *element* **of** *integers* **st** $+0<=N$ **holds**
$N<=VAL(X,FH)$ **implies**
(**ex** H **being** $HISTORY$
st $CP(H)=2$ & $VAL(Z,H)=N$ &
$VAL(Y,H)=FACTORIAL(N)$)

The lemma follows from the scheme of induction and the inductive assumptions named $FIRSTCONDITION$ and $SECONDCONDITION$ which should subsequently be proved. The latter actually corresponds to the loop invariant.

$FIRSTCONDITION$:
$+0<=VAL(X,FH)$ **implies**
(**ex** H **being** $HISTORY$ **st** $CP(H)=2$ & $VAL(Z,H)=+0$ &
$VAL(Y,H)=FACTORIAL(+0)$).

The proof is immediate by considering the two first instructions of the program and *AR1*.

SECONDCONDITION:
for N **being** *element* **of** *integers* **st** $+0<=N$ &
$(N<=VAL(X,FH)$ **implies**
(**ex** H **being** $HISTORY$ **st** $CP(H)=2$ & $VAL(Z,H)=N$ &
$VAL(Y,H)=FACTORIAL(N)$))
holds
$N+(+1)<=VAL(X,FH)$ **implies**
(**ex** H' **being** $HISTORY$ **st** $CP(H')=2$ & $VAL(Z,H')=N+(+1)$ &
$VAL(Y,H')=FACTORIAL(N+(+1))$).

The proof of this formula we obtain easily by one pass through the while body.

A proof of the desired properties of this program was recorded in full in MIZAR 2 and its correctness was checked on the ICL 1900 implementation of MIZAR 2 system. The entire text as presented to the machine consisted of 300 lines in which the notation introduction and axiomatics occupied 144 lines. These numbers seem to be exceedingly high and some effort was undertaken to reduce the text volume. Some limitations of the checking abilities of the MIZAR 2 implementation had substantial influence on the text size; we are not going to discuss them here.

We focus the discussion on the general aspects of the description technique and on the description of the program. The DESCRIBER is now in a very early stage of development and its operation results in description like the above. Let us call the style of the description the Burstall style since it is very similar to what Burstall [1] proposed. Winkowski [9] uses a slightly different approach. Instead of describing one instruction by one predicate he characterizes an instruction by the definitional expansion of all axioms concerning the predicate in a form proper for the specific instruction. At the first glance this offers only the increase of description volume and adds no power. That is true but observe that in doing so we eliminate e.g. all expression descriptions from our considerations.

Using the Winkowski style we may better simulate the actual compiler work while interpreting a program in the predicate calculus. And we may hope to benefit much from it, namely we may have a possibility to better reflect the specific features of a particular program. Let us try to prepare the Winkowski style description of the factorial program by hand simulation of a describer.

The declaration of variables we now describe as follows:

consider X,Y,Z **being** *element* **of** PTR (*integers*);
AYZ: $Y\langle\ \rangle Z$;
$AXVAL$: **for** H **being** $HISTORY$ **holds** $VAL(X,H)=VAL(X,FH)$.

One must agree that even a not very "clever" describer would be able to generate the last axiom. In fact, many real compilers make use of the program properties like the one above expressed by AXVAL and are able find them automatically.

The two starting assignments of the factorial program are now described as:

$AX01$: $CP(NH(FH))=1$;
$AX02$: $VAL(Y,NH(FH))=+1$;
$AX03$: $VAL(Z,NH(FH))=VAL(Z,FH)$;

$AX11$: $CP(NH(NH(FH)))=2$;
$AX12$: $VAL(Z,NH(NH(FH)))=+0$;
$AX13$: $VAL(Y,NH(NH(FH)))=+1$;

Observe that using the Winkowski style we may, directly in the description, express the fact that the first two instructions are performed only once in every run of the program.

Remark on the Simultaneous Assignment

Assume that in a programming language we have the simultaneous assignment. Then the description of $(y,z):=(1,0)$ occurring in control point 0 is very simple:

AX': $CP(NH(FH))=1$;
AX'': $VAL(Y,NH(FH))=+1$;
AX''': $VAL(Z,NH(FH))=+0$

This is all since our program contains no other variable than y, z and x, but the value of x is constant throughout the program run.
End of remark.

Further, let symbol H denote a *HISTORY* and V an element of $PTR(integers)$. The **while** statement has a very simple description.

$AX21$: **for** H **st** $CP(H)=2$ **holds**
$(VAL(Z,H)<>VAL(X,H)$ **implies** $CP(NH(H))=3)$ &
$(VAL(Z,H)=VAL(X,H)$ **implies** $CP(NH(H))=FINISH)$;
$AX22$: **for** H,V **st** $CP(H)=2$ **holds** $VAL(V,NH(H))=VAL(V,H)$

Let us describe the last assignment in the program skipping that of control point 3. At the control point 4 we have the instruction $y:=y*z$.

$AX41$: **for** H **st** $CP(H)=4$ **holds** $CP(NH(H))=2$;
$AX42$: **for** H **st** $CP(H)=4$ **holds**
$VAL(Y,NH(H))=VAL(Y,H)*VAL(Z,H)$;
$AX43$: **for** H,V **st** $CP(H)=4$ & $V<>Y$ **holds**
$VAL(V,NH(H))=VAL(V,H)$

Note that thus we have eliminated the control point 5 from our considerations.

The MIZAR 2 text containing the description of the factorial program in the Winkowski style and the proof of the same program properties as before was more than twice shorter than the previous one – both in the description and the proof part. The whole of the new text is contained in the Appendix 1. The inductive assumption labelled *FIRSTCONDITION* and *SECONDCONDITION* are formed there in the so called natural deduction reasonings – they are not recorded explicitly.

4. Example of the List Reversing Program

We will investigate a program reversing an existing list in place. The program (with inserted control points) is:

```
    type lrec = record next: ↑lrec; cont: integer end;
    var laux, sr, tr: ↑lrec;
    begin
(0)   tr := nil;
(1)   while sr <> nil do
          begin
(2)           laux := sr↑.next;
(3)           sr↑.next := tr;
(4)           tr := sr;
(5)           sr := laux
(6)       end
(7)   end
```

In this program we assume the structural type equivalence. As previously we omit the problems related to the program heading and input/output. The reversing is done in place, i.e. the program changes only the values of the field *next* in the nodes of the list. The head of the source list is pointed to by *sr* at the beginning of computations, the head of the target list is pointed to by *tr* at the end of computations. We assume that the source list has the *nil* value in the field *next* of its last node. The fact will later be mirrored in the assumed induction scheme.

The case of this program is more complicated than the factorial computing program since list is not a standard Pascal type while integer is. Initially the Pascal **record** type is described as a certain nonempty set. In our case:

given *LREG* **being** *nonempty*.

The description of the record structure is done with use of an auxilliary function *FIELD* proposed by Byliński [2]. The function is declared as:

for *A* **being** *nonempty*, *V* **being** (*element* **of** *PTR*(*A*)),
N **being** *natural*, *B* **being** *nonempty*
consider *FIELD*(*A*, *V*, *N*, *B*) **being** *element* **of** *PTR*(*B*)

The meaning of its arguments is as follows:

- description of a **record** type (a set),
- a pointer (an address) to a value of the **record** type,
- a natural serving as field counter,
- type of the field.

The function *FIELD* results in an address to a value of the field type. The third argument of the function *FIELD* serves to express the independence of memory elements. The following holds:

N <> *M* **implies** *FIELD*(*A*, *V*, *N*, *B*) **is** *INDEPENDENT* **of** *FIELD*(*A*, *V*, *M*, *B'*)

In order to describe fields for a specific **record** type variable the DESCRIBER introduces subsequent auxiliary functions. For the fields of our **record** type *lrec* the DESCRIBER introduces the functions *NEXT* and *CONT*.

for V **being** *element* **of** $PTR(LREC)$
take
$NEXT(V)=FIELD(LREC,V,1,PTR(LREC))$,
$CONT(V)=FIELD(LREC,V,2,integers)$

Note that an object denoted by $NEXT(V)$ is an *element* **of** $PTR(PTR(LREC))$. Note also that the nodes of lists discussed are of the type $PTR(LREC)$. The remainder of the description obtained for the above program is:

take $LAUX=VAR(1,PTR(LREC))$;
take $SR=VAR(2,PTR(LREC))$;
take $TR=VAR(3,PTR(LREC))$;
$AXSTART$: $START=0$;
$AX0$: $[.TR,.NIL(LREC)]$ **is** $ASSIGNMENT$ **of** 0;
$AX1$: $WHILE\ [1,NE(\uparrow.SR,.NIL(LREC)),6]$;
$AX2$: **for** H **being** $HISTORY$ **st** $CP(H)=2$ **holds**
$[.LAUX,\uparrow.NEXT(\uparrow.SR.H)]$ **is** $ASSIGNMENT$ **of** 2;
$AX3$: **for** H **being** $HISTORY$ **st** $CP(H)=3$ **holds**
$[.NEXT(\uparrow.SR.H),\uparrow.TR]$ **is** $ASSIGNMENT$ **of** 3;
$AX4$: $[.TR,\uparrow.SR]$ **is** $ASSIGNMENT$ **of** 4;
$AX5$: $[.SR,\uparrow.LAUX]$ **is** $ASSIGNMENT$ **of** 5;
$AXFINISH$: $FINISH=7$

Since our program concerns lists certain axioms about lists must be added to the discourse. Here we mention only those axioms or theorems of a list theory which are necessary to solve our example. Thus we introduce the following notions and facts:

- A parametrized type:

type $LIST$ *of* A **being** *nonempty*

- A name for the empty list:

for A **being** *nonempty* **consider** $NILL(A)$ **being** $LIST$ **of** A

- An operation for appending an element in front of a list:

for A **being** *nonempty*, E **being** (*element* **of** A),
L *being* $LIST$ *of* A
reconsider $E.L$ **as** $LIST$ **of** A

- An axiom linking the introduced operation with the empty list:

$LIST1$: **for** A **being** *nonempty*, E **being** (*element of* A),
L **being** $LIST$ **of** A
holds $E.L\langle\ \rangle NILL(A)$

- An axiom scheme for induction over lists:

scheme $LISTIND$;
const A **being** *nonempty*;
pred P;
for L **being** $LIST$ **of** A **holds** $P[L]$
since
$COND1$: $P[NILL(A)]$;
$COND2$: **for** E **being** (*element* **of** A), L **being** $LIST$ **of** A
st $P[L]$ **holds** $P[E.L]$
end

Note that the scheme is valid only for linear finite lists.

- An operation for adding lists of the same type: $L+L'$.
- By $REV(L)$ we denote the reversed list L.
- E **is** *MEMBER* **of** L is the predicate saying that E is an element of the list L.
- L *MISSES* L' is a predicate saying that the lists L and L' are disjoint (have no common element).
- The property of appending the empty list:

  $NILL(A)+L=L \;\&\; L+NILL(A)=L$
- The reverse of the empty list is empty:

  $REV(NILL(A))=NILL(A)$
- The property of reversing a list:

  $REV(E.L)+L'=REV(L)+(E.L')$

The above notions and facts are not sufficient to prove or even to formulate any interesting property of our example. We have to add certain axioms saying how lists are represented in program notions or – which is equivalent – how from the program notions we might abstract lists. To solve problems related to that small example we assume the following notions and facts, mirroring programmer's intention:

- A function for denoting lists abstracted from the machine memory:

  for V **being** (*element* **of** $PTR(PTR(LREC))$), H **being** *HISTORY*
  consider $LIST(V,H)$ **being** *LIST of* $PTR(LREC)$

  In the subsequent axioms V denotes an *element* **of** $PTR(PTR(LREC))$, E an *element* **of** $PTR(LREC)$, L a *LIST* **of** $PTR(LREC)$ and H a *HISTORY*.
- An axiom describing the empty list representation:

  $LIST(V,H)=NILL(PTR(LREC))$ **iff** $VAL(V,H)=NIL(LREC)$
- An axiom about the nonempty list representation:

  $E.L=LIST(V,H)$ **iff**
  $E=VAL(V,H) \;\&\; L=LIST(NEXT(VAL(V,H)),H)$
- An assumption that all lists in our consideration are loopfree:

  not $VAL(V,H)$ **is** *MEMBER* **of** $LIST(NEXT(VAL(V,H)),H)$

We would like to warn the reader that the axiomatics chosen for lists was especially tailored to our example and does not pretend to be universally applicable. The notation built so far allows us to formulate the program property we are going to prove:

LISTPROGPROPERTY:
ex H **being** *HISTORY* **st**
$CP(H)=FINISH \;\&\; LIST(TR,H)=REV(LIST(SR,FH))$.

The above sentence expresses the following:

1. the stop property of the program, and
2. the fact that the list pointed to by *tr* at the end of a program run is actually the reversed list pointed to by *sr* at the start (i.e. in the first history FH) of the program run.

The theorem is finally formulated but to prove it we have to add something to our environment. Namely, we have to say how Pascal instructions influence the run-time program heap, and, consequently the lists abstracted from memory. We assume the following axioms, where V and V' denote an *element* **of** $PTR(PTR(LREC))$, H denotes an $HISTORY$, K, M are *natural* for control points, N denotes a *natural* and all free variables are to be treated as universally quantified bound variables.

$LRAX1$: $CP(H)=K$ & $V=VAR(N,PTR(LREC))$ &
$[.V,\uparrow.V']$ **is** $ASSIGNMENT$ **of** K
implies $LIST(V,NH(H))=LIST(V',H)$;

$LRAX2$: $CP(H)=K$ & **not** $\uparrow.V.H$ **is** $MEMBER$ *of* $LIST(V',H)$ &
$[.NEXT(\uparrow.V.H),\uparrow.V']$ **is** $ASSIGNMENT$ **of** K
implies $LIST(NEXT(\uparrow.V.H),NH(H))=LIST(V',H)$;

$LRAX3$: $CP(H)=K$ & **not** $\uparrow.V.H$ **is** $MEMBER$ **of** $LIST(V',H)$ &
$[.NEXT(\uparrow.V.H),\uparrow.V']$ **is** $ASSIGNMENT$ **of** K
implies $LIST(V',NH(H))=LIST(V',H)$;

$LRAX4$: $CP(H)=K$ & $[.V,\uparrow.V']$ **is** $ASSIGNMENT$ **of** K &
$V=VAR(N,PTR(LREC))$ & $V<>V'$
implies $LIST(V',NH(H))=LIST(V',H)$;

$LRAX5$: $WHILE[K,EI,M]$ & $(CP(H)=K$ **or** $CP(H)=M)$
implies $LIST(V,NH(H))=LIST(V,H)$

The above axioms are sufficient for that single program. Their general formulation would be quite a task if we wanted to cover all programs concerning lists. What we have done here is a counterpart of building a verification base for this example in the terminology of Luckham and Suzuki [4].

Now we have finally completed the preparation of notions and axioms necessary to prove some properties of the list reversing program. The proof of $LISTPROGPROPERTY$ is straightforward and makes use of the following lemma:

$LOOP$:
for H **being** $HISTORY$, L **being** ($LIST$ **of** $PTR(LREC)$)
st $CP(H)=1$ & $L=LIST(SR,H)$ & L $MISSES$ $LIST(TR,H)$
ex H' **being** $HISTORY$
st $CP(H')=1$ & $VAL(SR,H')=NIL(LREC)$ &
$LIST(TR,H')=REV(LIST(SR,H))+LIST(TR,H)$

The proof of $LOOP$ we obtain easily from $LISTIND$ scheme proving the inductive assumptions first. The proof of the second inductive assumption requires the characterization of the properties of the while loop during one pass through its body. The lemma named WHILEBODY describes this property.

$WHILEBODY$:
for H **being** $HISTORY$, E **being** (*element* **of** $PTR(LREC)$),
L **being** $LIST$ **of** $PTR(LREC)$
st $CP(H)=1$ & $E.L=LIST(SR,H)$ & $E.L$ $MISSES$ $LIST(TR,H)$
ex H' **being** $HISTORY$
st $CP(H')=1$ & $L=LIST(SR,H')$ &
$LIST(TR,H')=E.LIST(TR,H)$ &
L $MISSES$ $LIST(TR,H')$

Remark on Induction

Observe that due to the accepted approach to list representation we have to use induction in a rather uncomfortable way. Namely, while proving the second inductive assumption we are forced to prove *WHILEBODY* first and only then are we able to use the inductive hypothesis. Usually (cf. the previous example) we work in reverse order: we make use of the induction hypothesis first and then we prove the inductive step. In order to do so also in this case we would have to change lists to be identified by not only the head but also by the last node on the list. Such an approach is used for instance by Luckham and Suzuki [4].

A. Trybulec informed us that even in this case the straightforward application of induction is possible if we formulate the lemma *LOOP* as follows:

for L, K **being** $LIST$ **of** $PTR(LREC)$
  st $LIST(SR, NH(FH)) = REV(L) + K$
    ex H **being** $HISTORY$ **st**
      $CP(H) = 1 \,\&\, LIST(TR, H) = L \,\&\, LIST(SR, H) = K$

and later on we make use of the following fact:

$$LIST(SR, NH(FH)) = REV(REV(LIST(SR, NH(FH)))) + NILL(LREC)$$

One must admit that this is very "unnatural" but still works.
End of remark.

The description of the program presented so far was in the Burstall style. We may now try to shift to the Winkowski style simulating the describer work by hand. Here we extend the hand simulation in the comparison to the previous case. We are not going to discard the assumed similarity between an intended describer and a compiler. We would like the describer to prepare the description not only in terms of control points, histories and elementary storage changes but also in the terminology of *LIST*s. In this particular case it seems to be easily done and not only by hand. One can agree that a describer might prepare instantiations of axioms from *LRAX1* to *LRAX5* for any specific instruction in the list reversing program.

We present here the description for control points *0*, *1* and *3*. The whole of the environment and proof of the desired properties is contained in Appendix 2. (The formulation of the *WHILEBODY* lemma is incorporated in the reasoning approving the *SECONDSTEP* necessary for induction scheme application.) As in the case of the factorial computing program we express the notion of memory locations independence by means of the nonequality relation which entirely suffices in both cases. The control point *START*, here equal to *0*, is eliminated and we use FH, the first history, which is always controlled by the *START* control point. Symbol V will denote an *element* **of** $PTR(PTR(LREC))$ and H an *HISTORY*.

AX01: $CP(NH(FH)) = 1$;
AX02: $VAL(TR, NH(FH)) = NIL(LREC)$;
AX03: **for** V **st** $V \langle\rangle TR$ **holds** $VAL(V, NH(FH)) = VAL(V, FH)$;
AX04: **for** V **st** $V \langle\rangle TR$ **holds** $LIST(V, NH(FH)) = LIST(V, FH)$;

The above *AX04* axiom is an instantiation of *LRAX4*.

AX11: **for** H **st** $CP(H)=1$ **holds**
$(VAL(SR,H)<>NIL(LREC)$ **implies** $CP(NH(H))=2)$ &
$(VAL(SR,H)=NIL(LREC)$ **implies** $CP(NH(H))=FINISH$;
AX12: **for** V,H **st** $CP(H)=1$ **holds** $VAL(V,NH(H))=VAL(V,H)$;
AX13: **for** V,H **st** $CP(H)=1$ **holds** $LIST(V,NH(H))=LIST(V,H)$;

The last axiom is an instantiation of *LRAX5*.

AX31: **for** H **st** $CP(H)=3$ **holds** $CP(NH(H))=4$;
AX32: **for** H **st** $CP(H)=3$ **holds**
$VAL(NEXT(VAL(SR,H)),NH(H))=VAL(TR,H)$;
AX33: **for** V,H **st** $CP(H)=3$ & $V<>NEXT(VAL(SR,H))$ **holds**
$VAL(V,NH(H))=VAL(V,H)$;
AX34: **for** H **st** $CP(H)=3$ &
not $VAL(SR,H)$ **is** $MEMBER$ **of** $LIST(TR,H)$
holds $LIST(NEXT(VAL(SR,H)),NH(H))=LIST(TR,H)$;
AX35: **for** H,V **st** $CP(H)=3$ &
not $VAL(SR,H)$ **is** $MEMBER$ **of** $LIST(V,H)$
holds $LIST(V,NH(H))=LIST(V,H)$

Axioms *AX34* and *AX35* instantiations of *LRAX2* and *LRAX3* respectively.

Remark on Simultaneous Assignment

In the previous example we considered how much we would benefit from the simultaneous assignment instruction. In this case the possible application of such an instruction in the form:

$$(sr\uparrow.next,\ tr,\ sr):=(tr,\ sr,\ sr\uparrow.next)$$

eliminates the auxiliary variable *laux* but does not otherwise help. A straightforward description of the instruction, similarly as in the previous case, gives inconsistent instantiations of list representation axioms. To describe the simultaneous assignment properly we have to do it in the same way as in the case of a single assignment, otherwise we would have to change our approach essentially.
End of remark.

The shift from the Burstall to the Winkowski style also in this case resulted in a considerable abridgement of the MIZAR 2 text describing the program theory and containing the proof of the same property. In the former case it numbered 540 lines as compared with 250 lines in the latter. Obviously, even this second number seems too large to consider practical proving of bigger programs exactly in this manner. But we have to take into account that half of those texts (in lines) was or was assumed to be produced automatically by a describer or taken from a library. Please observe that the main reason for the proofs being so long are the abilities of the implemented MIZAR 2 checker, which are rather modest.

The factorial program example is comparatively much simpler than the list example so we devote more attention to the latter. We have chosen a list

example to give an idea of how to cope with user-defined data types in the presented approach. Our proposal is close to that of Luckham and Suzuki [4]. In that paper a verification approach is used, i.e. the proof is done by the machine while in our case the proof is prepared by man. What we have called an axiomatization of data structures representation, there is called a basis for verification. Luckham and Suzuki [4] present a complete system for proving programs with pointers, arrays and records in a uniform way while here we have only presented some preliminary ideas.

One of the approaches to abstract data types is delivering an implementation of the abstract data type in a programming language (or encapsulating the implementation in a module). That solution really makes reasoning about programs very easy and the approach presented here is not tailored for this aim. But in any case one must first prove the implementation to be correct. For this purpose we see the approach as extremely suitable. Observe, that our example cannot be expressed in any implementation of the abstract data type *list* unless there exists a function in the implementation which exactly performs the in-place reversal. We may easily write and prove a program to obtain a list reversed but not in the place of the source list, cf. Salwicki [6].

5. A bit of Dessert

The list reversing program is very often used by programmers, which included the authors. It is usually understood that the source list is a *nil* ending list; the program reverses it as proven. The work on proving the program has given us a new motive to prove programs at all. This is: while proving a program property we have a chance to observe what the program can actually do.

While proving the list reversing program we have realized that it processes also cyclic lists, terminates and results in changing their orientation; but that is not proven yet. So we have:

1. For a linear list

Fig. 1

we obtain

Fig. 2.

2. For a cyclic list

Fig. 3

we obtain

Fig. 4.

3. For a cyclic list with a "stem"

Fig. 5.

the program also terminates, and in this case we obtain

Fig. 6.

Of the programmers known to us and using that program none was conscious of this fact.

Acknowledgements. We are particularly indebted to Dr. A. Trybulec for his inspiration to write this paper and for a lot of helpful comments. Prof. J. Winkowski and Prof. A. Blikle helped us very much to revise a previous version of the text.

Reference

1. Burstall, R.M.: Formal description of program structure and semantics in first order logic. Machine Intelligence **5**, 79–98 (1969)
2. Byliński, C.: Natural semantics of Pascal. (Unpublished report in Polish, ICS PAS, 1981)
3. Jensen, K., Wirth, N.: PASCAL user manual and report. Berlin-Heidelberg-New York: Springer 1975
4. Luckham, D.C., Suzuki, N.: Verification of array, record and pointer operations in Pascal. ACM Trans. Progr. Lang. Syst. **1**, 226–244 (1979)
5. Oryszczyszyn, H.: Description of Pascal programs. (Unpublished report in Polish, ICS PAS, 1981)
6. Salwicki, A.: On the algorithmic theory of stacks. Fundamenta Informaticae, vol. III, pp. 311–331 (1980)
7. Trybulec, A., Byliński, C., Oryszczyszyn, H., Rudnicki, P.: MIZAR processor for ICL 1900, listing of Pascal program. 1981
8. Welsh, J.: Pascal for ICL 1900, listing of the compiler, Belfast 1977
9. Winkowski, J.: A natural method of proving properties of programs. Fundamenta Informaticae **1**, 33–49 (1977)
10. Winkowski, J.: Towards an understanding of computer simulation. Fundamenta Informaticae **1**, 277–289 (1978)

Received August 24, 1983/September 4, 1984

Comment on Appendices

The first (marked with dashes) part of a MIZAR 2 text is called an environment and contains definitions and axioms. The lines following the heading 'Program description' are presumed to be generated by a describer. The lines following the heading 'Description technology' are prepared once and by hand, and later on are appended (possibly in parts) to a particular problem. The fragments of environments pertinent to a particular domain, in our case to arithmetic and list theory, are to be prepared by hand (and may be used in many proofs). The actual proof follows the symbol *BEGIN* and is done by hand. The part of a line starting with == constitutes an informal commentary. The final *THANKS O.K.* is the checker signal confirming correctness of the proof.