stichting
mathematisch
centrum

MC

AFDELING ZUIVERE WISKUNDE
(DEPARTMENT OF PURE MATHEMATICS)

ZN 94/79 NOVEMBER

T.M.V. JANSSEN & P. VAN EMDE BOAS

ON INTENSIONALITY IN PROGRAMMING LANGUAGES

2e boerhaavestraat 49 amsterdam

Printed at the Mathematical Centre, 49, 2e Boerhaavestraat, Amsterdam.

The Mathematical Centre, founded the 11-th of February 1946, is a non-profit institution aiming at the promotion of pure mathematics and its applications. It is sponsored by the Netherlands Government through the Netherlands Organization for the Advancement of Pure Research (Z.W.O).

1980 Mathematics subject classification: 68B10, 03B45, 68F20, 03A05

On Intensionality in Programming Languages

by

T.M.V. Janssen & P. van Emde Boas*)

ABSTRACT

This paper presents an application of Montague grammar to the theory of semantics of programming languages. The first sections present (for readers not familiar with computer science) a description of the assignment statement. It is demonstrated that the assignment statement creates an intensional context which exhibits the same phenomena as intensional contexts in natural language. Next we discuss the need for a formal semantics for programming languages and the problems that arise concerning the semantics of the assignment statement. It is demonstrated that Montague's approach to the semantics of natural languages can be used to solve these problems concerning the semantics of programming languages.

KEY WORDS & PHRASES: *Montague grammar, assignment statement, pointers, predicate transformers, opaqueness, transparency, Algol 68, possible worlds*

---

*) Inst. Appl. Math./VPW., University of Amsterdam

1. INTRODUCTION

Programs are pieces of text, written in some programming language. These languages were designed for the special purpose of instructing computers. They also are used in communication among human beings for telling them how to instruct computers or for communicating algorithms which are not intended for computer execution. So programs are used for certain kinds of communication, hence they have some meaning. The branch of computer science called 'semantics of programming languages' deals with the relation between programs and their meanings.

There exists nowadays several thousands of mutually partly incompatible programming languages. They are formal languages with a complete formal definition of the syntax of the language. Such a definition specifies exactly when a string of symbols from the alphabet of the language is a program and when not. The definition of a programming language also specifies how a program should be executed on a computer, or, formulated more generally, what the program is intended to do. In fact, however, several programming languages are not adequately documented in this respect. Each programming language has its own set of strange ideosyncrasies, design errors, perfectly nice ideas and clumsy conventions. A few kinds of instructions are present in most of the languages. The present paper deals with the semantics of one of those instructions: the assignment statement which assigns a value to a name.

It will appear that assignment statements exhibit the same phenomena as intensional operators in natural languages. A certain position in the context of an assignement statement is transparent (certain substitutions for names are allowed), whereas another position is opaque (such substitutions are not allowed). The traditional ways of treating the semantics of programming languages do not provide tools for dealing with intensional phenomena. A correct treatment of simple cases of the assignment statement can be given, but for the more complex cases the traditional approaches fail. We will demonstrate that the treatment of intensional operators in natural language, as in Montague grammar, may also be applied to programming languages, and that in this way a formalized semantics of assignment statements can be given which deals correctly with the more complex cases as well.

2. THE ASSIGNMENT STATEMENT

One may think of a computer as a large collection of *cells* each containing a *value* (usually a number). For some of these cells names are available in the programming language. Such names are called *identifiers* or, equivalently, *variables*. The term 'identifier' is mainly used in contexts dealing with syntax, 'variable' in contexts dealing with semantics. The connection of a variable with a cell is fixed at the start of the execution of a progam and remains further unchanged. So in this respect a variable does not vary. However, the cell associated with a variable stores a value, and this value may be changed several times during the execution of a program. So in this indirect way a variable can vary. The *assignment statement* is an instruction to change the value stored in a cell.

An example of an assignment statement is: $x := 7$, read as 'x becomes 7'. Execution of this assignment has the effect that the value 7 is placed in the cell associated with x. Let us assume that initially the cells associated with x, y and w contain the values 1, 2 and 4 respectively (figure 1a). The execution of $x := 7$ results in the situation shown in figure 1b. Execution of $y := x$ has the effect that the value stored in the cell associated with x is copied in the cell associated with y (figure 1c). The assignment $w := w + 1$ applied in turn to this situation, has the effect that the value associated with w is increased by one (figure 1d).

Fig. 1a

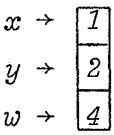


Fig. 1b

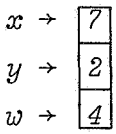


Fig. 1c

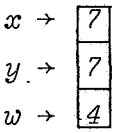
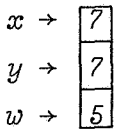


Fig. 1d



Now the necessary preparations are made for demonstrating the relation with natural language phenomena. Suppose that we are in a situation where the identifiers x and y are both associated with value 7. Consider now the assignment

(1) $\qquad x := y + 1$

The effect of (1) is that the value associated with x becomes 8. Now replace identifier y in (1) by x:

(2) $x := x + 1$

Again, the effect is that the value associated with x becomes 8. So an identifier on the right hand side of ':=' may be replaced by another which is associated with an equal value, without changing the effect of the assignment. One may even replace the identifier by (a notation for) its value:

(3) $x := 7 + 1$

Replacing an identifier on the left hand side of ':=' has more drastic consequences. Replacing x by y in (1) yields:

(4) $y := y + 1$

The value of y gets increased by one, whereas the value associated with x remains unchanged. Assignment (1), on the other hand, had the effect of increasing the value of x by one; likewise both (2) and (3). So on the left hand side the replacement of one identifier by another having the same value is not allowed. While (2) and (3) are in a certain sense equivalent with (1), assignment (4) certainly is not. Identifiers (variables) behave differently on each side of ':='.

It is striking the see the analogy with natural language. We mention an example due to QUINE 1960. Suppose that, perhaps as result of a recent appointment, it holds that

(5) *the dean = the chairman of the hospital board*

Consider now the following sentence:

(6) *The commissioner is looking for the chairman of the hospital board.*

The meaning of (6) would not be essentially changed if we replaced *the commissioner* by another identification of the same person. But consider now

(7) *The commissoner is looking for the dean.*

Changing (6) into (7) does make a difference: it thinkable that the commissioner affirms (6) and simultaneously denies (7) because of the fact that he has not been informed that (5) recently has become a truth. In the terminology for substitution phenomena, the subject position of *is looking for* is called *(referentially) transparent*, and its object position *(referentially) opaque*. Because of the close analogy, we use the same terminology for programming languages, and call the right hand side of the assignment 'transparent', and its left hand side 'opaque'.

Up until now, we only considered cells which contain an integer as value. Some programming languages also allows for handling cells containing a variable (identifier) as value (e.g. the languages Pascal and Algol 68). Names of such cells are called *pointer identifiers* or equivalently *pointer variables*, shortly *pointers*. The situation that pointer p has the identifier x as its value, is presented by figure 2a. In this situation, p is indirectly related with the value of x, so with 7. The assignment $p := w$ has the effect that the value stored in p's cell becomes w (figure 2b). Thus p becomes indirectly related with the value of w: the integer 5. When next the assignment $w := 6$ is executed, the integer value associated with p becomes 6 (figure 2c). So an assignment can have consequences for pointers which are not mentioned in the assignment statement itself. The value of the variable associated with the pointer may change.

Fig. 2a

$p \rightarrow$	x
$x \rightarrow$	7
$y \rightarrow$	7
$w \rightarrow$	5

Fig. 2b

$p \rightarrow$	w
$x \rightarrow$	7
$y \rightarrow$	7
$w \rightarrow$	5

Fig. 2c

$p \rightarrow$	w
$x \rightarrow$	7
$y \rightarrow$	7
$w \rightarrow$	6

In a real computer, a cell does not contain an integer or a variable, but rather a code for an integer or an code for a variable. For most real computers it is not possible to derive from the content of a cell, whether

it should be interpreted as an integer code or a variable code. In order to prevent the unintended use of an integer code for a variable code, or vice versa, some programming languages require for each identifier a specification of the kind of values to be stored in the corresponding cells (e.g. Pascal). The syntax of such a programming language then prevents unintended use of an integer code for an identifier code (etc.) by allowing only for programs in which each identifier is used for a single kind of value. Other languages leave it to the discretion of the programmer to use an identifier for only one kind of value (e.g. Fortran). Our examples are from a language of the former type: Algol 68. This programming language also allows for higher order pointers such as pointers to pointers to variables for integer values, and for assignments with a pointer occurring on the right hand side of the ':=' symbol. We will, however, not consider such constructs.

The observation concerning substitutions in assignments statements, as considered above, is not original. It is, for instance, decsribed in TENNENT 1976 and STOY 1977 (where the term 'transparent' is used) and in PRATT 1976 (who uses both 'transparent' and 'opaque'). The semantic treatments of these phenomena which have been proposed, are, however, far from ideal, and in fact not suitable for assignments involving pointers. The authors mentioned above, like many others, evade these difficulties by considering a language without pointers.

3. WHY STUDY THE SEMANTICS OF PROGRAMS?

We discuss as an example, a program which computes solutions of the quadratic equation $ax^2 + bx + c = 0$. The program is based upon the well-known formula

$$x_{1,2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

The program reads as follows:

```
begin real a, b, c, disc, d, x1, x2;
read ((a,b,c));
disc := b*b - 4*a*c;
   d := sqrt (disc);
  x1 := -b+d; x1 := x1/(2*a);
  x2 := -b-d; x2 := x2/(2*a);
print ((a,b,c,x1,x2,newline))
end
```

The first line of the program says that the identifiers mentioned there are only used as names of locations containing real numbers as values (e.g. *3.14159*). The second and seventh line illustrate that the computer may obtain data from outside (input) and communicate results to the outside world (output). The program also shows that the mathematical formula looks much compacter than the program, but that this compactness is made possible by the use of some conventions which have to be made explicit for the computer. For example, in the program we must write $4*a*c$ for 4 times a times c, while in the formula $4ac$ suffices. In the formula we use two dimensional features, which are eliminated in the program (*sqrt*(..) instead of $\sqrt{...}$). This linear character is neccessitated by the fact that programs have to be communicated by way of a sequential channel; for example, the wire connecting the computer with a card reader. The symbol *real* indicates that the identifiers mentioned may only be associated with real values, and the symbols *begin* and *end* indicate the begin and the end of the program.

There exists a considerable confusion among programmers, theoreticians, and designers as to what we should understand by the semantics of a programming language. For some of the relevant properties there is a measure of agreement on the need for a treatment within the field of semantics. These properties are:

correctness: A program should perform the task it is intended to perform. For example the program given above is incorrect: it does not account for $a = 0$ or $disc < 0$.

equivalence: Two different programs may yield the same results in all circumstances. For example, in the program under discussion we may interchange

the order of the computation of *x1* and *x2*, but we cannot compute *d* before we compute *disc*.

termination: If we start the execution of a program, will it ever stop? It might be the case that the computer keeps on trying to find the square root of -1, and thus for certain values of *a*, *b* and *c* never halts.

Each of the above properties tells us something about the possible computations the program will perform when provided with input data. We want to predict what may happen in case ...; more specifically, we want to prove that our predictions about the capabilities of the program are correct. How can we achieve this goal? Clearly it is impossible to try out all possible computations of the program. Instead one is tempted to run the program on a 'representative' set of input data. This activity is known as program debugging. This way one may discover errors, but one can never *prove* the program to be correct. Still, in practice, most programs used nowadays have been verified only in this way. One might alternatively try to understand the program simply by reading its text. Again this is not of great help, since mistakes made by the programmer can be remade by the reader. The only way out is the invention of an mathematical theory for proving correctness, equivalence, termination etc.. We need a formalized semantics on which such a theory can be based.

4. SOME APPROACHES TO SEMANTICS

What does a formal semantics for a program look like? The most common approach is a so-called *operational semantics*. One defines the meaning of a program by first describing some abstract machine (a mathematical model of an idealized computer) and next specifying how the program is to be executed on the abstract machine. Needless to say the problem is transferred in this way from the real world to some idealistic world. The possibly infinitely many computations of the program remain as complex as before. On the other hand, it is by use of an operational semantics that the meaning of most of the existing programming languages is specified. Examples are the programming languages PL/1 in LUCAS & WALK 1971, and, underneath its special description method, Algol 68 in VAN WIJNGAARDEN 1975.

For about 15 years so-called *denotational semantics* have been provided for programming languages (see e.g. TENNENT 1976, STOY 1977). The meaning of a program is represented by a mathematical object in a model; mostly a function which describes the input-output behaviour of the program. So we abstract from the intermediate stages of the computation, and the model has far less resemblance to a real computer than the abstract machines used in operational semantics. The programs are not considered as so much transforming values into values, but rather as transforming the entire initial state of a computer into some final state. In this approach, states are highly complex descriptions of all information present in the computer.

Mostly, we are not interested in all aspects of a computer state, but only in a small part (for instance the values of the input and output variables). This leads to a third approach to semantics, which uses so-called *predicate transformers* (FLOYD 1967, HOARE 1969, DIJKSTRA 1974, 1976). A (state) predicate is a proposition about states. So a predicate specifies a set of states: all states for which the proposition holds true. We need to correlate propositions about the state existing before the execution of the program with propositions about the state afterwards. We will from now on follow this approach to semantics.

As an example we consider the program from Section 3. An initial state may be described by specifying that on the input channel three numbers a, b and c are present such that $a \neq 0$, and $b^2 - 4ac \geq 0$. The execution of the program will lead such a state to a state where $x1$ and $x2$ contain the solutions to the equation $ax^2 + bx + c = 0$. Conversely, we observe that, if one wants the program to stop in a state where $x1$ and $x2$ represent the solutions of the equation $ax^2 + bx + c = 0$, it suffices to require that the coefficients a, b and c are present on the input channel (in this order!) before the execution of the program, and that moreover $a \neq 0$ and $b^2 - 4ac \geq 0$. In the semantics we will restrict our attention to the real computation, and therefore consider a reduced version of the program from which the input and output instructions and the specifications of the identifiers such as *real* are removed. Let us call this reduced program 'prog'. In presenting the relation between predicates and programs, we follow a notational convention attributed to HOARE 1969. Let π be a program, and ϕ and ψ predicates. Then $\{\phi\}\pi\{\psi\}$ means

that if we execute π starting in a state where ϕ holds true, and the execution of the program terminates, then predicate ψ holds in the resulting state. Our observations concerning the program are now expressed by:

$$\{a \neq 0 \wedge (b^2 - 4ac) \geq 0\} \text{ prog } \{a(x1)^2 + b(x1) + c = 0 \wedge$$

$$a(x2)^2 + b(x2) + c = 0 \wedge \forall z[az^2 + bz + c = 0 \rightarrow z = x1 \vee z = x2]\}.$$

The aim of predicate transformer semantics can now be described as follows. For any program π, find, based upon the structure of π, a predicate transformer which for any state predicate ϕ yields a state predicate ψ, such that if ϕ holds before the execution of π, then ψ gives all information about the final state which can be concluded from ϕ and π. Such a predicate ψ is called the *strongest postcondition* with respect to ϕ and π and denoted is $sp(\pi,\phi)$. Mathematically $sp(\pi,\phi)$ is characterized by

(I) $\{\phi\}\pi\{sp(\pi,\phi)\}$ and

(II) If $\{\phi\}\pi\{\eta\}$ then from $sp(\pi,\phi)$ we can conclude η.

Instead of this approach, one frequently follows an approach which goes the other way round. For a program π and a predicate α one wants to find the weakest precondition with respect to π and α: $wp(\pi,\alpha)$. It is the most general predicate which still ensures that, after execution of π, predicate α holds (see DIJKSTRA 1974, 1976 for more on this approach). In this paper we only discuss the strongest postcondition approach; a related publication concerning weakest preconditions is JANSSEN & VAN EMDE BOAS 1977b.

Above, we used the phrase "based upon the structure of π". We required this, since it would be useless to have a semantics which attaches to each program and predicate a strongest postcondition in an ad-hoc way, in particular because there are infinitely many programs. One has to use the fact that programs are formed in a structured way according to the syntax of the programming language. In this way simply Frege's principle of semantic compositionality is employed: the meaning of a compound expression is built up from the manings of its constitude parts.

5. FLOYD'S PREDICATE TRANSFORMER FOR THE ASSIGNMENT

Below, Floyd's description is given of the strongest postcondition for assignment statements of the form $v := \delta$, where v is an indentifier and δ some expression. But before doing so, we give some heuristics.

Suppose that $x = 0$ holds before the execution of $x := 1$. Then afterwards $x = 1$ should hold instead of $x = 0$. As a first guess at a generalization one might suppose that always after execution of $v := \delta$ it holds that $v = \delta$. But this is not generally correct, as can be seen from inspection of the assignment $x := x + 1$. One must not confuse the old value of an variable with the new one. To capture this old-value versus new-value distinction, the information about the old value is remembered using a variable (in the logical sense!) bound by some existential quantifier and using the operation of substitution. So after $v := \delta$ one should have that v equals 'δ with the old value of v substituted (where necessary) for v in δ'. This expression is described by the expression $v = [z/v]\delta$, where z stands for the old value of v and $[z/v]$ is the substitution operator. Thus we have obtained information about the final situation from the assignment statement itself. Furthermore we can obtain information from the information we have about the situation before the execution of the assignment. Suppose that ϕ holds true before the execution of the assignment. From the discussion on Section 2 we know that the execution of $v := \delta$ changes only the value of v. All information in ϕ not involving v remains true. So after the execution of the assignment $[z/v]\phi$ holds true. If we combine these two sources of information into one formula, we obtain Floyd's predicate transformation rule for the assignement statement (FLOYD 1967).

$$\{\phi\}v := \delta\{\exists z[[z/v]\phi \wedge v = [z/v]\delta]\}$$

Here ϕ denotes an assertion on the state of the computer, i.e., the values of the relevant variables in the program before execution of the assignment, and the more complex assertion $\exists z[[z/v]\phi \wedge v = [z/v]\delta]$ describes the situation afterwards.

The examples below illustrate how the assignment rule works in practice.

1. Assignment $x := 1$. Assertion $\phi \equiv x = 0$. Resulting assertion: $\exists z[[z/x](x = 0) \wedge x = [z/x]1]$. This reduces to $\exists z[z = 0 \wedge x = 1]$, from which one obtains $x = 1$.

2. Assignment $x := x + 1$. Assertion $\phi \equiv x > 0$. Resulting assertion: $\exists z[[z/x](x > 0) \wedge x = [z/x](x + 1)]$, reducing to $\exists z[z > 0 \wedge x = z + 1]$ from which one concludes $x > 1$.

During the last five years it has been noticed by several authors, that Floyd's assignment rule leads to incorrect results when applied to cases where the identifiers involved are not directly associated with a cell storing an integer value. The most well known example is the case of array identifiers (APT & DE BAKKER 1976, GRIES 1977). We ourselves have pointed out that such problems also arise in the case of pointers (JANSSEN & VAN EMDE BOAS 1977a). An example is the following program consisting of three consecutive assignment statements. The identifier p is a pointer variable and w an integer variable. The program is reduced in the sense of Section 4.

$$w := 5;\ p := w;\ w := 6.$$

Suppose that we have no information about the state before the execution of this program. This can be expressed by saying that the predicate $1 = 1$ holds in the initial state. By application of Floyd's rule, we find that after the first assignment $w = 5$ holds (analogously to the first example above). Note that the state presented in figure 2a (Section 2) satisfies this predicate. For the state after the second assignment Floyd's rule yields:

$$\exists z[[z/p](w = 5) \wedge p = [z/p]w]$$

which reduces to

$$\exists z[w = 5 \wedge p = w]$$

and further to

$$w = 5 \wedge p = w$$

It is indeed the case that after the second assignment the integer value related with p equals 5 (of Figure 2b). According to Floyd's rule, after the third assignment the following is true:

$$\exists z[[z/w](w = 5 \wedge p = w) \wedge w = [z/w]6]$$

reducing to

$$\exists z[z = 5 \wedge p = z \wedge w = 6].$$

This formula says that the integer value related with p equals 5. But as the reader may remember from the discussion in Section 2, the integer value related with p is changed as well (Figure 2c). So the straightforward application of Floyd's rule to this program involving pointers, yields an incorrect description of the final state.

The above example demonstrates the need for changing Floyd's rule. The main source of the problem is that the rule makes no clear distinction between a name and the object it refers to. Such an approach to semantics was considered in the field of philosophy of natural language in the beginnings of this century ('Fido' - Fido theories), but the approach was abondoned because it turned out to be too simple for treating interesting problems. In view of the analogy, it is not so surprising that Floyd's rule is not completely successful either.

6. A MONTAGUE SEMANTICS FOR THE ASSIGNMENT

As we observed in Section 2, the assignment statement creates an intensional context. Therefore it is attractive to try to apply in the field of programming languages the tools developed for the treatment of intensional phenomena in natural languages. The basic step for such an application is the transfer of the notion 'a possible world' to the context of programming languages. It turns out that we can take for this the set of possible states of a computer. With respect to this rather concrete interpretation, no ontological problems arise, we presume. The set of states will be introduced in the same way as the set of possible worlds in MONTAGUE 1973, henceforth cited

as PTQ. The set of states is just a nonempty set. The elements of this set are called states; they are not further analysed. The identification of possible worlds with computer states is nót new; authors who apply other kinds of modal logic for programming language semantics do so as well (e.g. PRATT 1976). We will work in the framework of PTQ and use intensional logic as it is defined there. We assume that the reader is familiar with the definitions and style of presentation of PTQ. The fragment of the programming language Algol 68 considered here, is a part of the fragment presented in JANSSEN & VAN EMDE BOAS 1977a.

In PTQ, categories are indices of sets. We will not be so precise and identify index and set, writing $\alpha \in C$ instead of $\alpha \in P_C$ (C a category). We have the following 5 categories:

1) NUM - the set of numbers. There are infinitely many basic expressions in this category: *1*, *2* ... *12*, ... *666* ... .

2) IID - the set of integer identifiers, with basic expressions x, y, w.

3) PID - the set of pointer identifiers, basic expressions are p and q.

4) ASS - the set of assignment statements.

5) PROG - the set of programs.

The syntactic rules are:

S1: If $\alpha \in$ NUM and $\beta \in$ NUM then F1(α,β) $\in$ NUM, where F1(α,β) $\equiv \alpha + \beta$ (e.g. *12* + *666* $\in$ NUM).

S2: If $\alpha \in$ IID then F2(α) $\in$ NUM, where F2(α) $\equiv \alpha$, so integer identifier x can be used as a number.

S3: If $\alpha \in$ IID and $\beta \in$ NUM then F3(α,β) $\in$ ASS, where F3(α,β) $\equiv \alpha$:= β (e.g. x := *7* $\in$ ASS).

S4: If $\gamma \in$ PID and $\beta \in$ IID then F4(γ,β) $\in$ ASS, where F4(γ,β) $\equiv \gamma$:= β (e.g. p := x $\in$ ASS).

S5: If $\delta \in$ ASS then F5(δ) $\in$ PROG, where F5(δ) = δ (e.g. x := *7* $\in$ PROG).

S6: If $\delta \in$ PROG and $\varepsilon \in$ PROG then F6(δ,ε) $\in$ PROG, where F6(δ,ε) $\equiv \delta;\varepsilon$ (e.g. x := *7*; p := x $\in$ PROG).

The semantics of the fragment is given by defining a translation function into intensional logic. The basic expressions *1*,*22*,... of the category NUM translate into constants $\underline{1}$,$\underline{22}$,... of type <e>. In order to deal with compound numbers such as *1 + 22*, intensional logic is extended with the binary operator $\underline{add}$, which has the usual semantics of addition. The integer identifiers x, y and w translate into constants $\underline{x}$, $\underline{y}$ and $\underline{w}$ of type <s,e>. So the interpretation of the translation of an integer identifier is an intension: a function from states to integers. The value associated with the identifier in any particular state S is obtained by taking the extension in S. The pointers p and q translate into the constants $\underline{p}$ and $\underline{q}$ of type <s,<s,e>>. Before discussing the translations of programs, we have to consider the model in which intensional logic is interpreted. Possible worlds are understood to represent internal situations of a computer. The execution of an assignment statement modifies a computer state in a rather specific way: the value of a single identifier is changed, while the values of all others are kept intact. Therefore not every model for intensional logic would be a reasonable candidate for the interpretation of programming languages. The model has to have enough structure to allow for such a way of changing a state. On the other hand, the model should not separate two states which agree in the value of each indentifier since on a real computer these states (should) behave alike. In JANSSEN & VAN EMDE BOAS 1977a, these requirements are formalized, and a model is constructed which satisifes them. Having introduced these constraints on the model, we are allowed to speak about *the* state obtained from a state S by assigning to $\underline{v}$ the value γ. This state is denoted by $<\underline{v} \leftarrow \gamma>S$.

The assignment statements and programs translate into predicate transformers. We recall that a state predicate is a proposition, so a predicate is of type <s,t>. Consequently, predicate transformers are of type <<s,t>, <s,t>>. In order to be able to formulate the predicate transformers associated with assignment statements, intensional logic is to be extended with new operators. In the discussion of Floyd's rule, we noticed that after the execution of $v := \delta$, the value of v equals the value of δ provided that in the course of evaluating δ we take for v its old value. We need operators which have the effect of interpreting an expression with respect to another state. Therefore we introduce a set of modal operators, called state

switchers. For each type τ, each constant c of type $\langle s,\tau\rangle$ and each logical variable z of type τ, a state switcher $\{z\ /\ ^{\vee}c\}$ is added to intensional logic. The interpretation of $\{z\ /\ ^{\vee}c\}\phi$ with respect to state s, is defined to be the same as the interpretation of ϕ with respect to state $\langle \underline{c} \leftarrow z' \rangle s$, where z' is the interpretation of z. Note that, due to the definition of interpretation of logical variables, the interpretation of z does not depend on the state with respect to which we interpret ϕ; so our definition of the interpretation of $\{z\ /\ ^{\vee}c\}\phi$ is a legitimate definition.

The state switchers have the useful property that in most circumstances they behave just as the usual substitution operator. The following identities are always true:

$$\{z\ /\ ^{\vee}c\}^{\vee}d = {}^{\vee}d \text{ if } d \text{ is a constant, provided that } d \neq c.$$

$$\{z\ /\ ^{\vee}c\}^{\vee}c = z$$

$$\{z\ /\ ^{\vee}c\}(\phi \wedge \psi) = \{z\ /\ ^{\vee}c\}\phi \wedge \{z\ /\ ^{\vee}c\}\psi$$

$$\{z_1/\ ^{\vee}c\}\{z_2\ /\ ^{\vee}c\}\phi = \{z_2\ /\ ^{\vee}c\}\phi$$

$$\{z\ /\ ^{\vee}c\}^{\wedge}\phi = {}^{\wedge}\phi$$

Notice that $\{z\ /\ ^{\vee}c\}^{\vee}P$ does not reduce (for P a variable).

After these preparations, we present the translation rules corresponding to the above syntactic rules. The reader may notice the analogy of T3 and T4 with Floyd's rule.

T1: If $\alpha \in$ NUM, $\beta \in$ NUM and α, β translate into $\underline{\alpha}$, $\underline{\beta}$ respectively, then F1(α,β) translates into $\underline{add}(\underline{\alpha},\underline{\beta})$.

T2: If $\alpha \in$ IID and α translates into $\underline{\alpha}$, then F2(α) translates into ${}^{\vee}\underline{\alpha}$.

T3: If $\alpha \in$ IID, $\beta \in$ NUM and α, β translate into $\underline{\alpha}$, $\underline{\beta}$ respectively, then F3(α,β) translates into $\lambda P[{}^{\wedge}\exists z\{z\ /\ {}^{\vee}\underline{\alpha}\}^{\vee}P \wedge {}^{\vee}\underline{\alpha} = \{z\ /\ {}^{\vee}\underline{\alpha}\}\underline{\beta}]$ where z is a variable of type $\langle e\rangle$.

T4: If $\gamma \in$ PID, $\beta \in$ IID and γ, β translate into $\underline{\gamma}$, $\underline{\beta}$ respectively, then F4(γ,β) translates into $\lambda P[^{\wedge}\exists r\{r \,/\, {}^{\vee}\underline{\gamma}\}{}^{\vee}P \wedge {}^{\vee}\underline{\gamma} = \{r \,/\, {}^{\vee}\underline{\gamma}\}\underline{\beta}]$ where r is a variable of type <s,e>.

T5: If $\delta \in$ ASS and δ translates into $\underline{\delta}$, then F5(δ) also translates into $\underline{\delta}$.

T6: If $\delta,\varepsilon \in$ PROG and δ, ε translate into $\underline{\delta}$, $\underline{\varepsilon}$ respectively, then F6($\underline{\delta}$,$\underline{\varepsilon}$) translates into $\lambda P[\underline{\varepsilon}(\underline{\delta}(P))]$.

We give two examples.

EXAMPLE 1. $y := x$.

The syntactic structure of this assignment is presented in Figure 3 by means of an analysis tree like the ones used in PTQ.

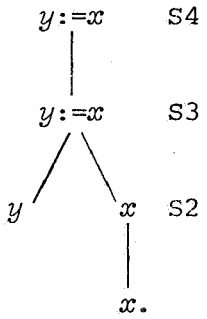


Fig. 3

So the direct unreduced translation is

$$\lambda P^{\wedge}(\exists z[\{z \,/\, {}^{\vee}\underline{y}\}{}^{\vee}P \wedge {}^{\vee}\underline{y} = \{z \,/\, {}^{\vee}\underline{y}\}({}^{\vee}\underline{x})])$$

this reduces (using one of the indentities for state switchers) to

$$\lambda P^{\wedge}(\exists z[\{z \,/\, {}^{\vee}\underline{y}\}{}^{\vee}P \wedge {}^{\vee}\underline{y} = {}^{\vee}\underline{x}])$$

Now suppose that before the execution of the assignment x equals *7* and y equals *2* (cf. Section 2, Figure 2c). So the initial state satisfies the predicate

$$\wedge({}^{\vee}\underline{x} = \underline{7} \wedge {}^{\vee}\underline{y} = \underline{2})$$

Then after the execution of the assignment the following holds:

$$\lambda P^{\wedge}(\exists z[\{z / {}^{\vee}\underline{y}\}{}^{\vee}P \wedge {}^{\vee}\underline{y} = {}^{\vee}\underline{x}]) \wedge ({}^{\vee}\underline{x} = \underline{7} \wedge {}^{\vee}\underline{y} = \underline{2})$$

reducing to

$$\wedge(\exists z[\{z / {}^{\vee}\underline{y}\}({}^{\vee}\underline{x} = \underline{7} \wedge {}^{\vee}\underline{y} = \underline{2}) \wedge {}^{\vee}\underline{y} = {}^{\vee}\underline{x}])$$

and further to

$$\wedge\exists z[{}^{\vee}\underline{x} = \underline{7} \wedge z = \underline{2} \wedge {}^{\vee}\underline{y} = {}^{\vee}\underline{x}]$$

which is equivalent with

$$\wedge({}^{\vee}\underline{x} = 7 \wedge {}^{\vee}\underline{y} = {}^{\vee}\underline{x}).$$

EXAMPLE 2. $w := 5;\ p := w;\ w := 6.$

Suppose that we have no information about the state before the execution of this program (cf. Section 5). This is expressed by saying that the predicate

$$\wedge(1 = 1)$$

holds. Then after the first assignment

$$\lambda P^{\wedge}(\exists z[\{z / {}^{\vee}\underline{w}\}{}^{\vee}P \wedge {}^{\vee}\underline{w} = \{z / {}^{\vee}\underline{w}\}\underline{5}]) \wedge (\underline{1} = \underline{1})$$

reducing to

$$\wedge({}^{\vee}\underline{w} = 5).$$

After the second assignment the following holds:

$$\lambda P(\hat{}\exists r[\{r / {}^{\vee}\underline{p}\}{}^{\vee}P \wedge {}^{\vee}\underline{p} = \{r / {}^{\vee}\underline{p}\}w])\hat{}({}^{\vee}\underline{w} = \underline{5})$$

reducing to

$$\hat{}({}^{\vee}\underline{w} = \underline{5} \wedge {}^{\vee}\underline{p} = \underline{w}).$$

After the third assignment we have:

$$\lambda P(\hat{}\exists z[\{z / {}^{\vee}\underline{w}\}{}^{\vee}P \wedge {}^{\vee}\underline{w} = \{z / {}^{\vee}\underline{w}\}\underline{6}])\hat{}({}^{\vee}\underline{w} = \underline{5} \wedge {}^{\vee}\underline{p} = \underline{w}).$$

This reduces to

(**) $$\hat{}\exists z[\{z / {}^{\vee}\underline{w}\}({}^{\vee}\underline{w} = \underline{5} \wedge {}^{\vee}\underline{p} = \underline{w}) \wedge {}^{\vee}\underline{w} = \underline{6}]$$

reducing to

$$\hat{}\exists z[z = \underline{5} \wedge {}^{\vee}\underline{p} = \underline{w} \wedge {}^{\vee}\underline{w} = \underline{6}]$$

and further to

$$\hat{}({}^{\vee}\underline{p} = \underline{w} \wedge {}^{\vee}\underline{w} = \underline{6}).$$

So the integer value related with p in the final state, is 6 (as it should be!). In (**) one observes the utility of working with intensional logic: the occurrence of w which has to be replaced by z can be discriminated from other occurrences.

7. CONCLUSIONS

The last example provides a more formal demonstration of our claim that the semantics of intensional contexts in programming languages can successfully be treated by means of the same tools as the semantics of intensional contexts in natural language: Montague grammar. This success opens perspectives for the semantical treatment of the difficult case of parameters of procedures, since the parameter position of a procedure-call also is opaque.

REFERENCES

APT, K.R. & J.W. DE BAKKER, 1976, 'Exercises in denotational semantics', in: A. Mazurkiewics (ed.), *Mathematical Foundations of Computer Science (proc. 5th Symp. Gdansk)*, Lect. Notes in Comp. Sc. vol. 45, Springer, Berlin, 1976, pp. 1-11.

DIJKSTRA, E.W., 1974, *A simple axiomatic base for programming language constructs*, Indag. Math. 36, 1-15.

DIJKSTRA, E.W., 1976, *A discipline of programming*, Prentice Hall, Englewood Cliffs (N.J.).

FLOYD, R.W., 1967, 'Assigning meanings to programs', in: J.T. Schwartz (ed.), *Mathematical aspects of computer science*, Proc. Symp. Appl. Math. 19, ACM, Providence (R.I.), 1967, pp. 19-32.

GRIES, D., 1977, *Assignment to subscripted variables*, Rep. TR 77-305, Dept. of Comp. Sc., Cornell Univ., Ithaca (N.Y.).

HOARE, C.A.R., 1969, *An axiomatic base for computer programming*, Comm. ACM 12, 576-580.

JANSSEN, T.M.V. & P. VAN EMDE BOAS, 1977a, 'On the proper treatment of referencing, dereferencing and assignment', in: A. Salomaa & M. Steinby (eds.), *Automata, Languages, and Programming (Proc. 4th Coll. Turku)*, Lect. Notes in Comp. Sc. 52, Springer, Berlin, 1977, pp. 282-300.

JANSSEN, T.M.V. & P. VAN EMDE BOAS, 1977b, 'The expressive power of intensional logic in the semantics of programming languages', in: J. Gruska (ed.), *Mathematical Foundations of Computer Science 1977 (Proc. 6th Symp. Tatranska Lomnica)*, Lect. Notes in Comp. Sc. 53, Springer, Berlin, 1977, pp. 303-311.

LUCAS, P. & K. WALK, 1971, 'On the formal description of PL/I', in: M.I. Halpern & C.J. Shaw (eds.), *Annual Review in Automatic Programming*, 6, Pergamon Press, Oxford, 1971, pp. 105-182.

MONTAGUE, R., 1973, 'The proper treatment of quantification in ordinary English', in: K.J.J. Hintikka, J.M.E. Moravcsik & P. Suppes (eds.), *Approaches to natural language*, Synthese library 49, Reidel, Dordrecht, 1973, pp. 221-242. Reprinted in: R.H. Thomason, *Formal philosphy. Selected Papers of Richard Montague*, Yale Univ. Press, New Haven, 1974, pp. 247-270.

PRATT, V.R., 1976, 'Semantical considerations on Floyd-Hoare logic', in: *Proc. 17th Symp. Found. of Comp. Sc. (Houston)*, IEEE, Long Beach (cal.), 1976, pp. 109-121.

QUINE, W.V.O., 1960, *Word and object*, The MIT Press, Cambridge (Mass.).

STOY, J.E., 1977, *Denotational semantics: the Scott-Strachey approach to programming language theory*, the MIT Press, Cambridge (Mass.).

TENNENT, R.D., 1976, *The denotational semantics of programming languages*, Comm. ACM 19, pp. 437-453.

VAN WIJNGAARDEN, A. et al., 1975, *Revised report on the algorithmic language ALGOL 68*, Acta Inform. 5, 1-236.