ON THE PROPER TREATMENT OR REFERENCING, DEREFERENCING AND ASSIGNMENT

T.M.V. Janssen
Mathematical Centre
Amsterdam, The Netherlands.

&

P. van Emde Boas
Institute for Applied Mathematics / I.P.W.
University of Amsterdam, The Netherlands.

ABSTRACT

A Floyd-like semantics is presented for the assignment statement in a fragment of ALGOL 68. The fragment considered contains array identifiers, referencing, dereferencing and conditionals. The semantics is based upon an interpretation in a model of intensional logic, without use of addresses or stores. In doing so, several ideas developed by R. Montague concerning the treatment of semantics for natural languages are applied for the first time in the area of semantics of programming languages. We also consider an operational semantics, based on the same model and prove that the Floyd-like semantics is valid with respect to the operational one and always yields the strongest postcondition.

1. INTRODUCTION

The purpose of the mathematical theory of semantics of programming languages is to describe in a computer-independent way those aspects of the processes taking place during execution of a program which are considered as mathematically relevant. In order to do so one needs a mathematical model in which such processes will be described. A fundamental question in choosing such a model is the treatment of identifiers, since this has immediate consequences for the treatment of assignments.

In the Scott-Strachey style of denotational semantics one relates identifiers with locations in an abstract store and assignments are then treated as modifications of the content of a location in the store (SCOTT & STRACHEY (1971)). The official description of ALGOL 68 (VAN WIJNGAARDEN (1976)) uses the fundamental relation " to refer to" which may hold between a name and a value. The meaning of assignments is expressed by describing the elaboration in terms of this relation. In both of these approaches, as in many others, one has to fall back on some abstract machine. We consider this to be a disadvantage, since the internal organization of a computer in stores and addresses is not a mathematically relevant aspect!

In the Floyd-Hoare approach of inductive assertions an identifier is not related with a location but only with its value. The semantics of assignments is connected to the input-output behaviour of the identifiers; one describes the relation which

exists between assertions about the values of the identifiers before and after the execution of the assignments. FLOYD (1967) gives the following rule for the assignment statement (in this [z/x] means: replace all occurences of x in ϕ by z).

(F) $\quad \phi \{x:= t\} \exists z[[z/x]\phi \wedge x = [z/x]t].$

The rules of Floyd and Hoare however yield undesirable results if applied to situations where the destination of an assignment (i.e., the expression modified by it) is not simple. For example consider the following assignments:

(1) if p then x else y fi:= t

(2) a[a[1]]:= 2

In example (1) rule (F) cannot be used since the destination is too complex to be substituted for. If we would apply (F) to program (2) odd things could happen. Suppose we take for ϕ the assertion a[1] = 1 $\wedge$ a[2] = 1. Then (F) implies that after execution of the assignment holds that:

$$\exists z[z/a[a[1]]] \; (a[1] = 1 \wedge a[2] = 1) \wedge a[a[1]] = [z/a[a[1]]]2;$$

This formula reduces to

$$a[1] = 1 \wedge a[2] = 1 \wedge a[a[1]] = 2,$$

which clearly is a contradiction. A solution for this problem is given by DE BAKKER (1976). He treats, however, only the one-dimensional case, while the same problem may arise in arrays of any dimension as is demonstrated by the example

(3) q[q[1][2]] [2] := 2.

Another source of problems is the use of higher order references. Consider the following program in which x is an integer identifier and xx a pointer:

(4) x:= 4; xx:= x; x:= 3

If one would apply (F) to these assignments one would obtain that after the second assignment x = 4 $\wedge$ xx = 4 holds , so that after the third

$$\exists y[y = 4 \wedge xx = y \wedge x = 3]$$

holds. This is incorrect: the integer value corresponding to xx (obtained by twice dereferencing) is modified by the assignment x:= 3 although this is hardly visible from the program text. In practice this is an easy source for program bugs!

The problems sketched above are related with the fact that the contribution of an expression to the meaning of an assignment may depend on its textual position: sometimes this contribution is only its value, and sometimes it is more. Consider the assignment y:= x+1, and assume that both x and y have value 7. Then the result of the assignment is insensitive for replacing "x" by "y" or "7", whereas replacing

of "y" by "x" leads to another assignment. STRACHEY (1967) has explained this situation by attributing to each identifier two values (the L-value (~ address) and the R-value (~ content)). In the formal definition of ALGOL 68 this is explained by attributing to "x" a "reference to integral" value from which an "integral value" may be obtained by the action of "dereferencing", which is allowed on the right hand side of an assignment but forbidden on the left hand side(with some exceptions).

It is striking that the same phenomenon can be observed in natural languages. We consider an example due to QUINE (1960). Suppose that by recent appointment holds that

(1) The dean = the chairman of the hospital board

Consider the following sentences:

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

(3) The commissioner if looking for the dean.

The meaning of (2) and (3) is not essentially changed if we replace "the commissioner" by another description of the same person. Such a context is called *referentially transparent*. Changing, however (2) into (3) clearly makes a difference: it is thinkable that the commissioner affirms (2) and simultaneously denies (3), because of the fact that he has not yet been informed that (1) recently has become a truth. A context like "the dean" in (3) is called *referentially opaque* (QUINE (1960)).

Problems concerning reference constitute an intriguing part of language philosophy. Many linguists, philosophers and logicians (among them D. SCOTT (1970)) worked on attempts to deal with them. The investigations culminated in the work of R. MONTAGUE. In *The proper treatment of quantification in ordinary English*, (1973), he presents the syntax and semantics of a fragment of English in which such problems are treated. References to the earlier works in this direction can be found in the introductory article of PARTEE (1975). We will refer in the sequel to Montague's article by "PTQ".

The basic idea in Montague's approach is the use of the concepts extension and intension. The extension of an expression is its value in the current world. The intension is the function yielding this value in any possible world. Consider for instance the sentence "John walks". The extension is a truth-value; in order to decide which one, we have to investigate the state of the actual world and find out whether John is actually walking or not. The intension of the sentence is the boolean function which tells us for each possible world whether John is walking or not.

The same concepts can be applied in the theory of programming languages. The extension of x is its integer value in the current computer state, its intension is the function which attaches to each state the value of x in that state. So as first orientation we have the following parallelism:

Montague:	Semantics of programming languages:
possible worlds	states within the computer
extension	R-value; dereferenced ref int value
intension	L-value; address; ref int value

We wish to point out that the concepts in the left column are in several respects more general. In denotational semantics L-values only exist for a restricted class of expressions, whereas we relate with each expression an intension (pointers, array identifiers as well as conditionals). Possible worlds are abstract sets which need not be structured with two types of values (such as addresses and integer denotations).

In this paper we will apply Montague's approach to natural languages in the area of programming languages. Consequently we have the following framework. Programs are syntactic structures produced in a generative formal system. By application of some rules on these structures we obtain the readable text of the programs. The semantics of the programs is obtained by application of translation rules to the syntactic structure; by translation each expression in a program becomes a meaningful expression in Intensional Logic (IL). IL is the kind of modal logic used by Montague for expressing meaning. Programs and statements are translated in forward predicate transformers. The translation rules are recursive operators: the translation of a compound expression is some combination of the translations of the subparts. This means that the translation rules are in a 1-1 correspondence with the syntactic rules.

This paper is organized as follows . In section 2 we describe the programming language fragment treated. Section 3 and 4 provide the syntax and semantics of the extension of IL we use. In section 5 we present the rules for the translation of the programming language in IL. Section 6 contains some illustrative examples. In section 7 we provide a more operationally defined semantics and prove the two semantics are nicely related: the predicate transformers compute for each predicate the strongest postcondition. Section 8 makes some comparisions with PTQ.

ACKNOWLEDGEMENTS

We wish to thank A.E. BROUWER, W.P. DE ROEVER, L.G.L.T. MEERTENS and an anonymous referee for their comments on an earlier version of this paper.

2. THE ALGOL 68 FRAGMENT

The semantical treatment of the assignments mentioned in section 1, will be given by presenting the semantics of a programming language containing these assignments. This language constitutes a fragment of ALGOL 68; the fragment contains no loops, jumps or procedures. The program text is understood to be obtained from a derivation tree. If it is relevant in the context, we will indicate the rule used to generate an expression by writing indexed brackets around the text, i.e., $[P]_X$ denotes that program text P is obtained by application of an instance of rule X.

The fragment is described by means of a van Wijngaarden grammar; the same tool has been used in the official definition (VAN WIJNGAARDEN (1976)). The grammar has one metanotion. The production rules of the metanotion are

MODE → int | ref MODE | row of MODE;

The hyper-rules (rule schemata) are listed below. One obtains a production rule of the grammar by taking a hyper-rule and substituting for all occurences of MODE in it the same string, namely a string produced by the metaproduction rules. Expressions between #-symbols are comments used to name the rules. The basic (lexical) symbols of the grammar are the underlined words, the identifiers and the symbols ; , :=, [,], (,), =, <, ≤, >, ≥, +, - and *; the symbols ǂ and ǂ are auxiliary symbols for delimiting the hypernotions. They disappear when the basic symbols are introduced by rules as ǂint idǂ → 1|2|3|... . We will not explicitely list these rules.

ǂprogramǂ → ǂsimple programǂ #P1# |
ǂprogramǂ ; ǂsimple programǂ #P2#

ǂsimple programǂ → ǂassignmentǂ #P3# |
if ǂboolexpǂ then ǂprogramǂ else ǂprogramǂ fi #P4#

ǂassignmentǂ → ǂref MODE idǂ := ǂMODE expǂ #A1# |
if ǂboolexpǂ then ǂref MODE unitǂ else ǂref MODE unitǂ fi
:= ǂMODE expǂ #A2# |
ǂref row of MODE unitǂ[ǂint expǂ]:= ǂMODE expǂ #A3#

ǂMODE expǂ → ǂMODE unitǂ #E1#| ǂref MODE expǂ #E2#

ǂMODE unitǂ → ǂMODE idǂ #E3# |
if ǂboolexpǂ then ǂMODE unitǂ else ǂMODE unitǂ fi #E4#

ǂref MODE unitǂ → ǂref row of MODE unitǂ[ǂint expǂ] #E5#

ǂboolexpǂ → (ǂboolexpǂ) and (ǂboolexpǂ) #B1# |
not (ǂboolexpǂ) #B2# | ǂboolidǂ #B3# |
ǂint expǂ * ǂint expǂ #B4-B9#
where * stands for =, <, ≤, > or ≥.

ǂint expǂ → -(ǂint expǂ) #I1# | +(ǂint expǂ) #I2# |
(ǂint expǂ) @ (ǂint expǂ) #I3-I7#
where @ stands for +, -, *, div or mod

The boolid's are true and false for the values TT (truth) and FF(falsehood). The intid's are the usual denotations for the integer: 1,2,...,. For other modes we have some privileged identifiers: ref int id: x, y; ref row of int id: a, b; ref ref int id: xx, yy; ref ref row of int id: aa, bb; ref row of row of int id: q.

The reader might be lured into constructing a grammar with less rules producing the same programs. The choice of the present rules was a consequence of the priciple that the syntactic rules have a 1-1 correspondence with the rules for semantics. One of the other aspects in which the above rules deviate from the ALGOL 68 definition is the treatment of the dereferencing of conditionals. We do this "outside", whereas officially this happens "inside" (combined with "balancing"). Moreover we don't generate expressions like aa[1]. These deviations are not essential (see section 7).

Finally: we have no identifiers for constant modes (i.e., modes not beginning with reference to; exceptions are 1,2,3,...).

The rule E2 introduces ambiguities in the sense that an expression constituting a mode expression could result from several other derivations and thus belong to several other modes. This rule, however, does not introduce ambiguous assignments since the rule cannot be freely used on the left hand side of an assignment. The syntactic rules avoid ambiguities by introducing parentheses (and); we will omit them if no confusion can arise.

Examples of generated programs are

(1) x:= y

(2) a[a[1]]:= 1

(3) x:= 4; xx:= x; x:= 3

(4) aa:= b

(5) q[q[1][2]][2]:= 2

(6) if x > 0 then x else y fi:= 0

3. SYNTAX OF INTENSIONAL LOGIC

Each expression of intensional logic will be an expression of a certain type. Therefore we first define TYPE, the set of all possible types. Let s, t and e be fixed distinct objects. Then TYPE is recursively defined by

(1) e, t $\in$ TYPE (e ~ "entity"; t ~ "truth value")

(2) if $\tau_1,\tau_2 \in$ TYPE then $\langle\tau_1,\tau_2\rangle \in$ TYPE

(3) if $\tau \in$ TYPE then $\langle s,\tau\rangle \in$ TYPE (s ~ "state").

Since some of the types in IL correspond to MODE's in our programming language fragment, we introduce the set of achievable types ATYPE by

(1) e $\in$ ATYPE

(2) $\tau \in$ ATYPE $\Longrightarrow$ $\langle s,\tau\rangle \in$ ATYPE and $\langle e,\tau\rangle \in$ ATYPE.

If we now introduce the correspondence bool ~ t, int ~ e ref int ~ <s,e>, row of int ~ <e,e>,..., it is easily seen that the set ATYPE contains all types corresponding to metaproductions of MODE.

As logical constants of type τ we use the same symbols as the identifiers of the corresponding mode, however a different type font is used. So beside the ref int id x we have a constant x in IL of type <s,e>; other constants are *2*, *true* and q. As variables of type τ we usually use z_τ. We drop the subscript τ if it can be predicted from the context. By CON_τ (VAR_τ) is understood the set of constants (variables) of type τ and $CON = \bigcup_{\tau\in TYPE} CON_\tau$ ($VAR = \bigcup_{\tau\in TYPE} VAR_\tau$). We suppose $CON_\tau \neq \emptyset$ for all $\tau \in$ ATYPE.

The set ME_τ of meaningful expressions of type τ is inductively defined as follows.

(1) $CON_\tau \subset ME_\tau$ (2) $VAR_\tau \subset ME_\tau$

(3) If $\phi,\psi \in ME_\tau$ then $\phi = \psi \in ME_t$

(4...7) If $\phi,\psi \in ME_e$ then $\phi * \psi \in ME_t$ where $*$ stands for $<$, $>$, $\leq$ and $\geq$.

(8...14) If $\phi,\psi \in ME_e$ then $+(\phi)$, $-(\phi)$, $(\phi)+(\psi)$, $(\phi)-(\psi)$, $(\phi)*(\psi)$, $(\phi)\ div\ (\psi)$, and $(\phi)\ mod\ (\psi) \in ME_e$.

(15...18) If $\phi,\psi \in ME_t$ then $(\phi)\rightarrow(\psi)$, $(\phi)\wedge(\psi)$, $(\phi)\vee(\psi)$ and $\neg(\phi) \in ME_t$.

(19,20) If $\phi \in ME_t$, $z \in VAR$ then $\forall z[\phi]$ and $\wedge z[\phi] \in ME_t$.

(21) If $\phi \in ME_{\tau_2}$ $z \in VAR_{\tau_1}$ then $\lambda z[\phi] \in ME_{<\tau_1,\tau_2>}$

(22) If $\phi \in ME_{<\tau_1,\tau_2>}$, $\psi \in ME_{\tau_1}$ then $\phi(\psi) \in M_{\tau_2}$

(23) If $\phi \in ME_{<s,\tau>}$ then $^\vee\phi \in ME_\tau$

(24) If $\phi \in ME_\tau$ then $^\wedge\phi \in ME_{<s,\tau>}$

(25) If $\phi,\psi \in ME_\tau$, $\beta \in ME_t$ then $\underline{if}\ \beta\ \underline{then}\ \phi\ \underline{else}\ \psi\ \underline{fi} \in ME_\tau$

(26) If $\phi \in ME_{\tau_1}$, $c \in CON_{<s,\tau_2>}$, $z \in VAR_{\tau_2}$ then $\{z/^\vee c\}\ \phi \in ME_{\tau_1}$

The clauses 25 and 26 are essential extensions to IL as described in PTQ. The symbols $^\wedge$ ("up") and $^\vee$ ("down") denote the intension and extension operators. The brackets [and] are sometimes used instead of (and) in order to keep in mind what we are modelling. The operator $\{z/^\vee c\}$ might be interpreted as syntactic substitution operator which replaces all occurences in ϕ of the expression $^\vee c$ by the variable z; the problem however is that not all the occurences of $^\vee c$ are yet visible in the expression on which the operator works. A semantical interpretation of the operator and a description of its syntactic behaviour is given in the next section.

The logical connectives are defined only for ϕ, $\psi \in ME_t$. It will however be convenient to have them also for the case that ϕ or ψ are intensions. If $\phi,\psi \in ME_{<s,t>}$ then $\phi \underline{\wedge} \psi$ denotes $^\vee\phi \wedge {}^\vee\psi$, if $\phi \in ME_{<s,t>}$ and $\psi \in ME_t$ then $\phi \underline{\wedge} \psi$ denotes $^\vee\phi \wedge \psi$. We use analogous conventions for $\underline{\vee}$ and $\underline{\neg}$.

It will be useful in the sequel to have constants of the achieveable types. We therefore introduce for each $\tau \in ATYPE$ sets $\underline{A}_\tau$ of achieveable value denotations (note that the elements of A_τ are syntactic objects). We define for each $\tau \in ATYPE$ the set $\underline{A}_\tau$ inductively by:

(1) $\forall\ i \in CON_e$ $\underline{i} \in \underline{A}_e$

(2) $\forall\ c \in CON_{<s,\tau>}$ $\underline{c} \in \underline{A}_{<s,\tau>}$

(3) $\forall\ \underline{i} \in \underline{A}_e$ $\forall\ \rho \in \underline{A}_{<s,<e,\tau>>}$ $\underline{\phi[i]} \in \underline{A}_{<s,\tau>}$

(4) If for all $i \in \mathbb{Z}$: $\phi_i \in \underline{A}_\tau$ then $(\phi_i)_{i \in \mathbb{Z}} \in \underline{A}_{<e,\tau>}$

So clause (4) introduces "infinite" symbols.

4. INTERPRETATION OF INTENSIONAL LOGIC

The meaningful expressions of intensional logic are interpreted in an intensional model. Such a model is triple $M = \langle \mathbb{Z}, S, F \rangle$ where S is a non-empty set and F a function which interpretes the constants. The elements of S are called states. As names for element of S we usually take s and t; the reader should not confuse them with the s and t occuring in the TYPE definition. The set $\mathbb{Z}$ is the set of integers; on $\mathbb{Z}$ are defined the operators $+, -, *$, *mod*, *div*, the relations $<, >, \leq, \geq$ and the monadic operators + and −; all with their usual meaning. The function F must be such that if c is a constant of type τ, then $F(c) \in D_\tau$ where the sets D_τ (domains of type τ) are defined as follows:

$D_e = \mathbb{Z}$, $D_t = \{TT, FF\}$, where TT and FF are the truthvalues for truth and false hood respectively

$$D_{\langle s,\tau \rangle} = (D_\tau)^S = \{f \mid f: S \to D_\tau\}.$$

$$D_{\langle \tau_1, \tau_2 \rangle} = (D_{\tau_2})^{D_{\tau_1}} = \{f \mid f: D_{\tau_1} \to D_{\tau_2}\}.$$

The function F should of course be "natural" in the sense that the integer constants *0, 1, 2,...* are interpreted as the corresponding numbers in $\mathbb{Z}$, and the constants *true* and *false* as TT and FF respectively.

A fixation g is a function which gives values to variables such that if $z \in \mathrm{VAR}_\tau$ then $g(z) \in D_\tau$. The expression $h \underset{u}{\sim} g$ indicates that h is a fixation with $h(z) = g(z)$ for all variables distinct from u. If $u \in \mathrm{VAR}_\tau$ and $d \in D_\tau$ then by $\{u \dashv d\}g$ is understood the fixation h with $h \underset{u}{\sim} g$ and $h(u) = d$.

The interpretation or valuation of a meaningful expression ϕ in model M with respect to fixation g and state s is denoted by ${}^{V}_{M,s,g}(\phi)$. This notion is defined by the following inductive definition; since the model M remains unchanged we dropped the subscript M.

(1) $${}^{V}_{s,g}(c) = F(c) \quad \text{if } c \in \mathrm{CON}.$$

(2) $${}^{V}_{s,g}(z) = g(z) \quad \text{if } z \in \mathrm{VAR}.$$

(3) $${}^{V}_{s,g}(\phi = \psi) = \begin{cases} TT & \text{if } {}^{V}_{s,g}(\phi) = {}^{V}_{s,g}(\psi) \\ FF & \text{otherwise} \end{cases}$$

(4...14) $${}^{V}_{s,g}(\phi < \psi) = \begin{cases} TT & \text{if } {}^{V}_{s,g}(\phi) < {}^{V}_{s,g}(\psi) \\ FF & \text{otherwise} \end{cases}$$

and similar for the other relational and arithmetical operators.

(15...18) $${}^{V}_{s,g}(\phi \wedge \psi) = \begin{cases} TT & \text{if } {}^{V}_{s,g}(\phi) = TT \text{ and } {}^{V}_{s,g}(\psi) = TT \\ FF & \text{otherwise} \end{cases}$$

and similar for the cases $\phi \to \psi$, $\phi \vee \psi$, $\neg\phi$.

(19,20) $$\underset{s,g}{V}(\forall z[\phi]) = \begin{cases} TT & \text{if there is a fixation } h \underset{z}{\sim} g \text{ such that } \underset{t,h}{V}(\phi) = TT \\ FF & \text{otherwise.} \end{cases}$$

and similar for the case $\wedge z[\phi]$.

(21) We define now the valuation of $\lambda z[\phi]$. Suppose $z \in VAR_{\tau_1}$ and $\phi \in ME_{\tau_2}$. Then $\underset{s,g}{V}(\lambda z[\phi])$ is that function f with domain D_{τ_1} such that whenever $d \in D_{\tau_1}$ then f(d) equals $\underset{s,\{z \to d\}g}{V}(\phi)$. We introduce $\underline{\lambda}$ as a meta-abstraction operator and rewrite the previous phrase as $\underset{s,g}{V}(\lambda z[\phi]) = \underline{\lambda} d[\underset{s,\{z \to d\}g}{V}(\phi)]$.

(22) $$\underset{s,g}{V}(\phi(\psi)) = \underset{s,g}{V}(\phi)(\underset{s,g}{V}(\psi)).$$

(23) $$\underset{s,g}{V}(\check{}\,\phi) = \underset{s,g}{V}(\phi)\ (s).$$

(24) $$\underset{s,g}{V}(\hat{}\,\phi) = \underline{\lambda} t[\underset{t,g}{V}(\phi)]$$

(25) $$\underset{s,g}{V}\ (\underline{\underline{if}}\ \beta\ \underline{\underline{then}}\ \phi\ \underline{\underline{else}}\ \psi\ \underline{\underline{fi}}) = \begin{cases} \underset{s,g}{V}(\phi) & \text{if } \underset{s,g}{V}(\beta) = TT \\ \underset{s,g}{V}(\psi) & \text{otherwise} \end{cases}$$

(26) This case is defined below since it requires more explication.

States may be understood to represent the internal situation of a computer. The execution of an assignment will modify the situation in a rather specific way: the value of a single identifier will be changed, keeping intact the values of other identifiers. So not every possible model for IL would be a reasonable candidate for the interpretation of programming languages. The model should have enough structure to allow for such a way of changing a state. On the other hand, the model should not separate two states in which all constants have "equal" values since on a real computer these states should behave equivalently.

In order to express these requirements formally, we define inductively sets A_τ of achievable values of type τ as follows:

(1) $A_e = \mathbb{Z}$

(2) $\forall\, \tau \in ATYPE \qquad \forall\, c \in CON_\tau \qquad F(c) \in A_\tau$

(3) if $\rho \in A_{<s,<e,\tau>>}$ then for each $i \in \mathbb{Z}$ and all $s \in S$: $\underset{s}{V}(\hat{}((\check{}\,\rho)(i))) \in A_{<s,\tau>}$

(4) $A_{<e,\tau>} = A_\tau^{\mathbb{Z}}$. Note that $A_\tau \neq \emptyset$ for $\tau \notin ATYPE$. By the definitions of $\underline{A}_\tau$ and A_τ it is clear that a natural bijection G can be defined. The characteristic function of A_τ will be denoted as $\underline{\underline{ach}}_\tau$; the quantification $\exists z$ will be used as an abbreviation for $\forall z_\tau[\underline{\underline{ach}}_\tau(z) \wedge \phi]$.

The above requirements are now dealt with by stating that we restrict our

attention to models for IL which satisfy the following postulates:

(i) PROPERNESS POSTULATE

For every $s \in S$ and every $c \in CON_{<s,\tau>}$ holds that $F(c)(s) \in A_\tau$

So the only possible values for constants are achievable values.

(ii) UPDATE POSTULATE

For every $s \in S$, every $c \in CON_{<s,\tau>}$ and every $a \in A_\tau$ there is a unique $t \in S$ such that

$$\begin{cases} F(c)(t) = a \\ F(c')(t) = F(c')(s) & \text{for all constants } c' \neq c. \end{cases}$$

So the value of one identifier can be changed, while all other identifiers remain unchanged; moreover, this new state is unique. We denote this state by $<c \leftarrow a>s$.

Having formulated and explained our update postulate, we give the remaining clause for the interpretation of IL:

(26) $$\overset{V}{s,g}\{z/^\vee c\}\phi = \overset{V}{<c \leftarrow g(z)>s,g}(\phi)$$

In this definition we assume that $g(z) \in A_\tau$ for some $\tau \in$ ATYPE; otherwise the result of the operator is undefined. We will only use the operator in cases that $g(z)$ is achievable. So interpreting $\{z/^\vee c\}\phi$ means that we have to shift the state and look to the resulting value of ϕ.

A model for IL satisfying the properness postulate and the update postulate is obtained as follows. We use the sets $\underline{A}_\tau$ of achieveable value denotations and define the set of states by

$$S = \prod_{\tau \in \text{ATYPE}} \prod_{CON_{<s,\tau>}} \underline{A}_\tau$$

For $c \in CON_{<s,\tau>}$ we denote the projection on the c-th coordinate set of state by Π_c

Having chosen the set S, the sets D_τ are determined for each type τ. To complete the description of the model we must explain how $F(c)$ is defined for constants. This function is defined simultaneously with a mapping $G: \bigcup_{\tau \in \text{ATYPE}} \underline{A}_\tau \to \bigcup_{\tau \in \text{ATYPE}} A_\tau$.

(1) $F(\underline{i}) = G(\underline{i}) = i$ for $\underline{i} \in A_e$

i.e. number denotations are mapped onto the integers denoted by them.

(2) $F(c) = G(\underline{c}) = \underline{\lambda} s[G(\Pi_c(s))]$ for $c \in CON_{<s,\tau>}$

(3) $G(\rho[\underline{i}]) = \underline{\lambda} s[G(\rho)(s)[G(\underline{i})]]$ for $\rho \in \underline{A}_{<s,<e,\tau>>}$

(4) $G((\phi_i)_{i \in \mathbb{Z}}) = \underline{\lambda} n[G(\phi_n)]$ for $(\phi_i)_{i \in \mathbb{Z}} \in \underline{A}_{<e,\tau>}$

Clearly the map $G: \bigcup_{\tau \in \text{ATYPE}} \underline{A}_\tau \to \bigcup_{\tau \in \text{ATYPE}} A_\tau$ in this way becomes a bijection. Moreover

the properness postulate and update postulate are satisfied by definition of S.

In the sequel we assume that we interprete the meaningful expressions in some intensional model M satisfying the properness and update postulate.

We write $M,s,g, \models \phi$ iff $\overset{V}{M,s,g}(\phi) \models TT$; we write $M,s \models \phi$ iff for all g: $M,s,g \models \phi$ and we write $M \models \phi$ if for all s: $M,s \models \phi$. Since we will not change the model we always omit the M.

Below we will mention some definitions and theorems on our extension of IL, most of the proofs can be found in JANSSEN (1976). Further information on IL and related logical subjects can be found in GALLIN (1976).

DEFINITION: Let $[\psi/z]\phi$ denote the formula obtained from ϕ by replacing each free occurence of z by ψ.

THEOREM: *If* $\models \psi \leftrightarrow \eta$ *then* $\models [\psi/z]\phi \leftrightarrow [\eta/z]\phi$.

REMARK: It is not true that $s \models \psi \leftrightarrow \eta$ implies $s \models [\psi/z]\phi \leftrightarrow [\eta/z]\phi$

CONVERSION THEOREM: *Let* $\lambda z[\phi](\alpha)$ *be a meaningful expression.*

Suppose I: No free occurrence of a variable in ϕ *becomes bound by substitution of* ψ *for* z *in* ϕ.

and II: For all states s *and* t: $\overset{V}{s,g}(\alpha) = \overset{V}{t,g}(\alpha)$

Then $\models \lambda z[\phi](\alpha) \leftrightarrow [\alpha/z]\phi$

REMARK: This theorem implies that for constant (i.e., state independent) arguments λ- conversion is allowed

THEOREM: $\models \check{}\hat{}\ \phi \leftrightarrow \phi$.

REMARK: It is not in general true that $\models \hat{}\check{} \phi \leftrightarrow \phi$. A counter example can be found in JANSSEN (1976).

SUBSTITUTION THEOREM: *The syntactic behaviour of the semantical defined substitution operator is described as follows* (z *is supposed to be achievable!*):

(1) $\{z/\check{}\chi\}[\phi \wedge \psi] = \{z/\check{}\chi\}\phi \wedge \{z/\check{}\chi\}\psi$
and also for the other connectives.

(2) $\{z/\check{}\chi\}\forall v[\phi] = \forall v\ [\{z/\check{}\chi\}\phi]$
provided that $v \neq z$, *(if* $v \equiv z$ *then we take an alphabetical variant of* $\forall v[\phi]$). *Analoguosly for* $\wedge z$, λz.

(3) $\{z/\check{}\chi\}\phi(\psi) = \{z/\check{}\chi\}\phi(\{z/\check{}\chi\}\psi)$

(4) $\{z/\check{}\chi\}\hat{}\ \phi = \hat{}\phi$

(5) $\{z/\check{}\chi\}\{w/\check{}\chi\}\phi = \{w/\check{}\chi\}\phi$

(6) $\{z/\check{}\chi\}c = c$ *for any constant* c, *including* $c \equiv \chi$

(7) $\{z/\check{}\chi\}\check{}\chi = z$; $\{z/\check{}\chi\}\check{}c = \check{}c$ *for any constant* $c \neq \chi$.

Note that in other cases $\{z/\check{}\chi\}\check{}\phi$ *does not reduce any further*

Proof of case 5):

$$\overset{V}{s,g}\{z/^{\vee}\chi\}\{\omega/^{\vee}\chi\}\phi = \overset{V}{<\chi\leftarrow g(z)>s,g}\{\omega/^{\vee}\chi\}\phi = \overset{V}{<\chi\leftarrow g(\omega)>(<\chi\leftarrow g(z)>s),g}\phi =$$

$$= \overset{V}{<\chi\leftarrow g(\omega)>s,g}\phi = \overset{V}{s,g}\{\omega/^{\vee}\chi\}\phi$$

□.

5. MONTAGUE SEMANTICS

As we have remarked in the introduction, Montague semantics consists in defining a translation which gives for each syntactic structure of the programming language some meaningful expression of IL. Since we already defined the meaning of IL (by means of its model theory) we provide by this translation the meaning of the expression from the programming language. If ω is such an expression (which may involve brackets [and $]_X$ indicating by which rule it was produced) then its translation will be denoted as ω'. Assignments and programs are translated into forward (state) predicate transformers, which map a predicate about the state before the execution of the assignment into a predicate about the state after the execution. A (state) predicate is a function from states to truth values, so a function of type <s,t>; it will have the format of an intension of an assertion: $\hat{}\phi \in ME_{<s,t>}$. Consequently predicate transformers are functions of type <<s,t>,<s,t>>; they will have the format $\lambda P[\psi]$, where $P \in VAR_{<s,t>}$ and $\psi \in ME_{<s,t>}$.

Identifiers like "x", "true", "1",... are translated into constants looking similar: x, *true*, *1*... (note the different type fond used). So x' = x. Translating mode expressions is in most cases more or less self evident, only E2 and E5 need to be mentioned. E2: $[\psi]'_{E2} = {}^{\vee}(\psi')$, so the translation of a dereferenced ref MODE expression is obtained by taking the extension of the corresponding ref MODE expression. E5: $[\rho[\nu]]'_{E5} = \hat{}(^{\vee}\rho')[\nu']$, so the translation of q[1][2], in which two instances of rule E5 are used, is $\hat{}(^{\vee}(\hat{}((^{\vee}q)[1])[2]))$; this reduces to $\hat{}(((^{\vee}q)[1])[2])$, usually written as $\hat{}(^{\vee}q)[1][2]$.

The translation rules involving programs and assignments are listed below. It must be noted that our translation rules are actually defined for a language extending our ALGOL 68 fragment. We translate also expressions of the format $\lambda n[\phi]$; the translation of such an expression is straightforward , e.g. n is translated by some $n \in VAR_e$.

P1, P3: $[\Pi]'_{P1} = \Pi'$ $\quad$ $[\Pi]'_{P3} = \Pi'$

So the translation of a program consisting of a simple program is the translation of of that simple program and the translation of a simple assignment program is obtained by translating the assignment.

P2: $[\Pi_1;\Pi_2]'_{P2} = \lambda P[\Pi'_2(\Pi'_1(P))]$

Note the change in order of Π_1 and Π_2 as is usual with forward predicate transformers.

$$P4: [\underline{\text{if}}\ \beta\ \underline{\text{then}}\ \Pi_1\ \underline{\text{else}}\ \Pi_2\ \underline{\text{fi}}]' = \lambda P\hat{}[\Pi_1'\hat{}(\beta'\underline{\wedge}P)\ \underline{\vee}\ \Pi_2'\hat{}(\neg\beta'\underline{\wedge}P)]$$

The formulation on the right hand side uses the connectives $\underline{\vee}$ and $\underline{\wedge}$; without them the expression should be written as:

$$\lambda P\ \hat{}[{}^{\vee}\Pi_1'\ \hat{}(\beta'\wedge{}^{\vee}P)\ \vee\ {}^{\vee}\Pi_2'\hat{}(\ \beta'\wedge{}^{\vee}P)]$$

which involves 4 more occurences of extension symbols.

$$A1: [\chi:=\delta]'_{A1} = \lambda P\ \hat{}\exists z[\{z/{}^{\vee}\chi'\}{}^{\vee}P \wedge {}^{\vee}\chi = \{z/{}^{\vee}\chi'\}\delta']$$

Note that by removing the intension and extension operators this rule reduces to a functional variant of Floyd's assignment rule.

$$A2: [\underline{\text{if}}\ \beta\ \underline{\text{then}}\ \zeta\ \underline{\text{else}}\ \theta\ \underline{\text{fi}}\ :=\ \delta]' = [\underline{\text{if}}\ \beta\ \underline{\text{then}}\ \zeta\ :=\ \delta\ \underline{\text{else}}\ \theta\ :=\ \delta\ \underline{\text{fi}}]'_{P4}$$

$$A3: [\rho[\nu]\ :=\ \delta]' = [\rho\ :=\ \underline{\lambda n}\ \underline{\text{if}}\ \underline{n}\ =\ \nu\ \underline{\text{then}}\ \delta\ \underline{\text{else}}\ \rho[\underline{n}]\underline{\text{fi}}]'_{X}$$

The labelled bracket $]_X$ stands for $]_{A1}$, $]_{A2}$ or $]_{A3}$ depending on the structure of ρ.

The syntax of our fragment does not generate, unlike ALGOL 68, the assignment aa[1]:= 1. This could be provided for by replacing "MODE unit" by "MODE exp" in syntactic rule A3. Consequently translation rule A3 would reduce this to an assignment with at the left hand side a dereferenced expression. This could be dealt with by introducing state operators of the format $\{z/{}^{\vee}{}^{\vee}aa\}$. Such operators, however, would not have such nice syntactic properties as the previously defined one. In order to avoid unnecessarily complications we left such assignments and balancing out of discussion. The semantics of aa[1] occurring on the right hand side is straightforward, but generating it only there would require much more syntactic rules.

6. EXAMPLES.

(1) x:=y

The translation of this assignment is as follows:

$$[x:=[y]_{E2}]'_{A1} = \lambda P\hat{}(\exists z[\{z/{}^{\vee}x\}{}^{\vee}P \wedge {}^{\vee}x = \{z/{}^{\vee}x\}[y]'_{E2}])$$

$$= \lambda P\hat{}\exists z[\{z/{}^{\vee}x\}{}^{\vee}P \wedge {}^{\vee}x = \{z/{}^{\vee}x\}{}^{\vee}y] = \lambda P\hat{}\exists z[\{z/{}^{\vee}x\}{}^{\vee}P \wedge {}^{\vee}x = {}^{\vee}y]$$

So the predicate $\hat{}({}^{\vee}x = 1 \wedge {}^{\vee}y = 2)$ is by this transformer transformed into

$$\hat{}\exists z[\{z/\check{}x\}\check{}\hat{}(\check{}x=1 \wedge \check{}y=2) \wedge \check{}x=\check{}y].$$

We apply $\check{}\hat{}\phi = \phi$ and the syntactic properties of substitution and obtain

$$\hat{}\exists z[z=1 \wedge \check{}y=2 \wedge \check{}x=\check{}y]$$

This reduces to

$$\hat{}(\check{}y=2 \wedge \check{}x=2)$$

We wish to interprete this transformer as follows: if for the initial state s the predicate $\hat{}(\check{}x=1 \wedge \check{}y=2)$ holds, so if $s \models \check{}x=1 \wedge \check{}y=2$, then the predicate $\hat{}(\check{}x=2 \wedge \check{}y=2)$ holds for the state after the execution of the assignment, so $t \models \check{}x=2 \wedge \check{}y=2$. This use of transformers will be justified in the next section, nevertheless we will use already now the corresponding terminology.

(2) x:=1; xx:=x; x:=2

Assume that we execute this program without any information on the initial state. Then we know that in the resulting state holds that $\check{}([x:=1; xx:=x; x:=2]'(\hat{}true))$. The computation proceeds in stages

$$[x:=1]'(\hat{}\underline{true}) = \lambda P\hat{}\exists z[\{z/\check{}x\}\check{}P \wedge \check{}x=1](\hat{}\underline{true})$$

which reduces to $\hat{}(\check{}x=1)$.

$$[xx:=x]'(\hat{}(\check{}x=1)) = \hat{}\exists z[\{z/\check{}xx\}(\check{}x=1) \wedge \check{}xx=x]$$

which reduces to $\hat{}(\check{}x=1 \wedge \check{}xx=x)$. From this we see that after the second assignation holds that $\check{}\check{}xx=1$.

$$[x:=2]'(\hat{}(\check{}x=1 \wedge \check{}xx=x)) = \hat{}\exists z[\{z/\check{}x\}(\check{}x=1 \wedge \check{}xx=x) \wedge \check{}x=2] =$$

$$= \hat{}\exists z[z=1 \wedge \check{}xx=x \wedge \check{}x=2] = \hat{}(\check{}xx=x \wedge \check{}x=2)$$

After the execution of the program holds $\check{}xx=x \wedge \check{}x=2$, so $\check{}\check{}xx=2$

(3) a[a[1]]:=2

We first notice that the syntactic structure of the subscript is $[a[1]]_{E5}]_{E2}$, so its translation is $\check{}(\hat{}((\check{}a)[1]))$, which reduces to $(\check{}a)[1]$. The translation of the program is

$$[a[a[1]]:=2]'_{A3} = [a:= \lambda \underline{n}\ \underline{\text{if}}\ \underline{n}=a[1]\ \underline{\text{then}}\ 2\ \underline{\text{else}}\ a[\underline{n}]\ \underline{\text{fi}}]'_{A1} =$$

$$= \lambda P \hat{}\exists z[\{z/{}^{\vee}a\}^{\vee}P \wedge {}^{\vee}a = \{z/{}^{\vee}a\}[\lambda n\ \underline{if}\ n={}^{\vee}a[1]\ \underline{then}\ 2\ \underline{else}\ {}^{\vee}a[n]\ \underline{fi}\,]] =$$

$$= \lambda P \hat{}\exists z[\{z/{}^{\vee}a\}^{\vee}P \wedge {}^{\vee}a = \lambda n\ \underline{if}\ n=z[1]\ \underline{then}\ 2\ \underline{else}\ z[n]\ \underline{fi}]$$

Assume that before the assignment holds that ${}^{\vee}a[1] = 1 \wedge {}^{\vee}a[2] = 1$. Then after the assignment holds

$$\exists z[z[1]=1 \wedge z[2]=1 \wedge {}^{\vee}a = \lambda n\ \underline{if}\ n=z[1]\ \underline{then}\ 2\ \underline{else}\ z[n]\ \underline{fi}]$$

From this we can derive that afterwards ${}^{\vee}a[1]=2 \wedge {}^{\vee}a[2]=1$ holds, so ${}^{\vee}a[{}^{\vee}a[1]]=1$. We recall that this was one of the examples for which Floyd's rule gave a wrong result. We consider our treatment of this case as an improvement of the solution of DE BAKKER (1976) since it covers all multidimensional cases and is, moreover, less complex.

(4) q[q[1][2]][2]:= 2

Assume that before the assignment holds ${}^{\vee}q[1][2]=1 \wedge {}^{\vee}q[2][2]=3$. Then it is not true that afterwards ${}^{\vee}q[{}^{\vee}q[1][2]][2]=2$ holds. (notice the parallelism with example 3). By two applications of rule A3 we obtain

$$[q[q[1][2]][2]:=2]' = [q[q[1][2]]:=\lambda n\ \text{if}\ n=2\ \underline{then}\ 2\ \underline{else}\ q[q[1][2]][n]\ \underline{fi}]' =$$

$$= [q:= \lambda m\ \underline{if}\ m = q[1][2]\ \underline{then}\ [\lambda n\ \ \underline{if}\ n=2\ \underline{then}\ 2\ \underline{else}$$

$$q[q[1][2]][n]\ \underline{fi}]\ \underline{else}\ q[m]\ \underline{fi}]'$$

So we obtain that afterwards:

$$\exists z[z[1][2]=1 \wedge z[2][2]=3 \wedge {}^{\vee}q[2][2]=z[2][2]=3 \wedge {}^{\vee}q[1][2]={}^{\vee}q[z[1][2]][2]=2]$$

and from this we derive that:

$${}^{\vee}q[{}^{\vee}q[1][2]][2]=3$$

(5) $\underline{\text{if}}\ x > 0\ \underline{\text{then}}\ x\ \underline{\text{else}}\ y\ \underline{\text{fi}} := 3.$

Assume that this program is executed with no information about the initial state. Then in the resulting state holds

$$\exists z_1[z_1 > 0 \wedge {}^{\vee}x=3] \wedge \exists z_2[\neg({}^{\vee}x > 0) \wedge {}^{\vee}y=3]$$

which reduces to

$^{\vee}x=3 \vee (^{\vee}x\leq 0 \wedge {}^{\vee}y=3)$

7. OPERATIONAL SEMANTICS

In this section we will consider a semantics for the programming language fragment that is based upon a more operational interpretation. With each mode expression the computer associates some object in our model (this object might depend on the current state). We denote the object associated with expression ϕ in state s by ϕ^s. The operational semantics is related to the Montague semantics by requiring $\phi^s = \underset{s}{V}\phi$. So the object associated with a complex expression has a certain relation with the objects associated with the subexpressions.

With an assignment two objects are associated: if the right hand object (source) is an object of type τ, the left hand object (destination) is of type $<s,\tau>$. Execution of the assignment brings the computer in a state where the extension of the destination equals the source. So from an operational point of view the semantics of an assignment is a mapping from states to states, rather than mapping from sets of states to sets of states as in the Montague semantics.

By the properness postulate we know that for each state s and each identifier χ holds $\underset{s}{V}(\chi)\in\underline{A} = \underset{s}{V}\underline{A}_{\tau}$. Consequently, if ϕ is the translation of some mode expression, then $\underset{s}{V}(\phi)\in\underline{A}_{\tau}$. We have already introduced constants for each element of $(\underline{A}_{\tau})$, namely the sets $\underline{A}_{\tau}$. These constants can be used to denote in IL the objects ϕ^s. Therefore the operational semantics " of an expression ϕ is defined as a function from states to constants in $\underline{A}_{\tau}$. With use of these constants we introduce new state operators $<\alpha\leftarrow\phi>$ where $\alpha\in\underline{A}_{<s,\tau>}$ and $\phi\in ME_{\tau}$. In this we use an extension of IL in which the constants from $\underline{A}$ are also allowed in the expressions. Their interpretation is defined by $\underset{s}{V}(\alpha) = G(\alpha)$, see section 4 for details concerning G. The new operators constitute mappings from state to states as follows:

$$<\underline{\alpha} \leftarrow \psi>s = <\alpha \leftarrow \psi>s \qquad \text{if } \alpha \text{ is the translation of some identifier.}$$

$$<\rho[\underline{\nu}] \leftarrow \psi>s = <\rho \leftarrow \lambda n\ \underline{\textit{if}}\ n=\nu\ \underline{\textit{then}}\ \psi\ \underline{\textit{else}}\ \rho[n]>s.$$

The operational semantics " of programs is now defined by:

P1: $[\Pi]''_{P1} = \Pi''$

P2: $[\Pi_1;\Pi_2]''_{P2} = \underset{=}{\lambda}s[\Pi_2''(\Pi_1''(s))]$

P3: $[\Pi]''_{P3} = \Pi''$

P4: $[\underline{\text{if}}\ \beta\ \underline{\text{then}}\ \Pi_1\ \underline{\text{else}}\ \Pi_2\ \underline{\text{fi}}]''_{P4} = \underset{=}{\lambda}s[\text{if } \beta''(s) \text{ then } \Pi_1''(s) \text{ else } \Pi_2''(s)].$

A: $[\phi:=\delta]_X = \underset{=}{\lambda}s<\phi'' \leftarrow \delta''>s$

Where X stands for A1, A2, or A3.

If we execute a program Π starting from a state s which satisfies a predicate ϕ, then it is a reasonable requirement that the Montague semantics yields a correct

result with respect to the operational semantics, and gives as much information as possible about the new state. Below we will formulate these requirements formally. The translation function ' is called *correct* with respect to the operational semantics " if for all programs Π, all state predicates ϕ and all states s:

$$\text{if} \quad s \models \check{}\phi \quad \text{then} \quad \Pi''(s) \models \check{}\Pi'(\phi)$$

The translation function ' is called *maximal* with respect to the operational semantics " if for all state predicates ϕ,ψ and all programs Π one has the following:

if for all states s: $s \models \check{}\phi$ implies $\Pi''(s) \models \check{}\psi$

then: $\models \check{}\Pi'(\phi) \rightarrow \check{}\psi$

If the translation rule ' is both correct and maximal with respect to " we say that it produces the *strongest postcondition* for the operational semantics. We say that the translation rule ' is *recoverable* with respect to " if for each state t, for each state predicate ϕ and for each program Π one has

$$t \models \check{}\Pi'(\phi) \Rightarrow \exists s \in S: s \models \check{}\phi \ \& \ \Pi''(s)=t$$

So for each state satisfying the transformed description there is another state which is operationally transformed into it and which satisfies the original description.

THEOREM: *If ' is recoverable then ' is maximal.*

PROOF: Assume that $s \models \check{}\phi$ implies $\Pi''(s) \models \check{}\psi$, but that not holds $\models \check{}\Pi(\phi) \rightarrow \check{}\psi$. Then there is a state t such that $t \models \check{}\Pi(\phi)$ and $t \models \neg\check{}\psi$. Since Π' is recoverable there is a state s such that $s \models \check{}\phi$ and $\Pi''(s)=t$. By assumption we also have $\Pi''(s) \models \check{}\psi$. Contradiction. □

THEOREM: *The translation rule ' yields the strongest postcondition with respect to the operational semantics ".*

PROOF: By induction to the structure of the possible programs. We only consider the case $[\chi:=\delta]_{A1}$.

CORRECTNESS. Let $s \models \check{}\phi$ and $t=\Pi''(s)$. Thus $t = <\underline{\chi} \leftarrow \delta''>s$. We have to prove that

$$t \models \check{} \hat{} \exists z[\{z/\check{}\chi\}\check{}\phi \wedge \check{}(\check{}\chi'=\{z/\check{}\chi\}\delta')]$$

Let h be a fixation such that $h(z) = \underset{s}{V}(\check{}\chi)$. Then for each formula ψ:

$$\underset{t,h}{V}(\{z/\check{}\chi'\}\check{}\psi) = \underset{<\chi' \leftarrow h(z)>t,h}{V}(\psi) = \underset{s,h}{V}(\psi).$$

Therefore

$$t,h \models \{z/\check{}\chi'\}\phi$$

Moreover

$$\underset{t,h}{V}(\{z/\check{}\chi'\}\delta') = \underset{<\chi\leftarrow h(z)>t,h}{V}\delta' = \underset{s,h}{V}\delta' = \underset{t,h}{V}\chi'$$

and thus ' is correct.

RECOVERABILITY. Let

$$t\models\exists z[\{z/\check{}\chi\}\check{}\phi \wedge \check{}\chi=\{z/\check{}\chi'\}\delta']$$

Thus there is a fixation g such that $g(z)$ is an achievable value (by definition of $\exists z$), and with

$$t,g\models\{z/\check{}\chi'\}\check{}\phi \quad \text{and} \quad \underset{t}{V}(\check{}\chi') = \underset{t,g}{V}(\{z/\check{}\chi'\}\delta').$$

We define $s=<\chi\leftarrow g(z)>t$, this state exists since we have the update postulate and $g(z)$ is an achievable value. From the definition of s we may immediately conclude that $s\models\check{}\phi$. We prove now that the value of $\check{}\chi'$ is the same in $\Pi''(s)$ and in t. Since this is the only identifier in which they might differ we conclude that the states are the same (the update postulate guarantees uniqueness!)

$$\underset{\Pi''(s)}{V}(\check{}\chi') = \underset{<\chi\leftarrow\delta''>s}{V}(\check{}\chi') = \underset{s}{V}(\delta') = \underset{<\chi\leftarrow g(z)>t}{V}(\delta') = \underset{t,g}{V}\{z/\check{}\chi'\}\delta' = \underset{t}{V}(\chi').$$

Notice that this proof also holds in case that δ is an λ-expression. □.

8. COMPARISON WITH PTQ.

In the preceding sections we have demonstrated that certain problems concerning the semantics of assignments can be treated by application in this area of Montague's approach to the syntax and semantics of natural languages. In this approach the use of intensional logic is an important tool; in order to deal with the problems under consideration we introduced two new schemes for obtaining meaningful expressions of IL: the *if then else fi* construct, and the state operators $\{z/\check{}c\}$. The former is an inessential extension of IL, the latter clearly gives new expressive power to IL and it is not at all clear whether these operators can be expressed in the original system. Also the model theory had to have been adapted. Whereas Montague uses meaning postulates for defining the subclass of the possible intensional models suitable for the interpretation of English, we use the properness and update

postulate to select the models in which certain intuitions about computer behaviour are respected. Montague used in PTQ a categorial grammar for his fragment of English; the Wijngaarden grammar we use can be considered as a categorial grammar with infinitely many rules. We did however not take over all refinements of the ALGOL 68 assignment rules. A typical difference with PTQ is the second kind of semantics we considered: operational semantics with state transformers.

REFERENCES

DE BAKKER, J.W. (1976), *Correctness proofs for assignment statements*, Report IW 55/76 Mathematical Centre, Amsterdam.

FLOYD, R.W. (1967), *Assigning meanings to programs*, in J.T. SCHWARTZ (ed.) Proc. Symp. in Appl. Math. 19, *Math. aspects of computer sciences*, AMS. pp. 19-32.

GALLIN, D. (1975), *Intensional and higher-order modal logic*, North Holland Publishing Company, Amsterdam.

JANSSEN, T.M.V. (1976), *A computer program for Montague grammar: theoretical aspects and proofs for the reduction rules*, in J. GROENENDIJK & M. STOKHOF (eds.) Amsterdam papers in formal grammar 1, *Proceedings of the Amsterdam colloquium on Montague grammar and related topics*, pp. 154-176 Centrale Interfaculteit, University of Amsterdam.

MONTAGUE, R. (1973), *The proper treatment of quantification in ordinary English*, in J. HINTIKKA, J. MORAVCSIK & P. SUPPES (eds.), *Approaches to natural language*, Reidel, Dordrecht; reprinted in R.H. THOMASON (1974), *Formal Philosophy, Selected papers of Richard Montague*, Yale University press, New Haven and London, pp. 247-270.

PARTEE, B. (1975), *Montague grammar and transformational grammar*, Linguistic Inquiry 6, pp. 203-300.

QUINE, W.V. (1960), *Word and object*, the M.I.T. Press, Cambridge, Mass.

SCOTT, D.(1970), *Advice and modal logic*, in K. LAMBERT (ed.), *Philosophical problems in logic*, Reidel, Dordrecht, pp. 143-173.

SCOTT, D. & C. STRACHEY (1971), *Towards a mathematical semantics for computer languages*, in J. FOX (ed.), Proc. Symp. on Computers and Automata, Polytechnic Institute of Brooklyn, pp. 19-46.

STRACHEY, C. (1966), *Towards a formal semantics*, in T.B. STEEL, jr. (ed.), *Formal language description languages for computer programming*, North Holland Publishing Company, Amsterdam, pp. 198-220.

VAN WIJNGAARDEN, et al. (eds.) (1976), *Revised report on the algorithmic language ALGOL 68*, Tract MCT 50, Mathematical Centre, Amsterdam.