Formal Descriptions of Programming Concepts, E.J. Neuhold (ed.)
North-Holland Publishing Company, (1978)

RECURSIVE PROGRAMS AS PREDICATE TRANSFORMERS

J.W. de Bakker
Mathematical Centre & Free University
Amsterdam

1. INTRODUCTION

Recent developments in programming theory have brought an increasing popularity of the methodology of denotational semantics (e.g. Milne & Strachey (1976)) on the one hand - with as one of its central themes the idea of assigning meaning to a program as a function from states to states - and of the proof-theory-oriented approach (e.g. Dijkstra (1975)) on the other hand, where meaning is (maybe implicitly) assigned to a program through the induced relationship between predicates holding before and after its execution. Our paper is devoted to an analysis of the relationship between the two approaches for an example programming language which includes assignment statements together with sequential composition, nondeterministic choice, if-then-else-fi, and parameterless recursion. The main result of the analysis is a theorem which may be seen as yielding the equivalence of the two approaches. Let us introduce some terminology to state the theorem (the definition reappear in more precise form in the following sections): The set Σ of *states* σ,... consists of all mappings from integer variables (syntactic objects) to integers (mathematical objects), together with the "undefined" state $\perp$. The set P of *predicates* p,... consists of all mappings from states to truth-values (with the convention that $p(\perp)$ yields false). The *meaning* of a statement S is obtained by applying a semantic function M (defined by induction on the structure of S) yielding $M(S) \in \Sigma \to \Sigma$. (Effects of nondeterminacy are not yet taken into account here.) Furthermore, for each statement S we define (syntactically) a corresponding "predicate transformer expression" $\tilde{S}$, and the meaning of an expression such as $\tilde{S}$ is obtained by applying a semantic function F yielding $F(\tilde{S}) \in P \to P$. (Note, however, that a development of the theory with "~" as the identity transformation is also possible: The same piece of text S is then on the one hand given meaning as a state transformation and on the other hand as a predicate transformation. The approach taken here was chosen mainly for didactic reasons, emphasizing our "dualistic" view of programs both syntactically and semantically.) The relationship between programs as state transformations and as predicate transformations is now expressed in

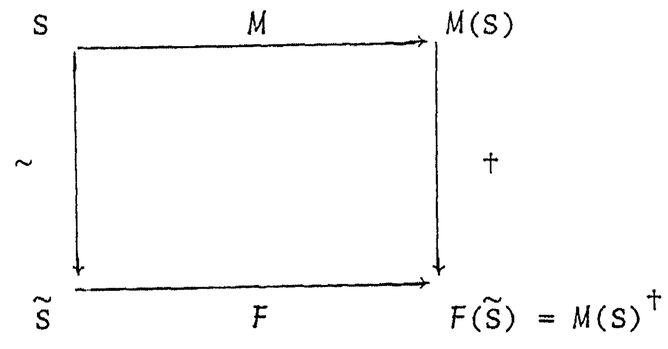
Main theorem. For each statement S, predicate p and state σ:

$$F(\tilde{S})(p)(\sigma) = p(M(S)(\sigma)).$$

For the reader who likes diagrams, the theorem may also be stated as follows: Let, for each $m \in \Sigma \to \Sigma$, $m^\dagger$ be defined as

$$m^\dagger = \lambda p \cdot \lambda \sigma \cdot p(m(\sigma)).$$

Then we have that the following diagram commutes:



The approach of viewing programs as predicate transformers has in particular been advocated by Dijkstra (1975), partly building upon earlier work by Floyd and Hoare, and the question may arise as to how his ideas connect with the present result. The answer is simple: Let, for each statement S and predicate p, $wp(M(S),p)$ be the *weakest precondition* of (the meaning of) S with respect to predicate p. (Remember that $wp(M(S),p)$ is defined as the weakest predicate such that, for all states σ, whenever σ satisfies $wp(M(S),p)$, then execution of S for input σ terminates with output state satisfying p.) We shall show that $wp(M(S),p) = \lambda\sigma \cdot p(M(S)(\sigma))$, and we have the following corollary of our main theorem:

$$F(\tilde{S})(p) = wp(M(S),p).$$

Observe that we introduce *wp* here as a *mathematical* object yielding, for each $m \in \Sigma \to \Sigma$ and $p \in P$, a result $p' = wp(m,p) \in P$. We also discuss the problem as to whether it is possible to give a *syntactic* characterization of *wp*. To be more specific, let us consider the (syntactic) class of assertions (which contains, e.g., the class of boolean expressions as a subset) with elements c,... . Let T be the semantic mapping which provides an assertion c with a meaning $T(c) \in P$. Let S:c be the above mentioned syntactic counterpart of *wp*, i.e., we have as intended meaning of S:c

$$T(S{:}c) = wp(M(S),T(c)).$$

As is well-known, it is possible to give meaning-preserving reductions of S:c to assertions involving only the components of S, in the case S is an assignment statement, or made up using sequential composition, nondeterministic choice, and <u>if</u>-<u>then</u>-<u>else</u>-<u>fi</u>. We describe how to extend the syntax of assertions in order to be able to perform a similar reduction for the while statement (or, somewhat more generally, for programs derivable from flow diagrams). For recursive procedures in general, however, we do not know how to do this, and we conjecture that such reduction is impossible.

The framework used to prove our main theorem is a more or less familiar part of denotational semantics, extended with the Egli-Milner-ordering (Egli (1975)), to deal with nondeterminacy. For an explanation and motivation of this ordering the reader might want to consult our De Bakker (1976).(The present paper may be seen as a successor to De Bakker (1976). Note, however, the difference in the use of ~; also, as pointed out by John Reynolds, in De Bakker (1976) we should not have omitted the restriction of *bounded* nondeterminacy.)

Besides by the wish to extend De Bakker (1976), our paper has also been motivated by De Roever's (1976) article, in which the same problem is investigated. Moreover, our definition of "~" for recursive procedures is taken from De Roever (1976) (credited there to C.P. Wadsworth).

Our paper is organised as follows: The syntax of the example language, together with the definition of the mapping "~" is given in section 2. Section 3 provides the necessary background from denotational semantics, including a treatment of the consequences of introducing the Egli-Milner-ordering. Section 4 presents the semantic mappings, elaborated in section 5 for integer and boolean expressions, and in section 6 for statements and predicate transformer expressions. The least-fixed-point approach plays a major part in the latter. Section 7 gives (without proof) a justification of the definitions in section 6, and section 8 contains the proof of the main theorem. Section 9 brings the link with the notion of weakest precondition, and section 10 introduces the extension of the class of boolean expressions to the class of assertions including the S:c construct. The well-known reductions of S:c for simple S, as mentioned above, are then given together with a treatment of *iterative* S. The section closes with a comment on a recent theorem by Basu & Yeh (1975), pointing out an error in one of their results on *wp* for the while statement. (For section 10 compare also our De Bakker (to appear)).

As closing remark of this introduction, let us point out that we hope to have achieved with our paper both a clarification of the status of recursive programs as predicate transformers, and an illustration of some of the more appealing features of denotational semantics, such as its expressive power and its succinctness of argumentation.

Acknowledgements. I am indebted to A. Nijholt and W.P. de Roever for a number of helpful discussions on the subject of this paper.

2. SYNTACTIC DOMAINS

This section gives the syntax of our programming language. Besides some simple kinds of integer and boolean expressions, it contains statements made up from assignment statements through sequential composition, nondeterministic choice, if-then-else-fi, and parameterless recursion. Moreover, the class of *predicate transformer expressions*

is introduced, and a syntactic mapping from statements to predicate transformer expressions is defined. The formalism used in the syntactic definitions is a slight variant of BNF, and should be self-explanatory. Throughout the paper, we do not bother about syntactic ambiguities which may be remedied by suitable addition of parentheses. "≡" is used to denote syntactic identity.

			Elements
(D1)	$Cnst = \{a_1, a_2, \ldots\}$	constants	a,...
(D2)	$Ivar = \{x_1, x_2, \ldots\}$	integer variables	x,...
(D3)	$Pvar = \{X_1, X_2, \ldots\}$	procedure variables	X,...
(D4)	$Tvar = \{\xi_1, \xi_2, \ldots\}$	predicate transformer variables	ξ,...
(D5)	*Iexp*	integer expressions	s,...

s::= $a \mid x \mid s_1+s_2 \mid$ <u>if</u> b <u>then</u> s_1 <u>else</u> s_2 <u>fi</u>

Integer expressions have integers as intended meaning.
Other arithmetic operations may be added, if desired.

(D6) *Bexp* — boolean expressions — b,...

b::= <u>true</u>|<u>false</u>|$s_1=s_2 \mid \neg b \mid b_1 \supset b_2 \mid b_1 \wedge b_2$

Boolean expressions have truth values as intended meaning.

(D7) *Stat* — statements — S,...

S::= $x:=s \mid X \mid S_1;S_2 \mid S_1 \cup S_2 \mid$ <u>if</u> b <u>then</u> S_1 <u>else</u> S_2 <u>fi</u>$\mid \mu X[S]$

Statements have functions from states to states as intended meaning.
Procedure variables are used (only) in the construct μX[S] of *parameterless recursion:* Occurrences of X in S correspond to recursive calls, and, for S ≡ ... X ... X ... , the meaning of the construct μX[... X ... X ...] is the same as the meaning - in an ALGOL-like language - of a call of the parameterless recursive procedure P with declaration <u>procedure</u> P;... P ... P.... $S_1 \cup S_2$ is executed by executing either S_1 or S_2 (not both).

(D8) *Prtr* — predicate transformer expressions φ,...

φ::= $[x:=s] \mid \xi \mid \phi_1 \circ \phi_2 \mid \phi_1 \wedge \phi_2 \mid$ <u>if</u> b <u>then</u> ϕ_1 <u>else</u> ϕ_2 <u>fi</u>$\mid \mu\xi[\phi]$

Let a *predicate* be a function form states to truth-values. Predicate transformer expressions have functions from predicates to predicates as intended meaning.

We now define a mapping "~" from *Stat* to *Prtr* in

(D9) $(x:=s)^{\sim} \equiv [x:=s]$

$\tilde{X} \equiv \xi$ (it is required that $\tilde{X} \equiv \tilde{Y}$ iff $X \equiv Y$)

$(S_1;S_2)^{\sim} \equiv \tilde{S}_1 \circ \tilde{S}_2$

$(S_1 \cup S_2)^{\sim} \equiv \tilde{S}_1 \wedge \tilde{S}_2$

$(\underline{\text{if}}\ b\ \underline{\text{then}}\ S_1\ \underline{\text{else}}\ S_2\ \underline{\text{fi}})^{\sim} \equiv \underline{\text{if}}\ b\ \underline{\text{then}}\ \tilde{S}_1\ \underline{\text{else}}\ \tilde{S}_2\ \underline{\text{fi}}$

$\mu X[S]^{\sim} \equiv \mu \tilde{X}[\tilde{S}]$

Example.

$\mu X[\underline{\text{if}}\ x=a_1\ \underline{\text{then}}\ x:=x+a_2;X;x:=x-a_3\ \underline{\text{else}}\ x:=a_4\ \underline{\text{fi}} \cup (x:=a_5)]^{\sim} \equiv$

$\mu\xi[\underline{\text{if}}\ x=a_1\ \underline{\text{then}}\ [x:=x+a_2]\circ\xi\circ[x:=x-a_3]\ \underline{\text{else}}\ [x:=a_4]\ \underline{\text{fi}} \wedge [x:=a_5]]$.

3. MATHEMATICAL DOMAINS

In denotational semantics, meaning is attributed to programming constructs by mapping them to mathematical domains provided with a certain structure. In our explanation of this, we shall sometimes refer to the underlying intuition about the meaning of these constructs as determined by *operational* semantics. However, we omit formal specification of this, which would proceed through one of the customary schemes for viewing programs as producing computation sequences.

In subsection 3.1 we introduce the basic domains V and W of integers and truth-values, together with the necessary operations. Subsections 3.2 and 3.4 summarize (without proofs) some of the essential facts about complete partially ordered sets (cpo's), continuity, and least fixed points. Subsection 3.3 forms the heart of the definitions. The set of *states* $\Sigma = (Ivar \to V) \cup \{\perp\}$ is introduced (as usual, "$\perp$" stands for the undefined element), and the Egli-Milner-proposal (Egli (1975)) to deal with nondeterminacy is used: The meaning $m = M(S)$ of a statement S will be given as an element of the cpo $M = \Sigma \to' T$ (i.e., all *strict* functions $\Sigma \to T$), where T is the collection of all subsets of Σ which are either finite, or, when infinite, contain $\perp$ as element. This is the usual restriction to *bounded* nondeterminacy *): For given input state, an infinite number of output states is allowed only when $\perp$ is among them. E.g., consider the statement $\mu X[(x:=x+1;X) \cup (x:=x)]$ for input state satisfying $x = 0$. This statement determines an infinite number of finite computations leading to an infinite set of output states (all natural numbers are possible output values for x), but also an infinite computation (always choose the first branch of the choice) yielding $\perp$ as corresponding output state. Subsection 3.3 also introduces the domains W of truth values, $P = [\Sigma \to W]$ of predicates, and $PT = [P \to P]$ of predicate transformers, which definitions involve no special features. We draw attention, however, to the definition of the extension of $[\Sigma \to W]$ to $[T \to W]$: For each predicate $p \in P$ and each $\tau \in T$, $p(\tau)$ is true iff τ does not contain $\perp$, and (τ thus being finite), $p(\sigma)$ holds for each $\sigma \in \tau$. Anticipating some of the definitions to be given later, we observe already that this corresponds to the weakest precondition, in the case that $\tau = m(\sigma)$ is obtained by applying the meaning $m = M(S)$ of some state-

---

*) As pointed out by John Reynolds, this restriction should not have been omitted in our De Bakker (1976).

ment S to input state σ: $p(\tau) = p(m(\sigma)) = (p \circ m)(\sigma) = (p \circ M(S))(\sigma) = wp(M(S),p)(\sigma)$. Note that *wp* is defined here as an operation upon *mathematical* objects. The possibility of a *linguistic* counterpart of *wp* is discussed in section 10.

3.1. *Basic domains and notations*

			Elements
(D10)	$V = \{...,-1,0,1,...\}$	integers	$\alpha,...$
(D11)	$W = \{tt,ff\}$	truth-values	$\beta,...$

(D12) We assume known

$plus : V \times V \to V$ $\quad\quad \Rightarrow : W \times W \to W$

$equal: V \times V \to W$ $\quad\quad \wedge : W \times W \to W$

$not : W \to W$

(D13) " $\sqsubseteq$ " and "$\cup$" have the usual set-theoretical meaning

(D14) For any set C, $c_1, c_2 \in C$ and $\beta \in W$, we put

$$\underline{\text{if}}\ \beta\ \underline{\text{then}}\ c_1\ \underline{\text{else}}\ c_2\ \underline{\text{fi}} = \begin{cases} c_1, & \text{if } \beta = tt \\ c_2, & \text{if } \beta = ff \end{cases}$$

3.2 *Complete partially ordered sets (cpo's)*

(D15) Let C be a set with a partial ordering $\sqsubseteq_C$. When confusion is unlikely, we omit the index C in the ordering $\sqsubseteq_C$. C is called a cpo iff

(i) C has a least element $\perp$ such that $\perp \sqsubseteq c$ for all $c \in C$.

(ii) Each *chain* $c_0 \sqsubseteq ... \sqsubseteq c_i \sqsubseteq ...$ has a least upper bound $\bigsqcup_{i=0}^{\infty} c_i$.

(D16) Let C,D be two cpo's, and let $e_1, e_2: C \to D$. We put $e_1 \sqsubseteq_{C\to D} e_2$ iff, for all c, $e_1(c) \sqsubseteq_D e_2(c)$.

(D17) Let C,D be two cpo's and let e: C → D be *monotonic* ($c_1 \sqsubseteq c_2 \Rightarrow e(c_1) \sqsubseteq e(c_2)$). We call e *continuous* iff, for each chain $\{c_i\}_{i=0}^{\infty}$, $e(\bigsqcup_i c_i) = \bigsqcup_i e(c_i)$.

(D18) Let C,D be two cpo's. [C → D] denotes the collection of all continuous functions: C → D.

(L1) (First lemma) For C,D cpo's, [C → D] is also a cpo. Also, for each chain $\{e_i\}_{i=0}^{\infty}$: $(\bigsqcup_i e_i)(c) = \bigsqcup_i (e_i(c))$.

(L2) For each cpo C and $\Phi \in [C \to C]$, Φ has a *least fixed point* $\mu\Phi = \bigsqcup_{i=0}^{\infty} \Phi^i(\perp)$, satisfying:

(i) $\Phi(\mu\Phi) = \mu\Phi$

(ii) $\Phi(c) = c \Rightarrow \mu\Phi \sqsubseteq c$.

3.1. *Basic domains and function domains as cpo's*

(D19) $\Sigma_0 = Ivar \to V$

(D20) $\Sigma = \Sigma_0 \cup \{\perp\}$ (Σ has elements $\sigma,...$)

(D21) $\sigma_1 \sqsubseteq_\Sigma \sigma_2$ iff $\sigma_1 = \perp$ or $\sigma_1 = \sigma_2$

(L3) Σ is a cpo with respect to $\sqsubseteq_\Sigma$

(D22) $W = \{tt,ff\}$ $(=D_{11})$

(D23) $\beta_1 \sqsubseteq_W \beta_2$ iff $\beta_1 \Rightarrow \beta_2$

(D24) $\perp_W = ff$

(L4) W is a cpo with respect to $\sqsubseteq_W$

(D25) T is the collection of all subsets τ of Σ such that if $\perp \notin \tau$ then τ is finite

(D26) (Egli-Milner)

$\tau_1 \sqsubseteq_T \tau_2$ iff either $\perp \in \tau_1$ and $\tau_1 \setminus \{\perp\} \subseteq \tau_2$, or $\perp \notin \tau_1$ and $\tau_1 = \tau_2$

(Let $\tau_1 \sqsubseteq \tau_2$. Interpreting τ_1 and τ_2 as approximations to the outcome $M(S)(\sigma)$ of (the computation specified by) a statement S for a given input state σ, we see that τ_1 can be properly contained (in the settheoretic sense) in τ_2 only if $\perp \in \tau_1$: The presence of $\perp$ in τ_1 indicates a path in the computation which has not (yet) delivered a result. Note also that, though it is true that $M(x:=1) \subseteq M((x:=1) \cup (x:=2))$, we do not have that $M(x:=1) \sqsubseteq M((x:=1) \cup (x:=2).)$

(D27) $\perp_T = \{\perp_\Sigma\}$

(L5) T is a cpo with respect to $\sqsubseteq_T$

(D28) $M = \Sigma \to' T$, which is, by definition, the collection of all functions $m: \Sigma \to T$ such that $m(\perp) = \{\perp\}$

(L6) $M \subseteq [\Sigma \to T]$ and M is a cpo

(D29) $P = \Sigma \to' W$, which is, by definition, the collection of all functions $p: \Sigma \to W$ such that $p(\perp) = ff$

(L7) $P \subseteq [\Sigma \to W]$ and P is a cpo

(D30) $PT = [P \to P]$, hence PT is a cpo (elements of PT are denoted by f,...)

We now extend $m: \Sigma \to' T$ to $\hat{m}: T \to T$ and $p: \Sigma \to' W$ to $\hat{p}: T \to W$, as follows:

(D31) $\hat{m}(\tau) = \bigcup_{\sigma \in \tau} m(\sigma)$

(D32) $\hat{p}(\tau) = \begin{cases} ff & , \text{ if } \perp \in \tau \\ \bigwedge_{\sigma \in \tau} p(\sigma), & \text{if } \perp \notin \tau \end{cases}$

(L8) $\tau_1 \sqsubseteq \tau_2 \Rightarrow \hat{m}(\tau_1) \sqsubseteq \hat{m}(\tau_2)$, $\tau_1 \sqsubseteq \tau_2 \Rightarrow \hat{p}(\tau_1) \sqsubseteq \hat{p}(\tau_2)$

(L9) (D31) and (D32) preserve continuity. Also, $(\bigsqcup_i m_i)\hat{} = \bigsqcup_i \hat{m}_i$, and $(\bigsqcup_i p_i)\hat{} = \bigsqcup_i \hat{p}_i$. (Note that these equalities do not hold, in general, without the condition of *bounded* nondeterminacy.)

(D33) $m_1 \circ m_2 = \lambda\sigma \cdot \hat{m}_1(m_2(\sigma))$, $m_1 \cup m_2 = \lambda\sigma \cdot m_1(\sigma) \cup m_2(\sigma)$

(L10) If $m_1, m_2 \in M$ then $m_1 \circ m_2$ and $m_1 \cup m_2 \in M$

(D34) $p \circ m = \lambda\sigma \cdot \hat{p}(m(\sigma))$

(L11) If $p \in P$ and $m \in M$ then $p \circ m \in P$

(C1) (First corollary) For $\{m_i\}_{i=0}^{\infty}$ a chain, $p \circ \bigsqcup_i m_i = \bigsqcup_i (p \circ m_i)$

(D35) $f_1 \circ f_2 = \lambda p \cdot f_1(f_2(p))$, $f_1 \wedge f_2 = \lambda p \cdot \lambda\sigma \cdot f_1(p)(\sigma) \wedge f_2(p)(\sigma)$

(L12) If $f_1, f_2 \in PT$, then $f_1 \circ f_2$ and $f_1 \wedge f_2 \in PT$.

Below, we omit explicit indication of "$\wedge$" on m or p.

3.4. *Properties of continuous functions:* $M \to M$

(D36) $M_1 = M$, $M_{n+1} = [M \to M_n]$

Lemma's (L13) to (L16) hold for each $n \geq 0$:

(L13) For each i such that $1 \leq i \leq n$: $\lambda m_1 \cdot \ldots \cdot \lambda m_n \cdot m_i \in M_{n+1}$.

(L14) For each $m \in M$, $\lambda m_1 \cdot \ldots \cdot \lambda m_n \cdot m \in M_{n+1}$

(L15) If $\Phi_1, \Phi_2 \in M_{n+1}$, then

$\lambda m_1 \cdot \ldots \cdot \lambda m_n \cdot \Phi_1(m_1)\ldots(m_n) \circ \Phi_2(m_1)\ldots(m_n) \in M_{n+1}$, and

$\lambda m_1 \cdot \ldots \cdot \lambda m_n \cdot \Phi_1(m_1)\ldots(m_n) \cup \Phi_2(m_1)\ldots(m_n) \in M_{n+1}$

(L16) If $\Phi \in M_{n+2}$, then

$\lambda m_1 \cdot \ldots \cdot \lambda m_n \cdot \mu[\lambda m_{n+1} \cdot \Phi(m_1)\ldots(m_n)(m_{n+1})] \in M_{n+1}$

(C2) Let $PT_1 = PT$, $PT_{n+1} = [PT \to PT_n]$, $n \geq 0$.

Results analogous to (L13) to (L16) hold for PT_{n+1}, $n \geq 0$.

4. SEMANTIC MAPPINGS

In this section we introduce the semantic functions mapping the elements in the four syntactic domains to their respective meanings in the corresponding mathematical domains. In each case, the semantic function is defined with respect to some given mapping from the integer-, procedure-, and predicate transformer variables to their meanings. These initially given mappings (elements of Σ, Γ and Θ) are subject to change during the evaluation of the semantic function through the effects of assignment on states $\sigma \in \Sigma$, and the effect of the least-fixed-point definition of recursion on elements $\gamma \in \Gamma$ and $\theta \in \Theta$. The notation used to describe these effects is also given in this section ((D41) to (D43)).

Elements

(D37) $A \in \mathit{Cnst} \to V$

(the function A remains the same throughout the paper)

(D38) $\Sigma = (\mathit{Ivar} \to V) \cup \{\perp\}$ $\sigma, \ldots$

(D39) $\Gamma = \mathit{Pvar} \to M$ $\gamma, \ldots$

(D40) $\Theta = \mathit{Tvar} \to PT$ $\theta, \ldots$

For each $\sigma \in \Sigma$, $\alpha \in V$, $x \in \mathit{Ivar}$, $\sigma\{\alpha/x\} \in \Sigma$ is defined by

(D41) $\perp\{\alpha/x\} = \perp$, and, for $\sigma \neq \perp$, $\sigma\{\alpha/x\}\{x\} = \alpha$, $\sigma\{\alpha/x\}(y) = \sigma(y)$ for each $y \neq x$.

Similarly.

(D42) $\gamma\{m/X\}(X) = m$, $\gamma\{m/X\}(Y) = \gamma(Y)$ for each $Y \neq X$

(D43) $\theta\{f/\xi\}(\xi) = f$, $\theta\{f/\xi\}(\eta) = \theta(\eta)$ for each $\eta \neq \xi$

(D44) V: $\mathit{Iexp} \to (\Sigma_0 \to V)$

T: $\mathit{Bexp} \to (\Sigma \to W)$

M: $\mathit{Stat} \to (\Gamma \to M)$

F: $\mathit{Prtr} \to (\Theta \to PT)$

(Extension of V to deal with $\mathit{V}(s)(\perp)$ would require a cpo structure on V; this

serves no further purpose in our paper and is therefore omitted.)
The definitions of V, T, M and F follow in sections 5 and 6.

5. SEMANTICS OF INTEGER AND BOOLEAN EXPRESSIONS

This section brings the definitions of the meaning of integer and boolean expressions, using the domains and functions as given in section 3. The definitions are straightforward and do not require additional comment.

(D48) $V(a)(\sigma) = A(a)$
(D49 $V(x)(\sigma) = \sigma(x)$
(D50) $V(s_1+s_2)(\sigma) = plus\ (V(s_1)(\sigma), V(s_2)(\sigma))$
(D51) $V(\underline{if}\ b\ \underline{then}\ s_1\ \underline{else}\ s_2\ \underline{fi})(\sigma) = \underline{if}\ T(b)(\sigma)\ \underline{then}\ V(s_1)(\sigma)\ \underline{else}\ V(s_2)(\sigma)\ \underline{fi}$
(D52) $T(b)(\perp) = ff$
In (D53) to (D57), we assume $\sigma \neq \perp$.
(D53) $T(\underline{true})(\sigma) = tt,\ \ T(\underline{false})(\sigma) = ff$
(D54) $T(s_1=s_2)(\sigma) = equal\ (V(s_1)(\sigma), V(s_2)(\sigma))$
(D55) $T(\neg b)(\sigma) = not\ (T(b)(\sigma))$
(D56) $T(b_1 \supset b_2)(\sigma) = (T(b_1)(\sigma) \Rightarrow T(b_2)(\sigma))$
(D57) $T(b_1 \wedge b_2)(\sigma) = (T(b_1)(\sigma) \wedge T(b_2)(\sigma))$

6. SEMANTICS OF STATEMENTS AND PREDICATE TRANSFORMER EXPRESSIONS

(D58) $M(x:=s)(\gamma) = \lambda\sigma\cdot\{\underline{if}\ \sigma = \perp\ \underline{then}\ \perp\ \underline{else}\ \sigma\{V(s)(\sigma)/x\}\ \underline{fi}\}$
(D59) $M(X)(\gamma) = \gamma(X)$
(D60) $M(S_1;S_2)(\gamma) = M(S_2)(\gamma) \circ M(S_1)(\gamma)$
(D61) $M(S_1 \cup S_2)(\gamma) = M(S_1)(\gamma) \cup M(S_2)(\gamma)$
(D62) $M(\underline{if}\ b\ \underline{then}\ S_1\ \underline{else}\ S_2\ \underline{fi})(\gamma) =$
$\lambda\sigma\cdot\ \underline{if}\ \sigma = \perp\ \underline{then}\ \{\perp\}\ \underline{else}\ \underline{if}\ T(b)(\sigma)\ \underline{then}\ M(S_1)(\gamma)(\sigma)\ \underline{else}\ M(S_2)(\gamma)(\sigma)\ \underline{fi}\ \underline{fi}$
(D63) $M(\mu X[S])(\gamma) = \mu[\lambda m\cdot M(S)(\gamma\{m/X\})]$.

Examples. (identifying for simplicity's sake *Cnst* and V). Let $\sigma \neq \perp$.

1. $M(x:=0;y:=x+1)(\gamma)(\sigma) = M(y:=x+1)(\gamma)(M(x:=0)(\gamma)(\sigma)) =$
$M(y:=x+1)(\gamma)(\sigma\{0/x\}) = \sigma\{0/x\}\{V(x+1)(\sigma\{0/x\})/y\} =$
$\sigma\{0/x\}\{plus(V(x)(\sigma\{0/x\}),V(1)(\sigma\{0/x\}))/y\} =$
$\sigma\{0/x\}\{plus(0,1)/y\} = \sigma\{0/x\}\{1/y\}$.
2. $M((x:=1)\cup(x:=2))(\gamma)(\sigma) = \{\sigma\{1/x\},\sigma\{2/x\}\}$.

Remarks

1. Note that for programs without recursion (and without free procedure variables), the definition of M is in fact independent of γ.

2. For a justification, in the framework of operational semantics, of the least-fixed-point definition of recursion, we refer to De Bakker (1976).
3. For each $S \in Stat$ and $\gamma \in \Gamma$, we have that $M(S)(\gamma) \in M$.
(This is a special case of theorem (T1.1) below.) Also, for each S, γ and X, $\lambda m \cdot M(S)(\gamma\{m/X\}) \in [M \to M]$ (also from (T1.1)). Thus, putting $\Phi \stackrel{df.}{=} \lambda m \cdot M(S)(\gamma\{m/X\})$, Φ has a least fixed point $\mu\Phi = \bigsqcup_i \Phi^i(\bot)$. From (D63) we therefore obtain $M(\mu X[S])(\gamma) = \mu\Phi = \bigsqcup_i m_i$, with $m_0 = \bot_M = \lambda\sigma \cdot \{\bot\}$, and $m_{i+1} = \Phi(m_i) = M(S)(\gamma\{m_i/X\})$.
(This result will be used in the proof of theorem (T2).)

(D64) $F([x:=s])(\theta) = \lambda p \cdot \lambda\sigma \cdot p(\underline{\text{if}}\ \sigma = \bot\ \underline{\text{then}}\ \bot\ \underline{\text{else}}\ \sigma\{V(s)(\sigma)/x\}\ \underline{\text{fi}})$

(D65) $F(\xi)(\theta) = \theta(\xi)$

(D66) $F(\phi_1 \circ \phi_2)(\theta) = F(\phi_1)(\theta) \circ F(\phi_2)(\theta)$

(D67) $F(\phi_1 \wedge \phi_2)(\theta) = F(\phi_1)(\theta) \wedge F(\phi_2)(\theta)$

(D68) $F(\underline{\text{if}}\ b\ \underline{\text{then}}\ \phi_1\ \underline{\text{else}}\ \phi_2\ \underline{\text{fi}})(\theta) =$
$\lambda p \cdot \lambda\sigma \cdot \underline{\text{if}}\ T(b)(\sigma)\ \underline{\text{then}}\ F(\phi_1)(\theta)(p)(\sigma)\ \underline{\text{else}}\ F(\phi_2)(\theta)(p)(\sigma)\ \underline{\text{fi}}$

(D69) $F(\mu\xi[\phi])(\theta) = \mu[\lambda f \cdot F(\phi)(\theta\{f/\xi\})]$

Remarks

Similar to the remarks on the definition of M.

7. FIRST THEOREM

(T1.1) For each $n \geq 0$, $S \in Stat$, $\gamma \in \Gamma$, and $X_1, \ldots, X_n \in Pvar$:
$\lambda m_1 \cdot \ldots \cdot \lambda m_n \cdot M(S)(\gamma\{m_1/X_1\} \ldots \{m_n/X_n\}) \in M_{n+1}$.

(T1.2) For each $n \geq 0$, $\phi \in Prtr$, $\theta \in \Theta$, and $\xi_1, \ldots, \xi_n \in Tvar$:
$\lambda f_1 \cdot \ldots \cdot \lambda f_n \cdot F(\phi)(\theta\{f_1/\xi_1\} \ldots \{f_n/\xi_n\}) \in PT_{n+1}$.

The proof is direct by induction on the complexity of S or ϕ (reduce the assertion of the theorem for some S (or ϕ) and all n to the same assertion for all S' (or ϕ') of less complexity and all n), using lemma's (L13) to (L16) and their analogues.

8. SECOND THEOREM

The theorem of this section is the central one of our paper, because it determines the relationship between the meaning of a statement S as state-transforming function ($M(S)$) and as predicate transforming function ($F(\tilde{S})$). (The connection with weakest preconditions follows in section 9.) The theorem is subjected to the condition of *consistency* of the pair $\langle\gamma,\theta\rangle$, which is nothing but a way of ensuring that the theorem holds (by definition) in the case that S is simply a procedure variable X.

(D70) (Consistency) Let $\gamma \in \Gamma$, $\theta \in \Theta$. The pair $\langle\gamma,\theta\rangle$ is called *consistent* iff, for each $\langle X,\xi\rangle$ such that $\tilde{X} \equiv \xi$, we have that

$$\theta(\xi) = \lambda p \cdot p \circ \gamma(X).$$

(T2) For each S, and each γ and θ such that $<\gamma,\theta>$ is consistent:

$$\tilde{F}(S)(\theta) = p \cdot p \circ (M(S)(\gamma)).$$

Proof. Induction on the complexity of S.

1. $\tilde{F}((x:=s)^{\sim})(\theta)(p)(\sigma) = (df.^{\sim})\tilde{F}([x:=s])(\theta)(p)(\sigma) = (df.\tilde{F})$
   $p(\underline{if}\ \sigma = \perp\ \underline{then}\ \perp\ \underline{else}\ \sigma\{V(s)(\sigma)/x\}\underline{fi}) = (df.M)p(M(x:=s)(\gamma)(\sigma))$
2. $\tilde{F}(\tilde{X})(\theta)(p)(\sigma) = (df.^{\sim})F(\xi)(\theta)(p)(\sigma) = (df.\tilde{F})$
   $\theta(\xi)(p)(\sigma) = (\text{consistency})p(\gamma(X)(\sigma)) = (df.M)p(M(X)(\gamma)(\sigma))$
3. $\tilde{F}((S_1;S_2)^{\sim})(\theta)(p)(\sigma) =$ $(df.^{\sim})$
   $\tilde{F}(\tilde{S}_1 \circ \tilde{S}_2)(\theta)(p)(\sigma) =$ $(df.\tilde{F})$
   $(\tilde{F}(\tilde{S}_1)(\theta) \circ \tilde{F}(\tilde{S}_2)(\theta))(p)(\sigma) =$ $(df.\circ)$
   $\tilde{F}(\tilde{S}_1)(\theta)(\tilde{F}(\tilde{S}_2)(\theta)(p))(\sigma) =$ (ind.)
   $\tilde{F}(\tilde{S}_2)(\theta)(p)(M(S_1)(\gamma)(\sigma)) =$ (ind.)
   $p(M(S_2)(\gamma)(M(S_1)(\gamma)(\sigma)) =$ (df.M)
   $p(M(S_1;S_2)(\gamma)(\sigma))$
4. $M((S_1 \cup S_2)^{\sim})(\theta)(p)(\sigma) =$ $(df.^{\sim})$
   $\tilde{F}(\tilde{S}_1 \wedge \tilde{S}_2)(\theta)(p)(\sigma) =$ $(df.\tilde{F})$
   $\tilde{F}(S_1)(\theta)(p)(\sigma) \wedge \tilde{F}(S_2)(\theta)(p)(\sigma) =$ (ind.)
   $p(M(S_1)(\gamma)(\sigma)) \wedge p(M(S_2)(\gamma)(\sigma)) =$ (D32)
   $p(M(S_1)(\gamma)(\sigma) \cup M(S_2)(\gamma)(\sigma)) =$ (df.M)
   $p(M(S_1 \cup S_2)(\gamma)(\sigma))$
5. $\underline{if}$-$\underline{then}$-$\underline{else}$-$\underline{fi}$ case omitted.
6. $\tilde{F}(\mu X[S]^{\sim})(\theta)(p)(\sigma) =$ $(df.^{\sim})$
   $\tilde{F}(\mu\xi[\tilde{S}])(\theta)(p)(\sigma) =$ $(df.\tilde{F})$
   $\mu[\lambda f \cdot \tilde{F}(\tilde{S})(\theta\{f/\xi\})](p)(\sigma) =$ $(T1.2)$
   $(\bigsqcup_i f_i)(p)(\sigma)$, (with $f_0 = \lambda p \cdot \lambda\sigma \cdot ff$, $f_{i+1} = \tilde{F}(\tilde{S})(\theta\{f_i/\xi\})$) =
   $\bigsqcup_i (f_i(p)(\sigma))$, by (L1).
   Also,
   $p(M(\mu X[S])(\gamma)(\sigma)) =$ $(df.M)$
   $p(\mu[\lambda m \cdot M(S)(\gamma\{m/X\})](\sigma)) =$ $(T1.1)$
   $p((\bigsqcup_i m_i)(\sigma))$, (with $m_0 = \lambda\sigma \cdot \{\perp\}$, $m_{i+1} = M(S)(\gamma\{m_i/X\})$) =
   $\bigsqcup_i p(m_i(\sigma))$, by (C1).
   Thus it is sufficient to show: For all i, and all p and σ,
   $f_i(p)(\sigma) = p(m_i(\sigma))$. We use induction on i:
   (i) i = 0: $(\lambda p \cdot \lambda\sigma \cdot ff)(p)(\sigma) = ff = p(\{\perp\}) = p((\lambda\sigma \cdot \{\perp\})(\sigma))$.
   (ii) Assume (*): $f_i(p)(\sigma) = p(m_i(\sigma))$. To show $f_{i+1}(p)(\sigma) = p(m_{i+1}(\sigma))$, or $\tilde{F}(\tilde{S})(\theta\{f_i/\xi\})(p)(\sigma) = p(M(S)(\gamma\{m_i/X\})(\sigma))$. Now this holds by the induction hypothesis on S (the complexity of S is less than the complexity of $\mu X[S]$), provided that the pair $<\gamma\{m_i/X\}, \theta\{f_i/\xi\}>$ is

consistent, i.e., that for all $\langle Y,\eta\rangle$ such that $\tilde{Y} = \eta$, and all p and σ, we have: $\theta\{f_i/\xi\}(\eta)(p)(\sigma) = p(\gamma\{m_i/X\}(Y)(\sigma))$. We distinguish two cases:

(ii.1) $\eta = \xi$ (hence $Y = X$). Then we have to show:
$\theta\{f_i/\xi\}(\xi)(p)(\sigma) = p(\gamma\{m_i/X\}(X)(\sigma))$, or
$f_i(p)(\sigma) = p(m_i(\sigma))$, which is nothing but (*).

(ii.2) $\eta \neq \xi$ (hence, $Y \neq X$). Then we have to show:
$\theta\{f_i/\xi\}(\eta)(p)(\sigma) = p(\gamma\{m_i/X\}(Y)(\sigma))$, or
$\theta(\eta)(p)(\sigma) = p(\gamma(Y)(\sigma))$, which follows from the consistency of $\langle\gamma,\theta\rangle$. □

9. WEAKEST PRECONDITIONS FOR RECURSIVE PROGRAMS

(D71) $wp(m,p) = p \circ m$.

Note that $(p\circ m)(\sigma)$ holds iff $p(\sigma')$ holds for each $\sigma' \in m(\sigma)$, or, equivalently, iff for each $\sigma' \in m(\sigma)$, both $\sigma' \neq \perp$ and $p(\sigma') = tt$ are satisfied. Hence, we have indeed the equivalence of $(p\circ m)(\sigma)$ and $wp(m,p)$ according to the usual definition of weakest precondition.

For $S \in Stat$ and $\phi \in Prtr$ which have no free occurrences of procedure variables or predicate transformer variables, we may as well omit the γ- and θ-arguments in the definitions of M and F. Using this, we obtain

(T3) For each $S \in Stat$ without free procedure variables, and each $p \in P$

$$F(\tilde{S})(p) = wp(M(S),p).$$

Proof. From (T2) and (D71). □

10. WEAKEST PRECONDITIONS FOR THE WHILE STATEMENT

In this section we extend the class of boolean expressions *Bexp* to the class of *assertions* *Assn*, including, in particular, for each statement S and assertion c, the construct S:c which is the *syntactic* counterpart of the weakest precondition. That is, we define its meaning $T(S:c)$ by: $T(S:c) = wp(M(S),T(c))$. Furthermore, we present the more or less well-known rules for expressing S:c through an induction on the complexity of S, in case it is an assignment statement, or made up from given statements through sequential composition, nondeterministic choice, if-then-else-fi, or the while statement. We do not know how a similar rule for a general recursive procedure would look like. In other words, we do not know how to reduce *syntactically* $\mu X[S]:c$ to a construct involving S:c (or, possibly, S:c' for suitably defined c'), and we conjecture that such reduction is impossible. (On the other hand, for so-

called *iterative* programs, i.e., recursive programs which are subjected to the restriction that they are derivable as equivalents of flow diagrams using McCarthy's well-known construction, we can easily generalize the result for the while statement, as illustrated by an example.)

Our results for the while statement are to some extent reformulations of, e.g., De Bakker & De Roever (1973) or Dijkstra (1975). We close this section with a comment on a recent theorem involving weakest preconditions for the while statement by Basu and Yeh (1975), to the effect that this theorem is incorrect.

		Elements
(D72) $Avar = \{\zeta_1,\zeta_2,\ldots\}$	assertion variables	$\zeta,\ldots$
(D73) $Assn$	assertions	c,d,...
c::= $b\|\zeta\|c_1 \wedge c_2\|$ <u>if</u> b <u>then</u> c_1 <u>else</u> c_2 <u>fi</u>$\|S:c\|\mu\zeta[c]$		
(D74) $\Delta = (Avar \to P) \cup \Gamma$		$\delta,\ldots$

(D75) $\delta|\Gamma$ denotes δ restricted to Γ

(D76) T: $Assn \to (\Delta \to P)$

T is extended from boolean expressions to assertions as follows:

(D77) $T(b)(\delta) = T(b)$

(D78) $T(\zeta)(\delta) = \delta(\zeta)$

(D79) $T(c_1 \wedge c_2)(\delta) = \lambda\sigma\cdot T(c_1)(\delta)(\sigma) \wedge T(c_2)(\delta)(\sigma)$

(D80) $T($<u>if</u> b <u>then</u> c_1 <u>else</u> c_2 <u>fi</u>$)(\delta) =$
$\lambda\sigma\cdot$<u>if</u> $T(b)(\sigma)$ <u>then</u> $T(c_1)(\delta)(\sigma)$ <u>else</u> $T(c_2)(\delta)(\sigma)$ <u>fi</u>

(D81) $T(S:c)(\delta) = wp(M(S)(\delta|\Gamma),T(c)(\delta))$

(D82) $T(\mu\zeta[c])(\delta) = \mu[\lambda p\cdot T(c)(\delta\{p/\zeta\})]$
(to be justified similarly to (T1)).

We now introduce notations for *validity* of assertions and of equivalences between them, for substitution in boolean expressions, and for the while statement.

(D83) An assertion c is *valid* - written as $\models$ c - iff, for all δ and all $\sigma \neq \perp$, $T(c)(\delta)(\sigma) = tt$ holds. Similarly, $\models c_1 = c_2$ holds whenever, for all δ and σ, $T(c_1)(\delta)(\sigma) = T(c_2)(\delta)(\sigma)$

(D84) b[s/x] denotes the result of replacing all occurrences of x in b by s (formal definition omitted)

(D85) <u>dummy</u> $\stackrel{df}{\equiv}$ (x:=x)

(D86) <u>while</u> b <u>do</u> S <u>od</u> $\stackrel{df}{\equiv}$ μX[<u>if</u> b <u>then</u> S;X <u>else</u> <u>dummy</u> <u>fi</u>]

The following lemma's show how to express S:c in terms of the components of S.

(L17) $\models$ (x:=s):b = b[s/x]
(it is left to the reader to verify that this equivalence does not hold for arbitrary c $\in$ *Assn*)

(L18) $\models (S_1;S_2):c = S_1:(S_2:c)$.

(L19) $\models (S_1 \cup S_2):c = (S_1:c) \wedge (S_2:c)$

(L20) $\models$ $\underline{\text{if}}$ b $\underline{\text{then}}$ S_1 $\underline{\text{else}}$ S_2 $\underline{\text{fi}}$: c =
$\underline{\text{if}}$ b $\underline{\text{then}}$ S_1 : c $\underline{\text{else}}$ S_2 : c $\underline{\text{fi}}$

(L21) $\models$ $\underline{\text{while}}$ b $\underline{\text{do}}$ S $\underline{\text{od}}$: c =
$\mu\zeta[\underline{\text{if}}$ b $\underline{\text{then}}$ S : ζ $\underline{\text{else}}$ c $\underline{\text{fi}}]$

Remarks.

1. As a consequence of (L21) we have - assuming no free procedure - or assertion variables in S or c -
$T(\underline{\text{while}}$ b $\underline{\text{do}}$ S $\underline{\text{od}}$: c) = $\bigsqcup_i p_i$, where, for each σ,
$p_0(\sigma) = ff$, $p_{i+1}(\sigma) = \begin{cases} \hat{wp}(M(S),p_i)(\sigma), & \text{if } T(b)(\sigma) = tt \\ T(c)(\sigma), & \text{if } T(b)(\sigma) = ff \end{cases}$
2. As an illustration of how (L21) might be generalized to iterative programs, we have, e.g. (using one of Böhm and Jacopini's (1966) non-while-statement-reducible flow diagrams):

$\mu X[\underline{\text{if}}$ b_1 $\underline{\text{then}}$ S_1;$\underline{\text{if}}$ b_2 then S_2;X $\underline{\text{else}}$ $\underline{\text{dummy}}$ $\underline{\text{fi}}$ $\underline{\text{else}}$ $\underline{\text{dummy}}$ $\underline{\text{fi}}]$:c

$\models$ =

$\mu\zeta[\underline{\text{if}}$ b_1 $\underline{\text{then}}$ S_1:$\underline{\text{if}}$ b_2 $\underline{\text{then}}$ S_2:ζ $\underline{\text{else}}$ c $\underline{\text{fi}}$ $\underline{\text{else}}$ c $\underline{\text{fi}}]$

We now introduce the notation to state our comment on Basu & Yeh's result (theorem 9 of (Basu & Yeh (1975)). Following them, we restrict ourselves (from now on) to deterministic programs.

(D87) $\frac{c_1}{c_2}$ holds iff validity of c_1 implies validity of c_2. Similarly, we define when $\frac{c_1=c_2}{c}$ or $\frac{c}{c_1=c_2}$ hold.

(D88) $\{c_1\}S\{c_2\} \overset{df.}{\equiv} (c_1 \wedge (S:\underline{\text{true}})) \supset (S:c_2)$ (this embodies (for deterministic S) the usual notion of *partial correctness* of S with respect to precondition c_1 and postcondition c_2)

We have

$\underline{\text{while}}$ b $\underline{\text{do}}$ S $\underline{\text{od}}$: c

(C3) $\models$ =

$\underline{\text{if}}$ b $\underline{\text{then}}$ S : ($\underline{\text{while}}$ b $\underline{\text{do}}$ S $\underline{\text{od}}$:c) $\underline{\text{else}}$ c $\underline{\text{fi}}$

(C4) $$\frac{d = \underline{\text{if}}\ b\ \underline{\text{then}}\ S{:}d\ \underline{\text{else}}\ c\ \underline{\text{fi}}}{(\underline{\text{while}}\ b\ \underline{\text{do}}\ S\ \underline{\text{od}}\ :\ c) \supset d}\ .$$

(C3) and (C4) together express the least-fixed-point property of $\underline{\text{while}}$ b $\underline{\text{do}}$ S $\underline{\text{od}}$:c. and are therefore, when taken together, equivalent with (L21).

According to Basu & Yeh, we also have

$$\frac{\{d \wedge b\}S\{d\} \wedge ((d \wedge \neg b) \supset c) \wedge ((\underline{\text{while}}\ b\ \underline{\text{do}}\ S\ \underline{\text{od}} : c) \supset d)}{d = \underline{\text{if}}\ b\ \underline{\text{then}}\ S{:}d\ \underline{\text{else}}\ c\ \underline{\text{fi}}}$$

However, taking $b \equiv \underline{\text{true}}$, and using the fact that $\models \underline{\text{while}}\ \underline{\text{true}}\ \underline{\text{do}}\ S\ \underline{\text{od}} : c = \underline{\text{false}}$ holds, we obtain as a special case of this

$$\frac{\{d\}S\{d\}}{d = S{:}d}$$

which is invalid: Take, e.g., $S \equiv (x{:=}x+1)$, and $d \equiv (x>0)$. Then, though $\models \{d\}S\{d\}$ holds, it is not true that $\models d = S{:}d$ also holds, since $S{:}d$ is equivalent with $x \geq 0$ (and not with $x > 0$).

REFERENCES

De Bakker, J.W. (1976), Least fixed points revisited, Theoretical Computer Science 2, pp. 155-181.

De Bakker, J.W. (1976), Semantics and termination of nondeterministic recursive programs, in Proc. 3d Coll. Automata, Languages and Programming (S. Michaelson & R. Milner, eds.), pp.435-477, Edinburgh University Press.

De Bakker, J.W. (to appear), Semantics and the foundations of program proving, IFIP Congress 77.

De Bakker, J.W. and W.P. de Roever (1973), A calculus for recursive program schemes, in Proc. 1st Coll. Automata, Languages and Programming (M. Nivat, ed.), pp.167-196, North-Holland.

Basu, S.K. and R.T. Yeh (1975), Strong verification of programs, IEEE Transactions on Software Engineering 1, pp.339-346.

Böhm, C. and G. Jacopini (1966), Flow diagrams, Turing machines, and languages with only two formation rules, Comm. ACM 9, pp.366-372.

Dijkstra, E.W. (1975), Guarded commands, nondeterminacy and formal derivation of programs, Comm. ACM 18, pp.453-457.

Egli, M. (1975), A mathematical model for nondeterministic computations, Technological University, Zürich.

Milne, R. and C. Strachey (1976), A Theory of Programming Language Semantics, Chapman & Hall.

De Roever, W.P. (1976), Dijkstra's predicate transformer, nondeterminism, recursion and termination, in Proc. 5th Symp. Math. Foundations of Computer Science (A. Mazurkiewicz, ed.), pp.472-481, Lecture Notes in Computer Science 45, Springer.

DISCUSSION

David Harel: The notion of "total correctness" that you have based your analysis on cannot be given without an operational basis. A non-deterministic program has, for a given starting state, a computation tree with three kinds of paths: "good" paths leading to a final state, paths leading to false or undefined nodes ("failures"), and infinite paths ("divergencies"). There are four ways of traversing these trees:

1. Depth first, where one stops upon reaching a final (good or bad) node.

2. Depth first, with backtracking upon reaching a failing node.

3. Breadth first, stopping when encountering any final node.

4. Breadth first, with ignoring failure nodes.

These four methods have associated with them four different concepts of total correctness, corresponding to the four combinations of allowing or disallowing failures or divergencies.

Your notion of weakest preconditions (and by the same token also Dijkstra's) is presupposing method (1), where the notion of total correctness involves outlawing both divergencies and failures. Some of the properties of wp mentioned by deBakker do not hold for the other notions of total correctness. wp is too complicated a concept to be taken as the basis for defining the semantics of languages. The more primitive concepts such as "there exists a path to a final state", "there are no divergencies", etc., should be taken as basics, and different "weakest preconditions" can then be constructed from them. The semantics of the programs themselves, should be defined first, independently of any specific notion of execution.

deBakker: I basically agree with all that was said, I mention that other concepts can also be interesting, and should be dealt with too.

Harel: I would like to stress the fact that those "other" concepts should be the basic primitive concepts, from which others like wp can be built. Then one can adopt a method of execution, and with it a notion of total correctness.

Albert Meyer: I don't understand what causes the technical complications in the result giving the equivalence of programs as relations and programs as predicate transformers. You seem to be saying only that the relation that a program denotes can be defined in terms of its inverse, i.e. the function from the final set of states to the initial state; so what is so interesting about a theorem which states that obvious fact? I seem to be missing some subtlety in the definitions which makes the result nontrivial. Can you tell me what the difficulty is?

deBakker: The ingenuity was used in the introduction of the Egli-Milner ordering, which was only invented two years ago. Before that ordering existed the things could not be worked out.

Meyer: Have you looked at questions like that of "elementary equivalence"? Questions which would classify equivalence of programs according to whether postconditions of a certain type have weakest preconditions of the same type.

deBakker: No, I have not.

Maarten van Emden: What states enter the weakest precondition? Does the weakest precondition have states in it which lead to infinite computations?

deBakker: That was discussed previously with Harel and I agree that our weakest preconditions include only states which do not lead to infinite computations and do not lead to failures.

Axel van Lamsweerde: Does the proof of your theorem use the restriction which outlaws unbounded nondeterminism?

deBakker: Yes, in the sense that the language I am using outlaws unbounded nondeterminism.

Dijkstra: I would like to remark that you do not loose anything by restricting yourself to bounded nondeterminism because nonbounded nondeterminism would result in an execution which would require an infinite amount of energy.