aut 81

Department of Mathematics
University of Technology
Eindhoven, The Netherlands
tel 040-472775

RELATIONAL SEMANTICS IN AN INTEGRATED SYSTEM

BY

R.M.A. Wieringa

0.ABSTRACT.

This paper contains the description of a system for handling semantics of computerprograms. The methodology used for the description of semantics is the relational semantics: - possibly incomplete - information about programs is represented by binary relations.

For the description we use the language AUTOMATH in which logic, mathematics, syntax and semantics are integrated. Moreover, the correctness of texts written in AUTOMATH can be checked mechanically by a computer.

We consider an ALGOL60-like programming language. The axiomatic basis of it is kept small, but it is large enough to make the definition of many ALGOL constructs possible. In the basis are included assignment, binary selection, concatenation, block structures and recursive parameterless procedures.

For these basic constructs semantics is presented, and some examples are given how new program constructs can be described in terms of these basic ones.

1. INTRODUCTION.

We shall present a formalism for the description of syntax and semantics of programs in an ALGOL-60-like programming language (i.e. a block-structured programming language with variables of various kinds, assignment to these variables, binary selection, recursion, etc.). An essential point is that program correctness proofs have to be subjected to an automatic verification system. So we have to deal with

a. the organisation of the variables, the so called state space,
b. the description of the syntax of the language: what kind of programs do we consider,
c. the description of the semantics of the programs: what information do we state about the programs.

The method we shall use to describe semantics will be relational semantics with strong emphasis on dealing with incomplete information about the relation between initial and final state.

A system for verification of the correctness of programs has to be able to cope with mathematical theories (e.g. number theory) and to keep track of the mathematical interpretation of values of state space variables.

In practice, the verification of the correctness of a program appears to be long and tedious, since it consists of very many elementary steps. We feel the need for a mechanical verification.

So alltogether, we need a language in which various formal systems (e.g. semantics, logic, mathematics) are integrated, and the correctness of what is written in the language should be decidable by a computer.

AUTOMATH ([1,2]) is such a wide-scope language. In an AUTOMATH book we can express all primitives we need about logic, mathematics, programming language, semantics, and on the basis of these primitives we can define particular programs and derive truths about their semantics.

We use the following notation for some of the essentials of AUTOMATH. Typing is denoted by colons (P:Q means P has type Q). Abstraction is written as [x:A]B , denoting the function with domain A and values B (this B may contain x). Application is written as <A>B (i.e. the value of the function B at the point A). We use Q:type for saying that Q is a type, and R:prop for saying that R represents a proposition (if S:R then S is a proof of that proposition).

The semantical framework described here is essentially based on various proposals by N.G de Bruijn [3,4]. In the present form it is used by the author of this paper for the development of an operational system intended to be useful for proving correctness of big programs.

2. THE STATE SPACE.

Since programs act on variables, we have to pay some attention to these variables and their possible values; in other words, to the state space. Roughly speaking a state is a set of variables each of a certain type (think e.g. on the types integer, boolean etc in ALGOL60) and having a value corresponding to that type. So we introduce the notion datatype, and several datatypes, like

```
datatype : type
bool : datatype
int : datatype
```

For each datatype dt the type of the corresponding values will be denoted by

```
elts(dt) : type
```

Since our programming language has an ALGOL-like block structure, we put our variables on stacks: one for each datatype. For simplicity we do not assume the stacks to have a bottom. The places in each stack are indexed by 0, 1, 2, ...; the 0 refers to the top of the stack. In the stack corresponding to dt the values have type elts(dt). Each pair (dt,i) of a datatype and an index now identifies a program variable: we do not talk about names of variables.

So we define (written in AUTOMATH)

```
State := [dt:datatype][i:nat]elts(dt) : type
```

(where nat is the type of the naturals). For a visual interpretation see fig. 2.1.

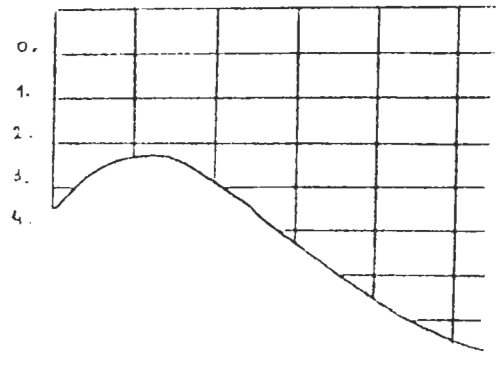

fig 2.1.
A state space

There are several operations on states. Let us fix a state σ.

By value(σ,dt,i) we denote the value in σ of the variable (dt,i); it has type elts(dt).

Furthermore there are some operations transforming states into states:

a. adapt(σ,dt,i,v) is the state that is obtained from σ by replacing the value of (dt,i) by a new value v;

b. extend(σ,dt,v) is the state we get when in σ we push an element with value v on the stack corresponding to dt, So, when σ'=extend(σ,dt,v) we have value(σ',dt,0)=v, value(σ',dt,i+1)=value(σ,dt,i), value(σ',t,i)=value(σ,t,i) when t≠dt;

c. restrict(σ,dt) is the state we get when in σ we remove the top element of the stack corresponding to dt. So when σ'=restrict(σ,dt) we have value(σ',dt,i)=value(σ,dt,i+1), value(σ',t,i)=value(σ,t,i) when t≠dt.

Having defined these operations on states, we can prove properties about them, e.g.

$$\text{value}(\text{extend}(\sigma,dt,v),dt,i+1) = \text{value}(\sigma,dt,i)$$
$$\text{restrict}(\text{extend}(\sigma,dt,v),dt) = \sigma$$

and write these in our AUTOMATH book.

In order to deal with nontermination, abortion because of thing like "devide by zero", indexing outside array bounds etc., we add an extra datatype ref (standing for refuser). The variables belonging to ref are quasi-variables, i.e. they do not appear in a program itself but only in its semantics. There are two values connected to refusers: ON and OFF, ON meaning "there is something wrong". The datatype ref plays an exceptional role in our discussions. In most cases we shall stipulate that datatypes are ≠ ref.

3 SYNTAX.

What kind of programs do we consider? It is our intention to have a rich class of programs with a set of primitives that is as small as possible. Therefore we do not consider expressions of complex shape in our primitive programs. The following programs are primitive (the word "Program" will be used as the type of all programs).

1. If dt:datatype, u:dt≠ref (i.e. u proves dt≠ref), i:nat, v:elts(dt), we have

   Const_ass(dt,u,i,v) : Program

   corresponding to "x:=v" in ALGOL (where x corresponds to (dt,i) and v is a constant of type dt).

2. If dt:datatype, u:dt≠ref, i1:nat, i2:nat, we have

   Var_ass(dt,u,i1,i2) : Program

   corresponding to "x:=y" in ALGOL, y being a variable.

3. If b:nat, π1:Program, π2:Program, we have

   Bin_select(b,π1,π2) : Program

   corresponding to "if b then π1 else π2".

4. If π1:Program, π2:Program, we have

   Concat(π1,π2) : Program

   corresponding to "π1;π2".

5. If dt:datatype, u:dt≠ref, π:Program, we have

   Block(dt,u,π) : Program

   corresponding to "begin dt x; π end" (where dt is one of the types in ALGOL).

6. If dt:datatype, u:dt≠ref, π:Program, we have

   Injection(dt,u,π) : Program

   In ALGOL there is no construction corresponding to this. It intends the following: Program π acts on a state space. When we want to use π in a situation where that state space has been extended with a variable of datatype dt, π has to act on that extended state space. for formal reasons this program has to get a new name.

7. If φ:Program -> Program (i.e. φ is a function from programs to programs) we have

   Recurs(φ) : Program

   more or less corresponding to "procedure p; <p>φ"

The idea behind this approach is the following: ALGOL uses in recursive procedures a kind of circular definition: in the specification of procedure p, p itself may appear: p:=<p>ϕ. The essential part of the procedure is ϕ, the program-program function. by the formula p := Recurs(ϕ) we turn ϕ into a program.

The above list of primitive programs is a reasonable basis for a programming language. We do not state it to be complete; if desirable we can add further primitives later, e.g. primitives about array assignment, and operations on records (as in PASCAL). And users of the system, handling special algorithms requiring special datatypes can add primitive notions for private use.

By means of the seven primitive program constructs given above we can build other program constructs. Once they have been written in our AUTOMATH book they are available for later use, just like the primitive ones. We give some examples.

8. To the boolean assignment "b1:=b2∨b3" in ALGOL (where b1,b2 and b3 are variables) corresponds the statement "if b2 then b1:=true else b1:=b3". It is written in AUTOMATH as follows: if b1:nat, b2:nat, b3:nat, we define

```
Bool_or_ass(b1,b2,b3) := Bin_select(b2,Const_ass(bool,boolnotref,b1,T),
                         Var_ass(bool,boolnotref,b1,b3)) : Program
```

(where boolnotref states bool≠ref, and T:elts(bool) denotes the value true).

9. The empty statement in ALGOL can be mimicked as follows:

```
Dummy := Var_ass(bool,boolnotref,0,0) : Program
```

so "b:=b" in ALGOL where b corresponds to (bool,0).

10. To the statement "if b1∨b2 then π" corresponds the block "begin boolean b; b:=b1∨b2; if b then π else end". If b1:nat,b2:nat, π:Program, we describe it by

```
Or_cond(b1,b2,π) := Block(bool,boolnotref,Concat(Bool_or_ass(0,
                    b1+1,b2+1),Bin_select(1,Injection(bool,boolnotref,π),
                    Dummy))) : Program.
```

Notice the effect of the introduction of a new boolean. It transforms b1, b2 and π into b1+1, b2+1 resp. Injection(bool,boolnotref, π).

11. To the while statement "while b do π" corresponds the recursive procedure "procedure p; if b then begin π; p end else". If b:nat, π:Program, we describe it by

While(b, π) := Recurs([π1:Program]Bin_select(b,Concat(π, π1), Dummy)) : Program.

We did not yet discuss integers and assignments like "a:=b+c". We can define the integers as sequences of bits 0 and 1 and write programs for addition, multiplication etc. It is a long way to go, but whatever we produce is available for ever.

4. SEMANTICS.

Semantics as we describe it is closely related to the methodology of denotational semantics with one of its central ideas the presentation of meaning of a program as a function from states to states (cf [5]). We take a different point of view: we do not consider functions from states to states but binary relations over the state space. This is called relational semantics (cf [6]).

When discussing semantics of a program in a particular situation it is, fortunately, often sufficient to deal with incomplete information. Some parts of the program may have semantic properties which are partly irrelevant for the properties of the program as a whole. Such incomplete information has the form of a binary relation, and can be trated in our system.

As an extra advantage we mention that we do not have the slightest trouble with non-deterministic programs.

We connect relations to programs by stating that a relation ρ presents information about a program π. In our AUTOMATH book we take this notion to be primitive, but we can give the following interpretation from an executional point of view: For every pair σ1:State, σ2:State where σ1 and σ2 are initial and final state of some execution of π, the relation ρ holds. Because of the possible incompleteness of the information, the converse (i.e. when ρ holds for σ1 and σ2, π can transform σ1 into σ2) need not be true. In the jargon of AUTOMATH, a relation is a function that adds to every σ1:State and σ2:State a proposition. So the type of all

relations is

Reln := [σ1:State][σ2:State] prop

So given ρ:Reln, σ1:State, σ2:State, "ρ holds for σ1 and σ2" is expressed by $\ll\sigma 2\gg\ll\sigma 1\gg\rho$. Further we write, given π:Program, ρ:Reln, the primitive notion

info(π,ρ) : prop

The interpretation of info(π,ρ) is the proposition "ρ presents information about π".

The basic properties embodied in this interpretation are given by the following axioms (where π:Program, ρ1:Reln, ρ2:Reln)

. info(π,ρ1 'and' ρ2) 'eqv' (info(π,ρ1) 'and' info(π,ρ2))

. (ρ1 'imp' ρ2) 'imp' (info(π,ρ1) 'imp' info(π,ρ2))

(We use 'and', 'imp', 'eqv' for the connectives ∧, =>, ≡ of ordinary propositional calculus).

The relations ρ we claim by axiom to present information about the seven primitive programs in section 3. all have a standard form, viz.

[σ1:State][σ2:State] if Some_ref_on(σ1) then σ1=σ2 else P(σ1,σ2)

where P(σ1,σ2) is a proposition, and

Some_ref_on(σ) := ∃r:nat(value(σ,ref,r)=ON).

The motivation for this is the following: Once a refuser is in ON position (because of things like nontermination, abortion), we do not want to "execute" the rest of the program anymore; in other words: this rest is equivalent to a skip, for which we present the information σ1=σ2.

So to each primitive program π we have a relation ρ in standard form and an axiom stating info(π,ρ). In this paper we do not give the relations in standard form , but only the essential part, i.e. the proposition P(σ1,σ2) in the else part.

1. To Const_ass(dt,u,i,v) is connected the proposition (playing the role of P(σ1,σ2)):

σ2 = adapt(σ1,dt,i,v)

2. To Var_ass(dt,u,i1,i2) is connected

$$\sigma 2 = \text{adapt}(\sigma 1, dt, i1, \text{value}(\sigma 1, dt, i2))$$

3. Given $\rho 1$:Reln, $\rho 2$:Reln, info($\pi 1, \rho 1$), info($\pi 2, \rho 2$), to Bin_select(b, $\pi 1, \pi 2$) is connected

$$\underline{\text{if}}\ \text{value}(\sigma 1, \text{bool}, b) = T\ \underline{\text{then}}\ \langle \sigma 2 \rangle \prec \sigma 1 \succ \rho 1\ \underline{\text{else}}\ \langle \sigma 2 \rangle \prec \sigma 1 \succ \rho 2$$

4. Given $\rho 1$:Reln, $\rho 2$:Reln, info($\pi 1, \rho 1$), info($\pi 2, \rho 2$), to Concat($\pi 1, \pi 2$) is connected

$$\exists\, \sigma\text{:State}\ (\langle \sigma \rangle \prec \sigma 1 \succ \rho 1\ \text{'and'}\ \langle \sigma 2 \rangle \prec \sigma \succ \rho 2)$$

5. Given ρ:Reln, info(π, ρ), to Block(dt,u,π) is connected

$$\exists v1\text{:elts}(dt)\ \exists v2\text{:elts}(dt)\ (\langle \text{extend}(\sigma 2, dt, v2) \succ \prec \text{extend}(\sigma 1, dt, v1) \succ \rho).$$

Since $\sigma 1$ and $\sigma 2$ are states belonging to the state space outside the block and ρ is a relation between states inside the block, we have to extend $\sigma 1$ and $\sigma 2$ with appropriate values when connecting them with ρ. They are extended with v1 and v2, representing the initial and final value of the variable local in the block.

6. Given ρ:Reln, info(π, ρ), to Injection(dt,u,π) is connected

$$\prec \text{restrict}(\sigma 2, dt) \succ \prec \text{restrict}(\sigma 1, dt) \succ \rho\ \text{'and'}\ \text{value}(\sigma 2, dt, 0) = \text{value}(\sigma 1, dt, 0)$$

Now ρ acts on a "smaller" state space than the one $\sigma 1$ and $\sigma 2$ belong to, so we have to restrict $\sigma 1$ and $\sigma 2$. The second part of 'and' states that the value of the added variable does not change.

7. In order to describe information on the recursive program Recurs(ϕ), we have to consider a sequence of relations with special properties. Given

$$\text{Seq} : \text{nat} \rightarrow \text{Reln}, \quad \text{with}$$

$$\forall \sigma 1\text{:State}\ \forall \sigma 2\text{:State}\ (\langle \sigma 2 \rangle \prec \sigma 1 \succ \langle 0 \rangle \text{Seq}\ \text{'eqv'}\ \underline{\text{if}}\ \text{Some\_ref\_on}(\sigma 1)\ \underline{\text{then}}\ \sigma 2 = \sigma 1\ \underline{\text{else}}\ \text{value}(\sigma 2, \text{ref}, \text{nonterm}) = \text{ON}))$$

$$\forall k\text{:nat}\ \forall \pi\text{:Program}\ (\text{info}(\pi, \langle k \rangle \text{Seq})\ \text{'imp'}\ \text{info}(\langle \pi \rangle \phi, \langle k+1 \rangle \text{Seq}))$$

to Recurs(ϕ) is connected

$\forall$n:nat $\exists$k:nat (k 'gtr' n 'and' <σ2><σ1><k>Seq)

The interpretation is as follows: We start from a program π0 to which we connect the proposition value(σ2,ref,nonterm)=ON. (π0 can be considered as a non-terminating program). We now build the programs $<\pi 0>\phi^0 := \pi 0$, $<\pi 0>\phi^1 := <\pi 0>\phi$, $<\pi 0>\phi^2 := <<\pi 0>\phi>\phi, \ldots$. For every k, <k>Seq is a relation that presents information on $<\pi 0>\phi^k$, by induction: <0>Seq presents information about π0, and for any k and π holds info(π,<k>Seq) 'imp' info(<π>φ,<k+1>Seq). The information presented on Recurs(φ) is now the least upperbound of the sequence Seq :

[σ1:State][σ2:State]$\forall$n:nat $\exists$k:nat (k 'gtr' n 'and' <σ2><σ1><k>Seq)

Starting from our semantics of the seven primitive programs, we can defibe relations for higher-level constructs and prove that these relations present information . Especially the while statement deserves some attention, and the programs that effect the arithmetic operations, such as "a:=b+c". Once all such standard programs have been written in our book, we gradually can start to write more complex programs and to present information about them. This set-up is completely parallel to the situation in mathematics where we start from very simple primitives, and gradually learn to say everything we want.

Much of the work we have to do when writing programs and proving semantics about them, is more or less standard. All the time we deal with complex expressions in terms of the operations on states (as given in section 2.). Those can be simplified by application of the rules we have mentioned at the end of 2., applying elementary logic and elimination of if-then-else constructs. At this moment we feel the need for a (limited) automatic simplifier. Given a complex expression in terms of extend, adapt, restrict etc. such a simplifier is supposed to deliver a simpler equivalent form of this expression (and written in AUTOMATH a proof of this equivalence). Occasionally, some human interaction might be helpful.

REFERENCES

1. N.G. de Bruijn. The mathematical language AUTOMATH, its usage and some of its extensions. Symposium on Automatic Demonstration (Versailles, Dec,1978), Lecture Notes in Mathematics, Vol. 125, pp.29-61. Springer Verlag 1970.

2. Proceedings Symposium APLASM (Dec. 1973) ed. P. Braffort. Fasciculel, The AUTOMATH mathematics checking project. Contributions by L.S. van Benthem Jutting, N.G. de Bruijn, D.T. van Daalen, I. Zandleven.

3. N.G. de Bruijn. A system for handling syntax and semantics of computer programs in terms of the mathematical language AUTOMATH. Report, Department of Mathematics, Technologilal University, Eindhoven 1973

4. N.G. de Bruijn. The use of the language AUTOMATH for syntax and semantics of programming languages. Report, Department of Mathematics, Technological University, Eindhoven. 1976.

5. Scott & Strachey, Towards a formal semantics for computer languages, in Proceedings Symposium on Computers and Automates, Polytechnic Inst of Brooklin, (1971)

6. P. Hitchcock & D. Park, Induction rules and termination proofs, in Automata, Languages and Programming, (ed M. Nivat) North Holland, 1973, p.225-252