Proving Properties of Pascal Programs in MIZAR 2

Piotr Rudnicki* and Wlodzimierz Drabent
Institute of Computer Science, Polish Academy of Sciences, P.O. Box 22, PL-00-901 Warsaw PLN, 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 consequence 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

At 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 $(\langle \text{control}, \langle \text{data} \rangle \rangle)$ 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

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

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

$$(\forall x, y)(x \in R \& \text{control}(x) = 1 \Rightarrow \text{control}(y) = 2 \& \text{data}(y) = \text{data}(x)).$$

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

$$(\forall x)(\text{control}(x) = 1 \Rightarrow \exists y(x \in R \& \text{control}(y) = 2 \& \text{data}(y) = \text{data}(x)).$$

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 $h' = \text{next}(h)$. Further we will in fact be concerned with a system equivalent to $M$ of the form

$$M' = (S^*, \text{next}).$$

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

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

is now

$$(\forall h)(\text{control(state}(h)) = 1 \Rightarrow \text{control(state(next}(h))) = 2 \& \text{data(state(next}(h))) = \text{data(state}(h))).$$
Scene effort has been made to automatize the process of deriving axioms which define the meaning of a program from the program text (Orechowicz [55]). The idea was to develop a software system called DESCRIPTER 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 DESCRIPTER 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 [13], 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 Bylinski [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 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

\textbf{type HISTORY}

and every objects 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 \texttt{HISTORY} \( H \) by \texttt{CP}(\( H \)) we denote the control point of \( H \) and by \texttt{NH}(\( H \)) the next history of \( H \) (Pascal programs are deterministic). In Mizar 2 the function symbols \texttt{CP} and \texttt{NH} are introduced in the following way:

\begin{itemize}
\item for \( H \) being \texttt{HISTORY} consider \texttt{CP}(\( H \)) being \texttt{natural};
\item for \( H \) being \texttt{HISTORY} consider \texttt{NH}(\( H \)) being \texttt{HISTORY}
\end{itemize}

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

\subsection*{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 [\textsuperscript{2}] 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 occurred in a program, except for standard types. For instance the standard Pascal type \texttt{integer} will be described by a set \texttt{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 \( A \) the elements of \( A \) and this set is denoted by \texttt{PTR}(\( A \)). For instance, a Pascal variable of type \( A \) is described as an element of \texttt{PTR}(\( A \)).

Due to a certain Mizar 2 restriction, for any set \( A \) we declare a unique \texttt{NIL}(\( A \)) being an element of \texttt{PTR}(\( A \)). Note that in Pascal the value \texttt{NIL} is compatible with any pointer type.

\subsection*{2.3.3. Variables} To each program variable of type \( A \) we assign an element of \texttt{PTR}(\( A \)) as its meaning. Declared Pascal variables are described with the help of the \texttt{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 \texttt{VAR} are declared in Mizar 2 as follows:

\begin{itemize}
\item for \( N \) being \texttt{natural}, \( A \) being nonempty
\item consider \texttt{VAR}(\( N, A \)) being element of \texttt{PTR}(\( A \))
\end{itemize}
For instance the declaration of variables

```pascal
var x, y, z: integer
```

is described as:

```pascal
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(integer). We assume that if \( X = VAR(X, A), Y = VAR(Y, B) \) and \( N \prec M \) then \( N \prec Y \). 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:

```pascal
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:

```pascal
EXPRESSION 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:

```pascal
(\( \uparrow E \), 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 a
an element of \( A \) (a value of type \( A \)) by \( \cdot \) \( C \) denote the EXPRESSION of \( A \) built from \( C \). E.g. \( \cdot +3 \) is an element of \( \) integer; while \( \cdot . +3 \) is an EXPRESSION of \( \) integer; \( X \) is an element of \( \) PTR(integer) and \( \cdot . X \) is an EXPRESSION of \( \) PTR(integer).

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(integer). The right hand side expression is described as \( \cdot x \) since in that context \( x \) denotes an integer value and \( x \) is an EXPRESSION of \( \) integer. The value of \( y \) at a certain stage of the program run, i.e. in an HISTORY \( H \), is described by \( (\cdot y), H \).

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

In the description technique we will use expression forming binary operators \( + \) and \( * \) defined for EXPRESSION of \( \) integer; and resulting in an EXPRESSION of \( \) integer.

The Pascal boolean expression is described as an EXPRESSION of \( \) integer. We use \( +1 \) for true and \( +0 \) for false. The only relational operator \( \) we will use later is that of inequality and is described by the function \( \) \( \) true \( \) which for \( E1 \) and \( E2 \) – each an EXPRESSION of \( A \), for some \( A \) – results in an EXPRESSION of \( \) integer. We assume that:

\[ E1, H \prec E2, H \text{ implies } NE(E1, E2), H = +1 \]

and

\[ E1, H \succ E2, H \text{ implies } NE(E1, E2), H = +0. \]

2.3.6. Statements - Description and Semantics. In this paragraph we show how Pascal statements are described by 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 semi-colon 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:

\([\text{LSE}, \text{RSE}]\) is \text{ASSIGNMENT} of \(N\)

where

\(\text{LSE}\) – is a description of the left hand side expression,

\(\text{RSE}\) – 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 \(x \leftarrow y\) occurring at the control point 3 of a program is described as:

\([x, 1, y]\) is \text{ASSIGNMENT} of 3

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

\[
\begin{align*}
&\text{for } A \text{ being nonempty, } V \text{ being } (\text{EXPRESSION of } \text{PTR}(A)), \\
&\quad E \text{ being } (\text{EXPRESSION of } A), K \text{ being natural,} \\
&\quad H \text{ being } \text{HISTORY,} \\
&\quad \text{st } [V; E] \text{ is } \text{ASSIGNMENT of } K & & \text{& } \text{CP}(H) = K \\
&\quad \text{holds } \\
&\quad \text{CP}(\text{NH}(H)) = K + 1 & & \text{& } \text{VAL}(V; H, \text{NH}(H)) = E, H & & \text{& } \text{for } B \text{ being nonempty, } V' \text{ being } \text{EXPRESSION of } \text{PTR}(B) \\
&\quad \text{st } V; H \text{ is } \text{INDEPENDENT of } V', H & & \text{holds } \text{VAL}(V'; H, \text{NH}(H)) = \text{VAL}(V; H, H).
\end{align*}
\]

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 \(V = \text{VAR}(N, A), V' = \text{VAR}(M, B)\) and \(N \subseteq M\) then \(V\) is \text{INDEPENDENT of } V'.

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

The \text{while} statement is described by the predicate \text{WHILE}*[K, EI, M]* where:

\(K\) – a control point in a program where the \text{while} statement occurred,

\(EI\) – a description of the boolean expression guarding the \text{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:

\[\text{while } \langle \text{expression} \rangle \text{ do } \langle \text{statement} \rangle \]

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:

\[
\begin{align*}
\text{for } & H \text{ being HISTORY, } K, M \text{ being natural,} \\
& E_I \text{ being EXPRESSION of integers} \\
& \text{st WHILE } [K, E_I, M] \text{ holds} \\
& (C_P(H) = K) \implies \langle E_I, H = +1 \implies C_P(N(H(H)) - K + 1) \& \rangle \\
& \langle E_I, H = +0 \implies C_P(N(H(H)) - M + 1) \& \rangle \\
& (C_P(H) = M) \implies C_P(N(H(H)) = K) \\
\end{align*}
\]

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:

\[
\begin{align*}
\text{for } & H \text{ being HISTORY, } K, M \text{ being natural} \\
& E_I \text{ being EXPRESSION of integers} \\
& \text{st WHILE } [K, E_I, M] \text{ holds} \\
& (C_P(H) = K \text{ or } C_P(H) = M) \\
& \text{holds } VAL(V, N(H(H)) = VAL(V, H)) \\
\end{align*}
\]

2.4. Example. Consider the following factorial computing program:

\[
\text{program factorial (input, output);} \\
\quad \text{var } x, y, z : \text{integer}; \\
\quad \begin{align*}
& \text{begin read } (x); y := 1; z := 0; \quad \text{while } z < x \text{ do begin } z := z + 1; y := y \times z \text{ end;} \\
& \quad \text{write } (y); \quad \text{end;}
\end{align*}
\]

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

\[
\begin{align*}
\text{var } & x, y, z : \text{integer}; \\
& \text{begin} \\
& (0) \quad y := 1; z := 0; \\
& (1) \quad \text{while } z < x \text{ do begin } z := z + 1; y := y \times z \text{ end;}
\end{align*}
\]

Below we present a part of the above program description as obtained from
the DESCRIBER.
take X = VAR3(integer);
take Y = VAR2(integer+);
take Z = VAR3(integer+);
\AxStart: \text{START} = 0;
\Ax0: \{., Y, +\} is ASSIGNMENT of 0;
\Ax1: \{., Z, +0\} is ASSIGNMENT of 1;
\Ax2: \text{WHILE} [\text{or}(., Z, =, \text{X} + \text{Y})];
\Ax3: \{., Z, (., Z, =, +1)\} is ASSIGNMENT of 1;
\Ax4: \text{FINISH} = \text{FINISH} + 6

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

- declarations of 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 de-
dscription obtained from the DESCRIBER was presented in the previous sec-
tion. In this 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 the 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 \textsc{factorial},
axiomatically characterizing its properties. In MIZAR 2 it is expressed as follows:

scheme INDUCTION;
  \text{prof} P;
  \text{for} N \text{ being element of integers st } +0 < N \text{ holds } P[N]
  \text{since}
  \text{COND1: } P[+0];
  \text{COND2: for } N \text{ being element of integers st } +0 < N \text{ & } P[N]
  \text{ holds } P[N + N]
end.
for $N$ being element of integers consider $\text{FACTORIAL}(N)$ being element of integers;

**AR1**: $\text{FACTORIAL}(0) = 1$;

**AR4**: for $N$ being element of integers st $0 < N$ holds

$\text{FACTORIAL}(N) = (N + 1) \cdot \text{FACTORIAL}(N + 1)$

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:

$\forall x \in \mathbb{N}, 0 < x \Rightarrow \text{FACTORIAL}(x)$

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 $\text{FINISH}$ 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

$N = \text{FACTORIAL}(x)$ implies

(ex $H$ being HISTORY st $C(P(H)) = 2 \land \text{VALIZ}(H) = N \land \text{VALIZ}(H) = \text{FACTORIAL}(N))$

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

**FIRSTCONDITION**:

$\forall x \in \mathbb{N}, 0 < x \Rightarrow \text{FACTORIAL}(x)$ implies

(ex $H$ being HISTORY st $C(P(H)) = 2 \land \text{VALIZ}(H) = 0 \land \text{FACTORIAL}(0)$

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

for \( N \) being element of integers \( s.t. \ 0 < N \) &
\( (N < val(x, f, h)) \) implies
\( (ex \ H \ being \ HISTORY \ at \ cp(h) = 2 \ & \ val(z, h) = N \ & \ val(x, h) = \text{FACTORIAL}(N)) \)

holds
\( N + (1) < val(x, f, h) \) implies
\( (ex \ H \ being \ HISTORY \ at \ cp(h) = 2 \ & \ val(z, h) = N + (1) \ & \ val(y, f, h) = \text{FACTORIAL}(N + (1)) \).

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 DESCRIPTER 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 character-
izes 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 \( \text{PTR}(\text{integers}) \):
\( \forall Z . X \subseteq Y \subseteq Z \).

\( \text{AXVAL} \) for \( h \) being \( \text{HISTORY} \) holds \( val(x, h) = val(y, f, h) \).

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 \( \text{AXVAL} \) and are able find them
automatically.

The two starting assignments of the factorial program are now described as:
AX01: CP(N(E(FB))) = t;
AX02: VALX.NH(FH) = + 1;
AX03: VALZ.NH(FH) = VALZ.FH;
AX11: CP(NH(NH(FH))) = 2;
AX12: VALZ.NH(NH(FH)) = + b;
AX13: VALN.NH(NH(FH)) = + 7;

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 \((y, z) = (t, 0)\) occurring in control point 0 is very simple:

\[ \text{AX': } CP(N(HN(FH))) = t; \]
\[ \text{AX': } VALX.NH(NH(FH)) = + 1; \]
\[ \text{AX': } VALZ.NH(FH) = + 0 \]

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 \(\text{HISTORY}\) and \(V\) an element of \(\text{PT} \text{(INTEGER)}\). The \text{while} statement has a very simple description.

\[ \text{AX21: for } H \text{ at } CP(H) = 2 \text{ holds } \]
\[ (VALZ.H) = 0 \text{ implies } CP(N(H)) = 3) \& \]
\[ (VALZ.H) = 0 \text{ implies } CP(N(H)) = \text{FINISH}; \]
\[ \text{AX22: for } H \text{ at } CP(H) = 2 \text{ holds } VAL(NH(NH)) = VAL(H); \]

Let us describe the last assignment in the program skipping that of control point 3. At the control point 4 we have the instruction \(y = y + z\).

\[ \text{AX41: for } H \text{ at } CP(H) = 4 \text{ holds } CP(N(H)) = 7; \]
\[ \text{AX42: for } H \text{ at } CP(H) = 4 \text{ holds } \]
\[ VAL(NH(NH)) = VAL(H) \& VALZ.H; \]
\[ \text{AX43: for } V \text{ at } CP(H) = 4 \& z < y \text{ holds } \]
\[ VALV.NH(E) = VALV.H; \]

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 I. The inductive assumption labelled \text{FIRSTCONDITION} and \text{SECONDCONDITION} are formed there in the so called natural deduction reasons — 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; ↑ rec; 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
      end
  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 auxiliary 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 as 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 ( ≠ 0) 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 DE-
SCRIBER 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 $\text{PTR}(\text{LREC})$
take $\text{NEXT}(V) = \text{FIELD}(\text{LREC}, V, 1, \text{PTR}(\text{LREC})).$
$\text{CONT}(V) = \text{FIELD}(\text{LREC}, V, 2, \text{integers})$

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

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

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:
  $\text{type LIST of } A \text{ being nonempty}$
  - A name for the empty list:
    - for $A$ being nonempty consider $\text{NIL}(A)$ being LIST of $A$
  - An operation for appending as element in front of a list:
    - for $A$ being nonempty, $E$ being (element of $A$),
      $\text{L}$ being LIST of $A$
      reconsider $E, \text{L}$ as LIST of $A$
      - An axiom linking the introduced operation with the empty list:
        - $\text{LIST1}: \text{for } A \text{ being nonempty, } E \text{ being (element of } A),$
          $\text{L}$ being LIST of $A$
          holds $E, \text{L} \subset \text{NIL}(A)$
    - An axiom scheme for induction over lists:
      - scheme $\text{LISTIND}:$
        - const $A$ being nonempty;
        - pred $P;$
        - for $L$, being LIST of $A$ holds $P[L]$
        - since
          - $\text{COND1}: P[\text{NIL}(A)];$
          - $\text{COND2}: \text{for } E \text{ being (element of } A), \text{L} \text{ being LIST of } A$
            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 \( L \).
- \( L \) is \( MEMBER \) of \( L \), is the predicate saying that \( E \) is an element of the list \( L \).
- \( L \) \( MISS \) \( E \) is a predicate saying that the lists \( L \) and \( E \) are disjoint (have no common element).
- The property of appending the empty list:
  \( NIL(L) + L = L \) & \( L + NIL(L) = L \)
- The reverse of the empty list is empty:
  \( REV(NIL(L)) = NIL(L) \)
- The property of reversing a list:
  \( REV(L \cdot E) = REV(L) \cdot (E \cdot L) \)

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 intentions:

- A function for denoting lists abstracted from the machine memory:
  
  \textbf{for } \textbf{if } \textbf{being } \textbf{element of } \textbf{PTR(PTR(L.REC))}, \textbf{H being HISTORY consider } \textbf{LIST(V, H) being LIST of PTR(L.REC)}.
  
  In the subsequent axiom \( V \) denotes an element of \( PTR(PTR(L.REC)) \), \( E \) an element of \( PTR(L.REC) \), \( L = \textbf{LIST of PTR(L.REC)} \), \( H = \textbf{HISTORY} \).

- An axiom describing the empty list representation:
  \( \text{LIST}(V, H) = \text{NIL}(PTR(L.REC)) \iff \text{VALUE}(H) = \text{NIL}(L.REC) \)

- An axiom about the nonempty list representation:
  \( \text{LIST}(V, H) \iff \text{VALUE}(H) \neq \text{NIL} \land \text{LIST}(\text{NEXT}(\text{VALUE}(H)), H) \)

An assumption that all lists in our consideration are loop-free.

\textbf{not VALUE(H) is MEMBER of LIST(NEXT(VALUE(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:

\textbf{LISTPROPPROPERTY:}

\textbf{ex } \textbf{if } \textbf{being } \textbf{HISTORY of}

\( CPS(H) = \text{FINISH} \\ \& \text{LIST}(\text{TR}, H) = REV(\text{LIST}(\text{SR}, F(H))) \)

The above sentence expresses the following:

1. the stop property of the program, and

2. the fact that the list pointed to by \( x \) at the end of a program run is actually the reversed list pointed to by \( y \) at the start (i.e., in the first history \( F(H) \) 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 list abstracted from memory. We assume the following axioms, where \( V \) and \( V' \) denote an element of \( PTR(PTR(LREC)) \). \( H \) denotes as \( HISTORY \), \( K \), \( M \) are natural for control points, \( N \) denotes a natural and all free variables are to be treated as universally quantified bound variables.

\[
\begin{align*}
LAX1: & \quad CP(H) = K \land V' = L(R(N, PTR(LREC)) \land \{[V', V']\} \text{ is ASSIGNMENT of } K \\
& \implies LIST(V, NH(H)) = LIST(V', H) \\
LAX2: & \quad CP(H) = K \land \text{ not } \{[V', V] \text{ is MEMBER of } LIST(V', H) \land \\
& \quad \{\text{NEXT}(V', V), H\} \text{ is ASSIGNMENT of } K \\
& \implies LIST(\text{NEXT}(V', V), NH(H)) = LIST(V', H) \\
LAX3: & \quad CP(H) = K \land \text{ not } \{[V', V] \text{ is MEMBER of } LIST(V', H) \land \\
& \quad \{\text{NEXT}(V', V), H\} \text{ is ASSIGNMENT of } K \\
& \implies LIST(V, NH(H)) = LIST(V', H) \\
LAX4: & \quad CP(H) = K \land \{[V', V] \} \text{ is ASSIGNMENT of } K \land \\
& \quad r = \Delta NR(V, PTR(LREC)) \land V = V' \\
& \implies LIST(V, NH(H)) = LIST(V', H) \\
LAX5: & \quad WHILE(K, E, M) \land (CP(H) = K \land CP(H') = M) \\
& \implies LIST(V, NH(H)) = LIST(V, H)
\end{align*}
\]

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 equations and axioms necessary to prove some properties of the list reversing program. The proof of \( LISTPROPERTY \) is straightforward and makes use of the following lemma:

\[
\text{LOOP:}
\]

\[
\text{for } H \text{ being } HISTORY, L \text{ being } \{\text{LIST of PTR(LREC)}\} \\
\text{st } CP(H) = 1 \land L = \text{LIST(SR.H)} \land L \text{ MISSES LIST(TR.H)} \\
ex \quad H \text{ being HISTORY} \\
\text{st } CP(H) = 1 \land L = \text{VALUE(SR.H)} = NH(LREC) \\
\text{LIST(TR.H) = REV(\text{LIST(SR.H)}) + LIST(TR.H)}
\]

The proof of \( \text{LOOP} \) we obtain easily from \( \text{LISTIND} \) scheme proving the inductive assumptions first. The proof of the second inductive assumption requires the characterization of the properties of the while loop using one pass through its body. The lemma named \( \text{WHILEBODY} \) describes this property.

\[
\text{WHILEBODY:}
\]

\[
\text{for } H \text{ being } HISTORY, E \text{ being } \{\text{element of PTR(LREC)}\}, \\
\text{L \text{ being } \{\text{LIST of PTR(LREC)}\} \\
\text{st } CP(H) = 1 \land E, L = \text{LIST(SR.H)} \land L \text{ MISSES LIST(TR.H)} \\
ex \quad H \text{ being HISTORY} \\
\text{st } CP(H) = 1 \land L = \text{LIST(SR.H)} \\
\text{LIST(TR.H) = E, LIST/TR.H} \\
\text{L \text{ MISSES LIST/TR.H}}
\]
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 WHILER0D1 first and only then are we able to use the inductive hypothesis. Usually, 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 Suenzi [4].

A. Ttrybulec informed us that even in the case of the straightforward application of induction it is possible if we formulate the lemma E00V as follows:

\[
\text{for } L, K \text{ being LIST of } PTR(LREC) \text{ st}
\]

\[
\text{LJST(SR,NH(FH)) = REV}(L) = K
\]

\[
\text{ex } H \text{ being HISTORY st}
\]

\[
CP(H) = L \& \text{LJST}(TR,H) = L \& \text{LJST}(SR,H) = K
\]

and later on we make use of the following fact:

\[
\text{LJST}(SR,NH(FH)) = REV}(\text{REV}(L)) = \text{REV}(L) = \text{NIL}(LREC)
\]

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 Backus 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 LISTS. In this particular case it seems to be clearly done and not only by hand. One can agree that a describer might prepare instantiations of axioms from LRA1 to LRA5 for any specific instruction in the list processing 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 WHILER0D1 lemma is incorporated in the reasoning approving the E2CONDSTEP necessary for induction schema application.) As in the case of the factorized computing program we express the notion of memory locations independence by means of the nonuniqueness 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 PTER(PTR,LREC) and H an HISTORY.

AX01: \( CP(NH(FH)) = L \);

AX02: \( \text{VAL}(TR,NH(FH)) = \text{NIL}(LREC) \);

AX03: \( \text{for } V \text{ st } V \& \text{TR holds } \text{VAL}(V,NH(FH)) = \text{VAL}(V,FH) \);

AX04: \( \text{for } V \text{ st } V \& \text{TR holds } \text{LJST}(V,NH(FH)) = \text{LJST}(V,FH) \);

The above AX04 axiom is an instantiation of LRA4.
AX11: for $H$ st $CP[H] = 1$ holds

\[ VALISR[H] \times NH[H] = VALIN[H] \times LEC \] implies $CP[NH[H]] = 2 \& $ \&

\[ VALISR[H] = NIL[LEC] \] implies $CP[NH[H]] = FINISH; $

AX12: for $V, H$ st $CP[H] = 1$ holds $VALIN[NH[H]] = VALIN[V, NH[H]]$

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

\[ VALIN[NEXT[VALISR[H], NH[H]]] = VALIN[TR[H]] \]

AX33: for $V, H$ st $CP[H] = 3 \& V \# NEXT[VALISR[H]]$ holds

\[ VALIN[NH[H]] = VALIN[V, H]; \]

AX34: for $H$ st $CP[H] = 3 \&

\[ valIN[SR[H]] \text{ is MEMBER of LIST[TR[H]]} \]

holds $LIST[NEXT[VALISR[H], NH[H]]] = LIST[TR[H]]; $

AX35: for $V, H$ st $CP[H] = 3 \&

\[ valIN[SR[H]] \text{ is MEMBER of LIST[V, H]} \]

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:

\[ (v, w, x, y) = (v, w, x, y) \]

eliminates the auxiliary variable proof 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 improvement 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 this half of those texts (i.e. lines) was in was assumed to be produced automatically by a descirber 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 readily 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 not 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 if 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 "start"

Fig. 5.

the program also terminates, and in this case we obtain

Fig. 6.

Of the programmers known to us and using this program none was conscious of this fact.
Acknowledgments. We are particularly indebted to Dr. A. Trybodec for his inspection as we write this paper and for a lot of helpful suggestions. Prof. J. Wiorowski and Prof. A. Wiatr helped us very much to revise a previous version of the text.

Reference

Received August 24, 1983; September 4, 1984.

Comment on Appendices
The first (marked with a square) part of a Mizar 2 text is called an environment and contains definitions and axioms. The lines following the heading "Program description" are presented to be generated by a deserializer. The lines following the heading "Description technology" are prepared once and by hand, and later on are appended precisely to parts to a particular problem. The diagnostic of environments permits to a particular language, in our case in arithmetic and list theory, are to be prepared by hand (and may be used in many proofs). The actual good follows the symbol BEGIN and is done by hand. The part of a line starting with "=" constitutes an informal commentary. The final TRAVERS OK is the checker signal confirming correctness of the proof.