CHAPTER IV

MONTAGUE GRAMMAR AND PROGRAMMING LANGUAGES

ABSTRACT

The present chapter starts with an introduction to the semantics of programming languages. The semantics of the assignment statement is considered in detail, and the traditional approaches which use predicate transformers are shown to give rise to problems. A solution is presented according to the algebraic framework defined in the first chapters of this book; it uses an extension of intensional logic.
1. ASSIGNMENT STATEMENTS

1.1. Introduction

Programs are pieces of text, written in some programming language. These languages are designed for the special purpose of instructing computers. They also are used in communication among human beings for telling them how to instruct computers or for communicating algorithms which are not intended for computer execution. So for programming languages we are in the same situation as for natural languages. We have a syntax and we have intended meanings, and we wish to relate these two aspects in a systematic way. Since we are in the same situation, we may apply the same framework. In this chapter we will do so for a certain fragment of the programming language ALGOL 68.

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

It appears that assignment statements exhibit the same phenomena as intensional operators in natural languages. A certain position in the context of an assignment statement is transparent (certain substitutions for names are allowed), whereas another position is opaque (such substitutions are not allowed). The traditional ways of treating the semantics of programming languages do not provide tools for dealing with intensional phenomena. A correct treatment of simple cases of the assignment statement can be given, but for the more complex cases the traditional approaches fail. I will demonstrate that the treatment of intensional operators in natural language, as given in the previous chapters, may also be applied to programming languages, and that in this way a formalized semantics of
assignment statements can be given which deals correctly with the more complex cases as well. Hence we will use the same logic: intensional logic (see chapter 3). The idea to use this logic goes back to JANSSEN & Van EMDE BOAS (1977a,b). We will however, not only use the same logic, but also the same compositional, algebraic framework. In chapter 1 the background of this framework was discussed, and in chapter 2 it was defined formally and compared with the algebraic approach of Adj. For a bibliography of universal algebraic and logical approaches in computer science see ANDREKA & NEMETI 1969. The first sections of the present chapter are a revision of JANSSEN & Van EMDE BOAS (1981).

1.2. Simple assignments

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

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

\[
\begin{align*}
x &\rightarrow 1 \\
y &\rightarrow 2 \\
w &\rightarrow 4 \\
x &\rightarrow 7 \\
y &\rightarrow 7 \\
w &\rightarrow 4 \\
w &\rightarrow 5
\end{align*}
\]

**Figure la**  **Figure 1b**  **Figure 1c**  **Figure 1d**

Initial Situation  After \( x := 7 \)  After \( y := 7 \)  After \( w := w + 1 \)
Now the necessary preparations are made for demonstrating the relation with natural language phenomena. Suppose that we are in a situation where the identifiers $x$ and $y$ are both associated with value 7. Consider now the assignment

(1) $x := y + 1$.

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

(2) $x := x + 1$.

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

(3) $x := 7 + 1$.

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

(4) $y := y + 1$.

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

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

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

Consider now the following sentence:

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

The meaning of (6) would not be essentially changed if we replaced the commissioner by another identification of the same person; a thus changed sentence would be true in the same situations as the original sentence. But consider now (7).
(7) The commissioner is looking for the dean.

Changing (6) into (7) does make a difference: it is conceivable that the commissioner affirms (6) and simultaneously denies (7) because of the fact that he has not been informed that (5) recently has become a truth. Sentence (7) is true in other situations than sentence (5). Hence they have a different meaning. In the terminology for substitution phenomena, the subject position of is looking for is called (referentially) transparent, and its object position (referentially) opaque or intensional position. Because of the close analogy, we will use the same terminology for programming languages, and call the right hand side of the assignment 'transparent', and its left hand side 'opaque' or 'intensional'.

The observation concerning substitutions in assignments statements, as considered above, is not original. It is, for instance, described in TENNENT 1976 and STOD 1977 (where the term 'transparent' is used) and in PRATT 1976 (who used both 'transparent' and 'opaque'). The semantic treatments of these phenomena which have been proposed, are, however, far from ideal, and in fact not suitable for assignments which are less simple than the ones above. The authors just mentioned, like many others, avoid these difficulties by considering a language without the more complex constructions.

1.3. Other assignments

Above we only considered assignments involving cells which contain an integer as value. In this section I will describe two other situations: cells containing an identifier as value (pointers) and rows of cells (arrays).

Some programming languages also allow for handling cells which contain a variable (identifier) as value (e.g. the languages Pascal and Algol-68). Names of such cells are called pointer identifiers or equivalently pointer variables, shortly pointers. The situation that pointer \( p \) has the identifier \( x \) as its value, is shown in figure 2a. In this situation, \( p \) is indirectly related to the value of \( x \), i.e. \( 7 \). The assignment \( p := \omega \) has the effect that the value stored in \( p \)'s cell becomes \( \omega \) (figure 2b). Thus \( p \) is indirectly related to the value of \( \omega \): the integer \( 5 \). When next the assignment \( \omega := 6 \) is executed, the integer value indirectly associated with \( p \) becomes \( 6 \) (figure 2c). So an assignment can have consequences for pointers which are not mentioned in the assignment statement itself: the value of the variable associated with the pointer may change.
In a real computer, a cell does not contain an integer or a variable, but rather a code for an integer or a code for a variable. For most real computers it is not possible to derive from the contents of a cell, whether it should be interpreted as an integer code or a variable code. In order to prevent the unintended use of an integer code for a variable code, or vice versa, some programming languages (e.g. Pascal) require for each identifier a specification of the kind of values to be stored in the corresponding cells. The syntax of such a programming language then prevents unintended use of an integer code for an identifier code (etc.) by permitting only programs in which each identifier is used for a single kind of value. Other languages leave it to the discretion of the programmer whether to use an identifier for only one kind of value (e.g. Snobol-4). Our examples are from a language of the former type: ALGOL 68.

The programming language ALGOL 68 also allows for higher order pointers, such as pointers to pointers to variables for integer values. They are related to cells which contain as value (the code of) a pointer of the kind described above. These higher order pointers will be treated analogously to the pointers to integer identifiers.

Several programming languages have names for rows of cells (arrays of cells). Names of such rows are called array identifiers, or equivalently array variables. An individual cell can be indicated by attaching a subscript to the array identifier. The element of an array \( a \) associated with subscript \( i \) is indicated by \( a[i] \). The cells of an array contain values of a certain kind: the cells of an integer array contain integers (see figure 3a), and the cells of an array of pointers contain pointers. The execution of the assignment \( a[2] := 2 \) has the effect that in the cell indicated by \( a[2] \) the value 2 is stored (see figure 3b). The subscript may be a complex integer expression. The effect of the assignment \( a[a[1]] := 2 \) is that the value in \( a[1] \) is determined, it is checked whether the value obtained (i.e. 1) is an acceptable index for the array and the assignment \( a[1] := 2 \).
is performed (figure 3c). In the sequel I, will assume that all integers are acceptable indices for subscripts for an array, i.e. that all arrays are of infinite length (of course an unrealistic assumption; but I am interested in formalizing other aspects of arrays). Other kinds of assignment which involve arrays are in the fragment (e.g. the assignment of the whole array in a single action), but I will deal primarily with assignments of the form just discussed.

\[
\begin{align*}
\text{Initial Situation} & \quad \text{After } a[3] := 2 & \quad \text{After } a[a[1]] := 2 \\
\{ a[1] \rightarrow 1 \} & \quad \{ a[2] \rightarrow 2 \} & \quad \{ a[3] \rightarrow 3 \} \\
\{ a[1] \rightarrow 1 \} & \quad \{ a[2] \rightarrow 2 \} & \quad \{ a[3] \rightarrow 3 \} \\
\{ a[1] \rightarrow 1 \} & \quad \{ a[2] \rightarrow 2 \} & \quad \{ a[3] \rightarrow 3 \} \\
\text{Figure 3a} & \quad \text{Figure 3b} & \quad \text{Figure 3c}
\end{align*}
\]

2. SEMANTICS OF PROGRAMS

2.1. Why?

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

\[
(8) \quad x_{1,2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}.
\]

The program reads as follows:

1. \text{begin real } a, b, c, \text{disc, } d, x1, x2;
2. \text{read } ((a,b,c));
3. \text{disc := } b*b - 4*a*c;
4. \text{d := sqrt (disc)};
5. \text{x1 := -b + d; x1 := x1/(2*a)};
6. \text{x2 := -b - d; x2 := x2/(2*a)};
7. \text{print } ((a,b,c,x1,x2,newline))
8. \text{end.}

The first line of the program says that the identifiers mentioned there, will only be used as names of locations containing real numbers as values (e.g. 3.14160). The second and seventh line illustrate that the computer
may obtain data from outside (input) and communicate results to the outside
world (output). The program also shows that the mathematical formula looks
much more compact than the program, but that this compactness is made
possible by the use of some conventions which have to be made explicit for
the computer. For example, in the program we must write $\star a \star a$ for $\star$ times $a$
times $a$, while in the formula $\star a \star a$ suffices. In the formula we use two di-
mensional features, which are eliminated in the program ($\sqrt{..}$) instead
of $\sqrt{..}$). This linear character is necessitated by the fact that programs
have to be communicated by way of a sequential channel; for example, the
wire connecting the computer with a card reader. The symbol $\text{real}$ indicates
that the identifiers mentioned may only be associated with real values,
and the symbols $\text{begin}$ and $\text{end}$ indicate the begin and the end of the program.

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

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

- **equivalence**: Two different programs may yield the same results in all cir-
cumstances. For example, in the program under discussion we may interchange
the order of the computation of $x1$ and $x2$, but we cannot compute $\star$ before
we compute $\text{disc}$.

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

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

2.2. How?

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

For about 15 years so-called denotational semantics have been provided for programming languages (see e.g. TENNENT 1576, STOY 1977, De BAKKER 1980) of a program is given as a mathematical object in a model; usually some function which describes the input-output behaviour of the program. By abstracting from the intermediate stages of the computation, the model has far less resemblance to a real computer than the abstract machines used in operational semantics. The programs are not considered so much to be transforming values into values, but rather as transforming the entire initial state of a computer into some final state. In this approach, states are highly complex descriptions of all information present in the computer.

Mostly, we are not interested in all aspects of a computer state, but only in a small part (for instance the values of the input and output variables). This leads to a third approach to semantics, which uses so-called predicate transformers (FLOYD 1967, HOARE 1969, DIJKSTRA 1974, 1976 and DE BAKKER 1980). A (state) predicate is a proposition about states. So a predicate specifies a set of states: all states for which the proposition holds true. We need to correlate propositions about the state before the execution of the program (preconditions) with propositions about the state afterwards (postconditions). This is the approach to semantics that we will
follow in the sequel. Usually one distinguishes approaches which associate preconditions and postconditions, but do not consider termination of the execution of the program, and approaches which consider termination as well. The former approaches are said to deal with *partial correctness*, and the latter with *total correctness*. Since all programs we will discuss are terminating programs, the distinction is for our fragment not relevant and will not be mentioned any further.

As an example we consider the program from Section 2.1. An initial state may be described by specifying that on the input channel three numbers \( a, b \) and \( c \) are present such that \( a \neq 0 \) and \( b^2 - 4ac \geq 0 \). The execution of the program will lead such a state to a state where \( x1 \) and \( x2 \) contain the solutions to the equation \( ax^2 + bx + c = 0 \). Conversely, we observe that, if one wants the program to stop in a state where \( x1 \) and \( x2 \) represent the solutions of the equation \( ax^2 + bx + c = 0 \), it suffices to require that the coefficients \( a, b \) and \( c \) are present on the input channel (in this order!) before the execution of the program, and that moreover \( a \neq 0 \) and \( b^2 - 4ac \geq 0 \). In the semantics we will restrict our attention to the real computation, and therefore consider a reduced version of the program from which the input and output instructions and the specifications of the identifiers such as *real* are removed. Let us call this reduced program ‘prog’.

In presenting the relation between predicates and programs, we follow a notational convention due to HOARE 1969. Let \( \pi \) be a program, and \( \phi \) and \( \psi \) predicates expressing properties of states. Then \( \{ \phi \} \pi \{ \psi \} \) means that if we execute \( \pi \) starting in a state where \( \phi \) holds true, and the execution of the program terminates, then predicate \( \psi \) holds in the resulting state. Our observations concerning the program are now expressed by:

\[
\{ a \neq 0 \land (b^2 - 4ac) \geq 0 \} \text{prog } \{ a(x1)^2 + b(x1) + c = 0 \land \\
\quad a(x2)^2 + b(x2) + c = 0 \land \forall z[ax^2 + bx + c = 0 \rightarrow z = x1 \lor z = x2] \}.
\]

There are two variants of predicate transformer semantics. The aim of the first variant, the forward approach or (Floyd-approach) can be described as follows. For any program \( \pi \), find, according to the structure of \( \pi \), a predicate transformer which for any state predicate \( \phi \) yields a state predicate \( \psi \), such that if \( \phi \) holds before the execution of \( \pi \), then \( \psi \) gives all information about the final state which can be concluded from \( \phi \) and \( \pi \). Such a predicate \( \psi \) is called a strongest postcondition with respect to \( \phi \) and \( \pi \).
Mathematically a strongest postcondition $sp$ (with respect to $\phi$ and $\pi$) is defined by

(I) $\{\phi\} \trianglerightsp \{sp\}$ and

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

Suppose that we have two predicates $sp_1$ and $sp_2$, both satisfying (I) and (II). Then they are equivalent. From (I) follows that $\{\phi\} \trianglerightsp \{sp_1\}$ and $\{\phi\} \trianglerightsp \{sp_2\}$. Then from (II) follows that $sp_1$ implies $sp_2$ and vice versa.

Since all strongest postcondition with respect to $\phi$ and $\pi$, are equivalent, we may speak about the strongest postcondition with respect to $\phi$ and $\pi$. For this the notation $sp(\pi, \phi)$ is used.

Instead of this approach, one frequently follows an approach which reverses the process: the backward-approach or Hoare-approach. For a program $\pi$ and a predicate $\psi$ one wants to find the weakest predicate which still ensures that, after execution of $\pi$, predicate $\psi$ holds. Such a predicate is called a weakest precondition. Mathematically a weakest precondition $wp$ (with respect to $\pi$ and $\psi$) is defined by

I $\{wp\} \trianglerightsp \{\psi\}$

II If $\{\eta\} \trianglerightsp \{\psi\}$ then from $\eta$ we can conclude $wp$.

Analogously to the proof for postconditions, it can be shown that all weakest preconditions are equivalent. Therefore we may speak about the weakest precondition with respect to $\pi$ and $\phi$. For this the notation $wp(\pi, \phi)$ is used (see DIJKSTRA 1974, 1976 for more on this approach).

Above, I used the phrase 'based upon the structure of $\pi$'. This was required since it would be useless to have a semantics which attaches to each program and predicate a strongest postcondition in an ad-hoc way, in particular because there are infinitely many programs. One has to use the fact that programs are formed in a structured way according to the syntax of the programming language, and according to our framework, we aim at obtaining these predicate transformers by means of a method which employs this structure.

3. PREDICATE TRANSFORMERS

3.1. Floyd's forward predicate transformer

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

\[ \{ \phi \} v := \delta \ (\exists z)[[z/v]\phi \land v = [z/v]\delta] \].

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

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

1) assignment: \( x := 1 \); precondition: \( x = 0 \)
   obtained postcondition:
   \( \exists z[[x/z](x=0) \land z = [x/z]1] \), i.e. \( \exists z[x=0 \land x=1] \), which is equivalent to \( x = 1 \).

2) assignment: \( x := x + 1 \); precondition: \( x > 0 \)
   obtained postcondition:
   \( \exists z[[x/z](x>0) \land z = [x/z](x+1)] \), i.e. \( \exists z[x>0 \land x=x+1] \), which is equivalent to \( x > 1 \).
   Obtained postcondition:
   \[ \exists z ([z/a[1]](a[1] = a[2]) \land a[1] = [z/a[1]](a[1]+1)) \], i.e.
   \[ \exists z : a[2] \land a[1] = z+1 \], which is equivalent to \( a[1] = a[2] + 1 \).

3.2. Hoare's backward predicate transformer

Below Hoare's description will be given of the weakest precondition for the assignment statement. First I will give some heuristics. Suppose we want \( x = 4 \) to hold after the execution of \( x := y + 1 \). Then it has to be the case that before the execution of the assignment, \( y + 1 = 4 \) holds. More generally, every statement about \( x \) holding after the assignment has to be true about \( y + 1 \) before its execution. This observation is described in the following rule for the backward predicate transformer (HOARE 1969)

(11) \[ ([\delta/u]\psi) \nu := \delta \{ \psi \} \).

Some examples illustrate how the rule works in practice.

1) Assignment: \( x := 1 \); postcondition: \( x = 1 \).
   Obtained precondition:
   \[ [1/x](x=1) \], i.e. \( 1 = 1 \), or \textit{true}.
   This result says that for all initial states \( x = 1 \) holds after the execution of the assignment. If the postcondition had been \( x = 2 \), the obtained precondition would have been \( 1 = 2 \) or \textit{false}, thus formalizing that for no initial state does \( x = 2 \) hold after execution of \( x := 1 \).

2) Assignment: \( x := x + 1 \); postcondition \( x > 1 \).
   Obtained precondition:
   \[ [x+1/x](x>1) \], i.e. \( x + 1 > 1 \) which is equivalent to \( x > 0 \).

   Obtained precondition:

3.3. Problems with Floyd's rule

Since 1974 it has been noticed by several authors that the assignment rules of Floyd and Hoare lead to incorrect results when applied to cases where the identifier is not directly associated with a cell storing an integer value. Examples are given in Van EMDE BOAS (1974), (thesis 13),
De Bakker (1976), Gries (1977), Janssen & Van Emde Boas (1977a,b). The examples concern assignments involving an identifier of an integer array, or a pointer to an integer identifier. In this section I will consider only examples concerning Floyd’s rule.

An example concerning assignment to a subscripted array identifier is

(12) \text{a}[\text{i}] := 2.

Suppose that the assertion which holds before the execution of the assignment is

(13) \text{a}[\text{i}] = 1 \land \text{a}[\text{j}] = 1.

Then Floyd’s rule implies that after the execution of the assignment holds

(14) \exists\text{z}[\text{z}/\text{a}[\text{i}]](\text{a}[\text{i}] = 1 \land \text{a}[\text{j}] = 1 \land \text{a}[\text{i}] = 2)

i.e.

(15) \exists\text{z}[\text{z}/\text{a}[\text{i}]] = 1 \land \text{a}[\text{i}] = 2]

which is equivalent to

(16) \text{a}[\text{i}] = 2 \land \text{a}[\text{j}] = 1.

This formula is a contradiction, whereas the assignment is a correctly terminating action. Compare this result with the situations in figure 3, where this assignment is performed in a situation satisfying the given precondition. Then it is clear that the postcondition should be

(17) \text{a}[\text{i}] = 2 \land \text{a}[\text{j}] = 1.

It turns out that problems also arise in the case of pointers (Janssen & Van Emde Boas 1977a). An example is the following program consisting of three consecutive assignment statements. The identifier \(p\) is a pointer and \(x\) an integer variable.

(18) \(x := \delta; p := x; x := \delta.\)

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

(19) \exists\text{z}[\text{z}/\text{p}](x = \delta) \land p = [z/p]x]
i.e.

(20) \( \exists x \left[ x = 5 \land p = x \right] \)

which is equivalent to

(21) \( x = 5 \land p = x \).

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

(22) \( \exists x \left[ [x/x] (x = 5 \land p = x) \land x = [x/x] \right] \)

i.e.

(23) \( \exists x \left[ x = 5 \land p = x \land x = \emptyset \right] \).

This formula says that the integer value related with \( p \) equals 5. But as the reader may remember from the discussion in Section 2, the integer value related with \( p \) is changed as well (figure 2c).

3.4. **Predicate transformers as meanings**

Floyd's assignment rule is one rule from a collection of proof rules: for each construction of the programming language there is a rule which describes a relation between precondition and post condition. The meaning of a construction is defined in a completely different way. A computer-like model is defined, and the meaning of a statement (e.g. the assignment statement) is described as a certain state-transition function (a function from computer states to computer states). The proof rule corresponding to the construction can be used to prove properties of programs containing this construction. A prime example of this approach is De Bakker (1980).

It is, however, not precisely the approach that I will follow in this chapter.

In the discussion in section 2.2 I have mentioned arguments why predicate transformers are attractive from a semantic viewpoint, and why state-transition function are less attractive. I will give predicate transformers a central position in my treatment: the meaning of a program, and in particular of an assignment statement, will be defined by means of a predicate transformer.

In theory I could define the meaning of an assignment by any predicate transformer I would like. But then there is a great danger of loosing contact with the behaviour of computer programs in practice. Therefore I will
give a justification of my choice of the predicate transformers. This will be done by defining a state-transition function that resembles the usual state-transition semantics. Then it will be proven that the defined predicate transformers are correct and yield strongest postconditions (or weakest preconditions). In the light of this connection with practice, it is not surprising that there is a resemblance between Floyd's (Hoare's) predicate transformer and the one I will define. But the formal position of the predicate transformers is essentially different in this approach. Actually, I shall argue that Floyd's (Hoare's) predicate transformer cannot be used for our purposes. The problems with the standard formulation of the transformers are mentioned below; they are solvable by some modifications which will be discussed in the next section. The discussion will be restricted to the Floyd-approach; for the Hoare approach similar remarks apply.

In the Floyd-approach the predicate-transformation rule for the assignment is an axiom in a system of proof rules. It can be considered as an instruction how to change a given predicate into its strongest postcondition. In our approach an assignment statement has to be considered semantically as a predicate transformer. Hence it has to correspond with a single expression which is interpreted in the model as a predicate transformer. This requires that Floyd's rule has to be reformulated into such an expression. This can be done by means of a suitable $\lambda$-abstraction. The predicate transformer corresponding with assignment $x := \delta$ will look like (24).

$$\lambda z z.([\delta/x] \phi \land x = [\delta/x] z).$$

This expression is not quite correct because of an inconsistency in the types of $\phi$. The subexpression $[\delta/x] \phi$ is part of a conjunction. Therefore both $[\delta/x] \phi$ and $\phi$ have to denote a truth-value. But in the abstraction $\lambda \phi$ the $\phi$ is not intended as an abstraction over truth-values (there are only two of them), but as an abstraction over predicates (there are a lot of them). This means that the types of $\phi$ in (24) are not consistent, so it cannot be the predicate-transformer which we will use.

A second problem is the occurrence of the substitution operator in Floyd's rule (and in (24)). It is an operator which operates on strings of symbols. The operator does not belong to the language of logic and there is no semantic interpretation for it. Hence expressions containing the operator have no interpretation. To say it in the terminology of our framework: expressions like (24) are not a polynomial operator over the logic used. Remember that no logical language has the substitution operator
as one of its operators. Substitution belongs to the meta-language, and is
used there to indicate how an expression of the logic has to be changed in
order to obtain a certain other expression. Since proof rules and axioms
are, by their nature, rules concerning syntactic objects, there is no ob-
jection against a substitution operator occurring in a proof rule. But we
wish to use predicate transformers to determine meanings. If we would use
substitution operators in predicate-transformers, then our transformers
would be instructions for formula manipulation, and we would not do seman-
tics. The same observation is made by Tennent with respect to another rule.
He stated in a discussion (NEUHOLD 1978, p.69):

Substitution is purely syntactic, function modification semantic.

The third problem can be illustrated by considering the assignment
\( x := y + 1 \). The identifier \( x \) is used in the execution of the program in an
essentially different way than the identifier \( y \). The \( y \) is used to indicate
a certain value. The \( x \) is used as the name of a cell, and not to indicate
a value. This different use corresponds with the semantic difference: in
section 1.2 we observed that the left-hand side of the assignment statement
is referentially opaque, whereas the right-hand side is transparent. Floyd's
rule does not reflect these differences. The rule makes no clear distinction
between a name and the value associated with that name. In my opinion this
is the main source of the problems with Floyd's rule. Remember that all
problems we considered above, arose precisely in those situations where
there are several ways available for referring to a certain value in the
computer: one may use an identifier or a pointer to that identifier; one
may use an array identifier subscripted with an integer, or subscripted
with an compound expression referring to the same value.

In the field of semantics of natural languages an approach which iden-
tified name and object-referred-to was employed in the beginnings of this
century. Ryle epitomizes this feature of these theories in his name for
them: 'Fido'-Fido theories! The word 'Fido' means Fido, the dog, which is
its meaning (see STEINBERG & JAKOBOVITS 1971, p.7). The approach was
abandoned, because it turned out to be too simple for treating the less
elementary cases. In view of the analogy of the behaviour of names in na-
tural languages and in programming languages we observed in section 1, it
is not too surprising that Floyd's rule is not completely successful either.
4. SEMANTICAL CONSIDERATIONS

4.1. The model

In section 5 the syntax and semantics of a small fragment of a programming language will be presented; in section 7 a larger fragment will be dealt with. The treatment will fit the framework developed in the first chapter. So we will translate the programming language into some logical language, which is interpreted in some model. In the present section the semantical aspects (model, logic) will be discussed which are relevant for the treatment of the first fragment. In sections 6 and 7 this discussion will be continued.

In section 2.1 we observed that the assignment statement creates an intensional context. Therefore it is tempting to try to apply in the field of programming languages the notions developed for intensional phenomena in natural languages. The basic step for such an application is the transfer of the notion 'possible world' to the context of programming languages. It turns out that possible worlds can be interpreted as internal states of the computer. Since this is a rather concrete interpretation, I expect that the ontological objections which are sometimes raised against the use of possible world semantics for natural languages (e.g. Potts 1976), do not apply here. The idea to use a possible world semantics and some kind of modal logic can be found with several authors. An influencing article in this direction was Pratt 1976; for a survey, see Van Emde Boas 1978 or Pratt 1980.

An important set in the model is the set of possible worlds, which in the present context will be called set of states. This set will be introduced in the same way as possible worlds were introduced in the treatment of natural languages. It is just some non-empty set (denoted by ST). They are not further analysed; so we do not build explicitly in our semantic domains some abstract model of the computer. But this does not mean that every model for intensional logic is an acceptable candidate for the interpretation of programming languages. Below I will formulate some restrictions on these models, which determine a certain subclass, and these restrictions have, of course consequences for the set ST as well. In this indirect way certain properties of the computer are incorporated in the model. The formulation of the restrictions only concern the simple assignment statement, and they will be generalized in section 7.
An integer identifier is associated with some cell in the computer, and for each state we may ask which value is contained in this cell. The semantic property of an integer identifier we are interested in, is the function which relates a state with the value contained (in that state) in the cell corresponding to that identifier. So we wish to associate with an identifier a function from states to values, see chapter 1 for a discussion (the same idea can be found in ADJ 1977 or 1979). In order to obtain this effect, integer identifiers are translated into constants of type \(<s,e>\) (e.g. the identifiers \(x,y\) and \(w\) are translated into the constants \(x, y\) and \(w\) of type \(<s,e>\)). But something more can be said about their interpretation. The standard interpretation of constants of intensional logic allows that for a given constant we obtain for different states different functions from states to values as interpretation. But we assume that on the computers on which the programs are executed, the relation between an identifier and the corresponding cell is never changed, so that for all states the function associated with an identifier is the same. The interpretations of \(x, y\) and \(w\) have to be state independent (in chapter 5, section 2 a related situation will arise for natural language; one uses there for such constants the name 'rigid designators'). This requirement implies that not all models for intensional logic are acceptable as candidates for formalizing the meaning of programming languages. We are only interested in those models in which the following postulate holds.

4.1. Rigidness Postulate

Let \(c \in \text{CON}_{<s,e>}\) and \(v \in \text{VAR}_{<s,e>}\). Then the following formula holds:

\[ \exists v \forall c [c = v]. \]

4.1. END

The above argumentation in favour of the rigidness postulate is not completely compelling. For a fragment containing only simple assignment statements one might alternatively translate integer identifiers into constants of type \(e\) which are interpreted non-rigidly. In such an approach the constant relation between an identifier and a cell would not have been formalized. This aspect will, however, become essential if the fragment is extended with pointers. Although there are no essentially non-rigid constants in the fragment under consideration, it is also possible to consider
such constructs e.g. the integer identifier \( xory \) which denotes the same as the integer identifier \( x \) or the integer identifier \( y \), depending on which of both currently has the greatest integer value. The rigidness postulate guarantees that the interpretation of constants is state independent. Therefore we may replace the usual notation for their interpretation, being \( F(c)(s) \), by some notation not mentioning the current state. I will use \( V(c) \) as the notation for the interpretation of a constant with respect to an arbitrary state.

Two states which agree in the values of all identifiers should not be distinguishable, since on a real computer such states (should) behave alike. Two states only count as different if they are different with respect to the value of at least one identifier. This is expressed in the following postulate.

4.2. Distinctness Postulate

Let \( s, t \in ST \). If for all \( c \in CON \), \( V(c)(s) = V(c)(t) \), then \( s = t \).

4.2. END

The execution of an assignment modifies the state of the computer in a specific way: the value of a single identifier is changed, while the values of all other identifiers are kept intact. This property is expressed by the update postulate, which requires that the model to be rich enough to allow for such a change. The term 'update' should not be interpreted as stating that we change the model in some way; the model is required to have a structure allowing for such a transition of states.

4.3. Update Postulate

For all \( s \in ST \), \( c \in CON_{<S,e>} \), \( n \in \mathbb{N} \) there is a \( t \in ST \) such that

\[
V(c)(t) = n \\
V(c')(t) = V(c')(s) \text{ if } c' \neq c.
\]

4.3. END

The update postulate requires the existence of a certain new state, and the distinctness postulate guarantees the uniqueness of this new state.
I formulated the update postulate for constants of type <s,e> only, but in section 7 it will be generalized to constants of many other types as well. If the update postulate holds for a constant c and a value d, then the (unique) state required by the postulate is denoted <c→d>s.

Note that the postulates differ from the meaning postulates given for natural languages in the sense that they are formulated in the meta-language and not in intensional logic itself. This allowed us to use quantification over states and over constants in the formulation of the postulates.

One might wish to construct a model which satisfies these three postulates. It turns out that the easiest way is to give the states an internal structure. The rigidness postulate and the distinctness postulate say that we may take for elements of ST sets of functions from (translations of) identifiers to integers. The update postulate says that ST has to be a sufficiently large set. Let ID be the set of integer identifiers. Then we might take ST = ID. Another possibility (suggested by J. Zucker) is ST = {s ∈ ID | s(x) ≠ 0 for only finitely many x}. Sets of states with a completely different structure are, in principle, possible as well.

In the introduction I have said that the set of states (set of possible worlds) is just some set. This means that states are, in our approach, a primitive notion and that no internal structure is required for them. But the models just described correspond closely with the models known from the literature (e.g. the one defined by De BAKKER (1980, p.21)); for the larger fragment we will consider this correspondence is less obvious (see section 7). The difference between these two approaches is that here we started with requiring certain properties, whereas usually one starts defining a model. A consequence is that we are only allowed to use the properties we explicitly required, and that we are not allowed to use the accidental properties of a particular model. This is an advantage when a model has to be explicit about a certain aspect, whereas a theory is required to be neutral in this respect. An example could be the way of initialization of identifiers as discussed in De BAKKER (1980, p.218). He says about a certain kind of examples that it: ' [...] indicates an overspecification in our semantics [...] , it also leads to an incomplete proof theory'. He avoids the problem by eliminating them from his fragment. By means of the present approach such an overspecification could probably avoided.
4.2. The logic

We will use a possible-world semantics for dealing with phenomena of opaque and transparant contexts. Therefore it is tempting to use as logical language the same language as we used in the previous chapters: intensional logic. Since we deal with a programming language, some of the semantic phenomena will differ considerably from the ones we considered before. Intensional logic will be extended with some new operators which allow us to cope with these new phenomena.

The programs deal with numbers, and this induces some changes. The constants of type \( \mathbb{N} \) \((v, v', v'', \ldots)\) will be written in the form \(0, 1, 2, 3, \ldots\) and interpreted as the corresponding numbers. The logic is extended with operators on numbers: +, *, -, <, =. The symbols true and false abbreviate \(1 = 1\) and \(1 \neq 1\) respectively. The programming language has an if-then-else-fi construction (the fi plays the role of a closing bracket; it eliminates syntactic ambiguities). A related construction is introduced in the logic. Its syntax and semantics are as follows:

4.4. DEFINITION. For all \( \tau \in \text{Ty}, \alpha \in \text{ME}_\tau, \beta \in \text{ME}_\tau, \) and \( \gamma \in \text{ME}_\tau \) we have

\[
\text{if } \alpha \text{ then } \beta \text{ else } \gamma \text{ fi } \in \text{ME}_\tau.
\]

The interpretation is defined by:

\[
V_{s, g} \text{ if } \alpha \text{ then } \beta \text{ else } \gamma \text{ fi } = \begin{cases} V_{s, g}(\beta) & \text{if } V_{s, g}(\alpha) = 1 \\ V_{s, g}(\gamma) & \text{otherwise.} \end{cases}
\]

4.4. END

The update postulate and the distinctness postulate guarantee for \( n \in \mathbb{N} \) and \( c \in \text{CON} \) existence and uniqueness of a state \(<c^n>\). It is useful to have in the logic an operator which corresponds with the semantic operator \(<c^n>\). These operators, which I will call state switchers, are modal operators (since they change the state, i.e., world) with respect to which its argument is interpreted. The syntax and semantics of state switchers is defined as follows.

4.5. DEFINITION. For all \( \sigma, \tau \in \text{CAT}, \phi \in \text{ME}_\sigma, c \in \text{CON}_{<s, \tau>}, \alpha \in \text{ME}_\tau \) we have
$(a^c)^\psi \in ME^c.$

The interpretation is defined by:

$$V_{s,e}((a^c)^\psi) = \begin{cases} V_{<c^e>V_{s,e}^c(a)>s,e}^c(\psi) & \text{if } <c^e>V_{s,e}^c(a)>s \\
V_{s,e}^c(\psi) & \text{otherwise.} \end{cases}$$

Note that in the present stage of exposition, the 'defined' case only applies for $c \in \text{CON}_{s,e}^c$.

4.5. END

One might wonder why the state-switcher contains an extension operator, for only the constant $c$ and the expression $a$ are relevant for determining which state-switcher is intended. The reason is that state-switchers have many properties in common with the well-known substitution operators. The state-switcher determined by $c$ and $a$ behaves almost the same as the substitution operator $[a^c]$. This will be proven in section 4.3.

The meaning of a program will be defined as a predicate transformer. Since we will represent meanings in intensional logic, we have to find a representation of predicate transformers in intensional logic. Let us first consider state-predicates. These are properties of states. For some states the predicate holds, for others it does not hold, so a state predicate is a function $f: S \to \{0,1\}$. Since the interpretation of intensional logic is state-dependent, such a state predicate can be represented by means of an expression of type $t$.

A predicate transformer should, in the present approach, not be an operation on expressions, but a semantic function which relates state-predicates with state-predicates. So it should be a function $f: (S\to\{0,1\}) \to (S\to\{0,1\})$. This means that it is a function which yields a truth-value, and which takes two arguments: a state-predicate, and a state. Changing the order of the arguments does not change the function essentially. We may consider a state-predicate as a function which takes a state and a state-predicate, and yields a truth-value. Hence we may say that a predicate transformer is a function $f: S \to ((S\to\{0,1\})\to\{0,1\})$. This view is, in a certain sense, equivalent to the one we started with. A formula of type $\langle<s,t>,s\rangle$ has as its meaning such a function, hence formulas of type
$<s,t>,t>$ are suitable as representations of predicate transformers. Therefore programs and assignments can be translated into expressions of this type.

One might have expected that programs and assignments are translated into expressions of type $<s,t>,<s,t>$. This was the type of the translations of programs and assignments in JANSSEN & Van ENDE BOAS (1977a,b). The first argument for using the type $<s,t>,t>$ of theoretical nature. An expression of type $<s,t>,<s,t>$ has as its meaning a function $f : S \to (S\to L) \to (S\to L)$, and this is not a predicate transformer (although it is closely connected, and could be used for that purpose).

The second argument is of practical nature: the type of the present translation gives rise to less occurrences of the $\land$ and $\lor$ signs.

A consequence of the representations which we use for (state-)predicates and predicate transformers is the following. Suppose that program $\pi$ is translated into predicate transformer $\pi'$, and that this program is executed in a state which satisfies predicate $\phi$. Then in the resulting state the predicate denoted by $\pi'(\phi)$ holds; it is intended as the strongest condition with respect to program $\pi$ and predicate $\phi$ (i.e. $sp(\pi,\phi)$).

4.3. Theorems

The substitution theorem says that the state-switcher behaves almost the same as the ordinary substitution operator. The iteration theorem describes a property of the iteration of state-switchers.

4.6. SUBSTITUTION THEOREM. The following equalities hold with respect to all variable assignments and states.

1. $\{a/\cdot c\}c' = c'$ for all $c' \in \text{CON}$.
2. $\{a/\cdot c\}v = v$ for all $v \in \text{VAR}$.
3. $\{a/\cdot c\}(\phi \land \psi) = \{a/\cdot c\}\phi \land \{a/\cdot c\}\psi$
   analogously for $\lor, \rightarrow, \neg$, if-then-else-fi constructs.
4. $\{a/\cdot c\}(\exists x\phi) = \exists x\{a/\cdot c\}\phi$ if $x$ does not occur free in $a$
   analogously for $\forall x\phi$, $\lambda x\phi$.
5. $\{a/\cdot c\}(\delta(\gamma)) = [\{a/\cdot c\}\delta](\{a/\cdot c\}\gamma)$.
6. $\{a/\cdot c\}\gamma$ analogously for $\Box \delta$.
7. $\{a/\cdot c\}$ $c' = c$.
Consequence

The state switcher \(a^\lor c\) behaves as the substitution operator \([a^\lor c]s\), except if applied to \(\Box \beta, \Diamond \beta\) or \(\lor \beta\) (where \(\beta \neq c\)). The formulas \([a^\lor c]\Box \beta\) and \([a^\lor c]\Diamond \beta\) reduce to \(\Box \beta\) and \(\Diamond \beta\) respectively, whereas \((a^\lor c)^\lor \beta\) cannot be reduced any further.

**PROOF.** Let \(t\) be the state \(<c^\lor \forall s, g(a)>s\), so \(V_{s, g}([a^\lor c]t) = V_{t, g}\). The equalities hold because of the Rigidness Postulate.

1. \(V_{s, g}([a^\lor c]t) = V_{t, g}(c') = V_{s, g}(c')\).
2. \(V_{s, g}(a^\lor c) = V_{t, g}(\forall \theta) = g(\nu) = V_{s, g}(\nu)\).
3. \(V_{s, g}(a^\lor c)(t^\theta) = 1 \iff V_{t, g}(t^\theta) = 1 \iff V_{t, g}(\theta) = 1 \iff V_{s, g}(a^\lor c) = 1 \iff V_{s, g}(a^\lor c) \land (a^\lor c) = 1\).
   Analogously for the other connectives.
4. \(V_{s, g}(a^\lor c)(\exists x t) = 1 \iff V_{t, g}(\exists x t) = 1 \iff \exists x (s \sim g)\) such that \(V_{t, g}(\theta) = 1 \iff \exists x (s \sim g)\) such that \(V_{s, g}(a^\lor c) = 1 \iff V_{s, g}(\exists x (a^\lor c)\theta) = 1\).
5. \(V_{s, g}(a^\lor c)(\beta(\gamma)) = V_{t, g}(t^\beta(\gamma)) = V_{t, g}(\delta(t)) = V_{t, g}(\delta(t^\gamma)) = V_{s, g}(a^\lor c) \beta = V_{s, g}(a^\lor c) \gamma\).
6. \(V_{s, g}(a^\lor c)(\Box \beta) = V_{t, g}(t^\beta) = \forall t^{\beta} V_{t, g}(\beta) = V_{s, g}(a^\lor c)\).
7. \(V_{s, g}(a^\lor c)(c) = V_{t, g}(c^\lor c) = V(c) (\forall c^\lor V_{s, g}(a) > s) = V_{s, g}(a)\).

4.7. **ITERATION THEOREM.**
\([a^\lor c]([a^\lor c]s) = ([a^\lor c]a^\lor c)(s)\).

**PROOF.** Note that also here the state switcher behaves as a substitution operator: first a substitution of \(a_2\) for all occurrences of \(c\), and next a substitution of \(a_1^\lor\) for the new occurrences of \(c\), is equivalent with an immediate substitution of \([a^\lor c]a_2\) for all occurrences of \(c\). The proof of the theorem is as follows.

First consider \(<c^\lor d, s>(<c^\lor d_2>)s\), where \(d_1\) and \(d_2\) are possible values of \(c\). This denotes a state in which all identifiers have the same value as in \(s\), except for \(c\) which has value \(d_1^\lor\). So it is the same state as \(<c^\lor d_1^\lor\)s.
(due to the distinctness postulate). This equivalence is used in the proof below.

\[
\begin{align*}
V_{s,\mathcal{G}}(a_1/c)(a_2/c)(\psi) &= V_{c+C}s,\mathcal{G}(a_1)>s,\mathcal{G}(a_2/c)(\psi) = \\
V_{c+C}s,\mathcal{G}(a_1)>c+c>s,\mathcal{G}(a_1)>s,\mathcal{G}(\psi) = \\
V_{c+C}s,\mathcal{G}(a_1)>c+c>s,\mathcal{G}(a_2/c)(\psi) = \\
V_{c+C}s,\mathcal{G}(\{a_1/c\}a_2/c)(\psi) &= V_{s,\mathcal{G}}(a_1/c)(a_2/c)(\psi)
\end{align*}
\]

4.7. END

5. FIRST FRAGMENT

5.1. The rules

In this section the syntax and semantics will be presented of a small fragment of a programming language. The fragment contains only programs which consist of a sequence of simple assignment statements; many programming languages have a fragment like the one presented here. The treatment will be in accordance with the framework developed in the first chapters of this book. This means that for each basic expression (generator of the syntactic algebra) there has to be a translation into the logic, and that for each syntactic rule there has to be a corresponding semantic rule which says how the translations of the parts of a syntactic construction have to be combined in order to obtain the meaning of the compound construction.

The syntax of the fragment has the following five categories:

1. INT The set of representations of integers. Basic expressions in this category are: 1, 2, 3, ..., 12, ..., 666, ..., .
2. ID The set of integer identifiers. Basic expressions are x, y and z.
3. ASS The set of assignments.
4. PROG The set of programs.
5. BOOL The set of boolean expressions.

The basic expressions of the category INT translate into corresponding constants of type e; the translation of I is 1 etc. The identifiers x, y and z translate into corresponding constants of type <s,e>: the translation of x is x.

The syntactic rules are presented in the same way as in previous chapters. In the clause called 'rule', the categories involved are mentioned;
first the categories of the input expressions, then the category of the resulting expression. The F-clause describes the operation which is performed on the input expressions; here α always stands for the first input expression, β for the second, and γ for the third. The T-clause describes how the translation of the resulting expression is built up from the translations of the input expressions. Here α' denotes the translation of the first input expression, β' of the second, and γ' of the third.

Rule

\[ S_{la} : \text{INT} \times \text{INT} \rightarrow \text{BOOL} \]
\[ F_{la} : \alpha = \beta \]
\[ T_{la} : \alpha' = \beta'. \]

Example

\[ S_{la} : \text{Out of the integer expressions } l \text{ and } z, \text{ we may build the boolean expression } l < z, \text{ with as translation } l < 2. \]

Rules \( S_{lb} \cdot S_{le} \): Analogously for the relations >, <, ≥, =.

Rule

\[ S_{2a} : \text{INT} \times \text{INT} \rightarrow \text{INT} \]
\[ F_{2a} : \alpha + \beta \]
\[ T_{2a} : \alpha' + \beta'. \]

Example

\[ (l+z)' = l + 2 \]

Rules \( S_{2b} \cdot S_{2c} \): Analogously for the operations × and +

Rule

\[ S_{3} : \text{ID} \rightarrow \text{INT} \]
\[ F_{3} : \alpha \]
\[ T_{3} : \alpha'. \]

Example

\[ : \text{The integer identifier } x \text{ can be used to denote an integer.} \]

Rule

\[ S_{4} : \text{ID} \times \text{INT} \rightarrow \text{ASS} \]
\[ F_{4} : \alpha := \beta \]
\[ T_{4} : \lambda P \exists z \{ [z/x'] \forall P \land \forall z'_{e} = \{ z/x' \} \beta' \} (z \in \text{VAR}_{e}) \]

Example

\[ : \text{See below. Notice the similarity and differences between this predicate transformer and Floyd's original rule. Some extension operators have been added, and the substitution operator is replaced by an operator with a semantical interpretation.} \]

Rule

\[ S_{5} : \text{ASS} \rightarrow \text{PROG} \]
\[ F_{5} : \alpha \]
\[ T_{5} : \alpha'. \]

Example

\[ : \text{Every assignment statement can be used as a (reduced) program.} \]
Rule
\[ S_6 : \text{PROG} \times \text{PROG} \rightarrow \text{PROG} \]
\[ F_6 : a \beta \]
\[ T_6 : \lambda P[a' \beta'(\beta' P)’. \gamma'. \gamma'. \gamma'] \]

Rule
\[ S_7 : \text{BOOL} \times \text{PROG} \times \text{PROG} \rightarrow \text{PROG} \]
\[ F_7 : if a then \beta else \gamma f' \]
\[ T_7 : \lambda P[\beta' (\beta' \gamma P) \lor \gamma' (\gamma' \gamma')] \]

5.2. Examples

5.1. EXAMPLE: \( x := y \).

The derivational history of this assignment is presented in figure 4. Also the successive steps of the translation process are presented in the tree. At each stage the number of the rule used and the category of the produced expression are mentioned between braces.

![Figure 4](image)

The obtained translation of the program can be reduced, using the substitution theorem, to (25)

(25) \( \lambda P[\exists[x/y] \gamma \beta P \land \gamma y = x] \).

Now suppose that before the execution of the assignment \( x \) equals \( \gamma \) and \( y \) equals \( \beta \) (cf. Section 1, Figure 2c). So the initial state satisfies predicate (26):

(26) \( x = \gamma \land y = \beta \).
Then after the execution of the assignment the following holds:

(27) $\lambda \exists z [[z/y]^P \land y = x]](\forall \cdot x = 7 \land y = 2)).$

This reduces to (28), and further to (29) and (30).

(28) $\exists z [[z/y](x = 7 \land y = 2) \land y = x])$
(29) $\exists y (\forall x = 7 \land z = 2 \land y = x)$
(30) $\forall x = 7 \land y = x.$

5.2. EXAMPLE: $y := x; \ y := y + 1.$

The translation of the second assignment statement is obtained in the same way as the translation of $y := z$ in example 5.1. Its translation is (31), which reduces to (32).

(31) $\lambda \exists z [[z/y]^P \land y = \{z/y\}(y+1)]$
(32) $\lambda \exists z [[z/y]^P \land y = z+1].$

The translation of the whole program is therefore

$\lambda \exists y := y + 1(\forall [[y := x](y)] =
\lambda \exists y := y + 1(\exists z [[z/y]^P \land y = z+1])(\forall \exists z [[z/y]^Q \land y = x]) =
\lambda \exists z [[z/y]^P(\exists w [[w/y]^Q \land y = x]) \land y = z+1) =
\lambda \exists z \exists w [[z/y]^Q \land z = x \land y = z+1].$

Suppose now that before the execution of the program $x > 0$ holds. Then afterwards (33) holds, which reduces in turn to (34) and further to (35).

(33) $\exists z \exists w [[w/y]^Q(x > 0) \land z = x \land y = z+1]
(34) \exists z x > 0 \land z = x \land y = z+1
(35) x > 0 \land y = x + 1.$

In the treatment of this program we first determined the translation of the program, and then considered some specific precondition. If we knew the precondition beforehand, and were only interested in obtaining the post-condition (and not in obtaining the translation of the whole program), we could first calculate the post-condition after the first assignment. This postcondition could then be taken as precondition for the second assignment.
5.3. EXAMPLE: \( \text{if } y < 0 \text{ then } x := z \text{ else } y := y+1 \). 

The predicate transformer corresponding with this program is

\[
\lambda \Box \lambda P [\exists z' \forall y \cdot P \wedge y \equiv x] \equiv \forall y \leq y_0 \wedge \forall q \cdot \lambda P [\exists z' \forall y \cdot P \wedge y \equiv z+1] \equiv \forall y \leq y_0 \wedge \forall q \cdot 
\]

This reduces to

\[
\lambda \Box [\exists z < 0 \wedge (z' \forall y) \forall y \equiv x] \equiv \exists z \forall y \leq y_0 \wedge (z' \forall y) \forall y \equiv y + 1. 
\]

Suppose that we have no information about the state before the execution of the assignment. This is expressed by the precondition \( I = \top \). Then afterwards (38) holds, which reduces to (39).

\[
\exists z < 0 \wedge y \equiv x \equiv \exists z \forall y \leq y_0 \wedge y \equiv y + 1. 
\]

(39) \( y \equiv x \equiv y \geq 1 \).

5.3. END

6. POINTERS AND ARRAYS

6.1. Pointers

An application of Floyd's rule to assignments containing pointers may give rise to problems, see the example in section 3. In sections 4–6 we have developed a compositional, algebraic approach for simple assignments. This algebraic approach can be generalized in a straightforward way to the case of pointers. I will consider at this moment only pointers to integer identifiers; a more general and formal treatment will be given in section 7.

Pointers to integer identifiers are expressions which have as value in a given state some integer identifier. In another state they may have another identifier as value. Therefore we associate with a pointer some function from states to interpretations of integer identifiers. In analogy to the treatment of integer identifiers, this is done by translating the pointer into a rigid constant; so pointer \( p \) translates into constants \( p \in CON_{\leq s, \leq s_{e}} \). The execution of the assignment \( p := y \) has as an effect that the current state is changed in such a way that in the new state all identifiers have the same value as before, except for \( p \) which now has value \( y \). This effect can be described by means of a state switcher like the ones we introduced in relation with simple assignments. Below I will
introduce some postulates which guarantee that the state switchers \( \{z/\nu p\} \)

\( \lambda p[\exists z([z/\nu p] \nu p \land \nu p = [z/\nu p] \delta)] \quad (z \in \text{VAR}_{<s,e>}.) \)

6.1. **Example.** \( x := \delta; \ p := \alpha; \ z := \beta. \)

Let us assume that this program is executed in an arbitrary state, so the

precondition is **true**. We are interested in the postcondition after the last

assignment. That postcondition is obtained by calculating the postcondition

each assignment in turn, and taking that postcondition as input for the

predicate transformer of the next assignment. The postcondition of the

first assignment for precondition **true** reduces as follows.

\[ \lambda p[\exists z([z/\nu p] \nu p \land \nu p = [z/\nu p] \delta)](\nu \text{true}) = \exists z(\nu x = 5) = \nu x = 5. \]

The postcondition of the second assignment (using the predicate transformer

described above) reduces as follows

\[ \lambda p[\exists z([z/\nu p] \nu p \lor \nu p = x)](\nu \text{true}) = \exists z([z/\nu p] (\nu x = 5) \land \nu p = x) = [\nu x = 5 \land \nu p = x]. \]

Finally, the postcondition of the last assignment reduces as follows:

\[ \lambda p[\exists z([z/\nu x] \nu x = 6)](\nu x = 5 \land \nu p = x) = \exists z([z/\nu x] (\nu x = 5 \land \nu p = x) \land \nu x = 6) = \nu p = 6, \]

Therefore, the postcondition has as consequence that the integer value related with \( p \) is 6. This is as it should be (see figure 2).

If we compare the treatment of this program with the treatment using

Floyd's rule see (18)-(23), then we observe that this success is due to a

careful distinction between the representation of the interpretation of

identifier \( x \), namely \( x \), and the representation of the value of that identi-

fier, namely \( x \). This has as its effect that in the calculation of the last

postcondition the \( x \) in the identity \( p = x \) is not replaced by \( z \) as would be

the case if Floyd's rule were used.

6.1. END

For the constants which translate pointers, we have postulates analogous to

the ones we have for constants translating integer identifiers. (rigidness
postulate, distinctness postulate, update postulate). Something more, however, has to be said about the possible values of pointer constants. Consider \( p \in \text{CON}_{\langle s, e \rangle} \). This constant is interpreted as a function from states to objects of type \( \langle s, e \rangle \). Not all such objects are possible values of pointers. In a given state the extension of \( p \) has to be the interpretation of some integer identifier and we have already formulated some requirements concerning such interpretations (update postulate etc.). For instance, the interpretation of an integer identifier cannot be a constant function yielding for all states the same value. Consequently the extension of \( p \) cannot be such an object. Thus we arrive at the following postulate concerning the constants of type \( \langle s, e \rangle \) (for higher order pointers analogous requirements will be given).

6.2. Properness postulate

For all \( c \in \text{CON}_{\langle s, e \rangle} \), \( s \in \text{ST} \)

\[
V(c)(s) \in \{ V(c') \mid c' \in \text{CON}_{\langle s, e \rangle} \}.
\]

6.2. END

6.2. Arrays

In section 3 it was shown that a straightforward application of Floyd's rule to assignments containing subscripted array identifiers may yield incorrect results. Here a compositional treatment of the semantics of such assignments will be developed (the formal treatment will be given in 7). In order to have a comparison for the treatment, I will first sketch a treatment due to De BAKKER (1976, 1980).

De BAKKER presents an extension of Floyd's proof rule for the case of assignment statements. His treatment is based on the definition of a new kind of substitution operator \([a/\beta]\). In most cases this operator behaves as the ordinary substitution operator, but not in the case that both \( a \) and \( \beta \) are of the form array-identifier-with-subscript. Then this substitution may result in a compound expression containing an if-then-else construction. The relevant clause of the definition of the operator is as follows.
\begin{equation}
\begin{array}{l}
[t/a[s_1]](b[s_2]) = b[[t/a[s_1]](s_2)] \\
[t/a[s_1]](a[s_2]) = \text{if } [t/a[s_1]]s_2 = s_1 \text{ then } t \text{ else } a[[t/a[s_1]]s_2].
\end{array}
\end{equation}

Using this operator, De Bakker gives a variant of Floyd’s rule for assignment statements:

\begin{equation}
\{ \emptyset \} a[s] := t \{ \exists y \exists z [[y/a[z]](\emptyset) \land z = [y/a[z]](s) \land a[z] = [y/a[z]](t)\}.
\end{equation}

6.3. **Example**

Assignment: \(a[\alpha[1]] := 2\). Precondition: \(a[1] = 1 \land a[2] = 1\).

Postcondition:

\( \exists y \exists z ([y/a[z]](a[1]) = 1 \land a[2] = 1) \land z = [y/a[z]](a[1]) \land a[z] = [y/a[z]](z) \).

By the definition of substitution this reduces to

\( \exists y \exists z \text{ if } l=z \text{ then } y \text{ else } a[1] \text{ fi } = 1 \land \text{ if } 2=z \text{ then } y \text{ else } a[2] \text{ fi } = 1 \land z = \text{ if } l=z \text{ then } y \text{ else } a[1] \text{ fi } \land a[z]=2). \)

From the second and the third boolean expression in the conjunction, we see that we must take \(z=1\), and the whole expression reduces to:

\( \exists y \text{ if } 1=y \land a[2]=1 \land l=y \land a[1]=2\).

This is in turn equivalent to \(a[1]=2 \land a[2]=1\), from which it follows that \(a[a[1]]=2\).

This proof rule works correctly! It is not easy to understand why the rule works, but De Bakker has proven its correctness.

6.3. **End**

From our methodological point of view this solution has the same disadvantages as Floyd’s original proposal, the main one being that the substitution operator defined in (41) has no semantic interpretation. In order to obtain a solution within the limits of our framework, let us consider the 'parts' of the assignment \(a[s] := t\). The usual syntax says that there are two parts: the left hand side, (i.e. \(a[s]\)), and the right hand side (i.e. \(t\)). The left hand side has as its parts an array identifier (i.e. \(a\)) and an integer expression (i.e. \(s\)). This analysis has as a consequence that, in our algebraic approach, we have to associate with the array identifier \(a\) some semantic object. In the papers by De Bakker this is not done, nor is this done by several other authors in the field. One usually employs a model which is an abstract computer model with cells, and it is not possible to associate some cell with \(a\). In our model, on the other hand, it is not difficult to associate some semantic object with \(a\). For each state an array identifier determines a function from integers (subscripts) to
integers (the value contained in the cell with that subscript). In analogy
to the treatment of integer identifiers, this relation between an identi-
fier and the associated function, is given by translating the array identi-
fier into a rigid constant of type \(<s,\langle e, e, \rangle>\).

Using the fact that it makes sense to speak about the value of (the
translation of) an array identifier \(a\), we can easily describe the effect of
the execution of an assignment \(a[\beta] := \gamma\). By this assignment a state is
reached in which the value associated with \(a\) differs for one argument from
its old value. If the old value of \(a\) is denoted by \(z\), then the new value of \(a\) is,
roughly, described by: \(\lambda n[\text{if } n = \beta \text{ then } \gamma \text{ else } z(n) \text{ fi}]\). I said
'roughly' since it is not yet expressed, for \(\beta\) and \(\gamma\), to take here the old
value of \(a\). These considerations give rise to the following predicate trans-
former associated with \(a[\beta] := \gamma\):

\[
(43) \lambda P(\exists z[\langle z/a \rangle^P \land a = \{z/a\}(\lambda n[\text{if } n = \beta \text{ then } \gamma \text{ else } z(n) \text{ fi}])))
\]

Notice the direct analogy of this predicate transformer with the predicate transformer
for the simple assignment. The correctness of (43) is, I believe,
much clearer than of the one given by De Bakker. This perspicuity is due
to the fact that we treat the array identifiers as having a meaning. In a
model based upon the use of 'cells', such an approach does not come naturally.
The main point of the present approach (arrays as functions) is the
basis for the treatment of arrays in GRIES 1977. It turned out that already
in HOARE & WIRTH 1973 arrays are considered as denoting functions (however
not in the context of the problems under discussion).

6.4. EXAMPLE. Consider the assignment \(a[a[1]] := z\), executed in a state in
which \(a[1] = 1\) and \(a[2] = 1\). We wish to find the strongest postcondition
in this situation. This is found by application of the predicate trans-
former (associated with the assignment) to the precondition expressing the
mentioned property of the state. In the logical formulas given below I
should write \(a(l)\) etc., since we interpret \(a\) as a function. But in order
to keep in mind what we are modelling, I prefer the notation \(a[l]\)

\[
[a[a[1]] := z](\langle [a[1]] = 1 \land [a[2]] = 1 \rangle) = \\
= \lambda P(\exists z[\langle z/a \rangle^P \land a = \{z/a\}(\lambda n[\text{if } n = a[1] \text{ then } 2 \text{ else } \\
\langle z(n) \text{ fi} \rangle)(\langle [a[1]] = 1 \land [a[2]] = 1 \rangle) = \\
= \exists z[z[1] = 1 \land z[2] = 1 \land z = \lambda n[\text{if } n = z[1] \text{ then } 2 \text{ else } z(n) \text{ fi}] = \\
= \exists z[z[1] = 1 \land z[2] = 1 \land \langle z/a \rangle = \lambda n[\text{if } n = 1 \text{ then } 2 \text{ else } z(n) \text{ fi}].
\]
From this postcondition the value of $a[a[1]]$ can be calculated:

$$V[a] \cdot a[1]] = V[a \land n \text{ if } n=1 \text{ then } 2 \text{ else } a[n] \cdot f_i[1]] = V[a[2] = z[2] = 1.$$

6.4. END

Now that we know which predicate transformer should be used, let us look at how it was obtained. We could have tried to find some translation for the left hand side of the assignment (i.e. for $a[n]$), out of which the predicate transformer could be formed. It turned out to be preferable to use the insights obtained from considerations based on the principle of compositionality. We observed that $a[s] := t$ is a notation for changing the function associated with $a$. This suggests to consider such an assignment as a three-place syntactic operation which takes as inputs the array identifier, the subscript expression, and the expression at the right hand side of the $:= \text{ sign}$. In such an approach it is easy to obtain the desired predicate transformer, and therefore this approach will be followed. This shows that semantic considerations may influence the design of the syntactic rules (however, a binary approach to the assignment is not forbidden).

In JANSSEN & Van EMDE BOAS (1977a) assignments to multi-dimensional arrays are treated. Since the proposal given there, is not strictly in accordance with the principle of compositionality, it is not mentioned here. One could incorporate assignments to $n$-dimensional arrays by introducing a separate rule for each choice of $n$; then an $n$-dimensional array is considered as a function of $n$ arguments.

7. SECOND FRAGMENT

7.1. The rules

In this section I will present the syntax and semantics of a fragment of the programming language ALGOL-68 (Van WIJNGAARDEN 1975). The fragment contains integer identifiers, pointers to integer identifiers, pointers to such pointers, etc., so there is in principle an infinite hierarchy of pointers. The fragment also contains arrays of integers, arrays of integer identifiers, arrays of pointers to integer identifiers, etc., so in principle an infinite hierarchy of arrays. In order to deal with such infinite sets, the syntax will contain rule schemata. These schemata are like the hyperrules used in the official ALGOL-68 report (Van WIJNGAARDEN 1975). Following the framework from ch.1, the semantics of the fragment will be
described by means of a translation into intensional logic. As explained in the previous section, this logic has to be interpreted in a restricted class of models. The models have to satisfy certain postulates; these will be presented in section 7.3. In section 7.4 a model will be constructed that satisfies these postulates.

The names of the categories used in section 5 have to be changed in order to follow the ALGOL-68 terminology. The category of integers will be called 'int id' (i.e. integer identifier) and the category of integer identifiers ID will be called 'ref int id' (i.e. reference to integer identifier). As explained above there will be an infinite set of categories. In the description of a category name we may use the meta notion mode. The possible substitutions for this metanotation are described by the following meta rules:

\[
\begin{align*}
\text{mode} & \rightarrow \text{int} \\
\text{mode} & \rightarrow \text{ref mode} \\
\text{mode} & \rightarrow \text{row of mode}
\end{align*}
\]

These modes correspond with types of intensional logic; this correspondence is formalized by the mapping \( \tau \) which is defined as follows.

\[
\begin{align*}
\tau(\text{int}) & = e \\
\tau(\text{bool}) & = t \\
\tau(\text{ref mode}) & = \langle s, \tau(\text{mode}) \rangle \\
\tau(\text{row of mode}) & = \langle e, \tau(\text{mode}) \rangle.
\end{align*}
\]

For each 'mode' there is a category 'mode id' which contains denumerable many expressions: the identifiers of that mode. Examples are:

<table>
<thead>
<tr>
<th>Category</th>
<th>Typical identifiers</th>
</tr>
</thead>
<tbody>
<tr>
<td>int id</td>
<td>1, 2, z, ..., ( \ldots )</td>
</tr>
<tr>
<td>ref int id</td>
<td>( x, y, w, x_1, x_2, \ldots )</td>
</tr>
<tr>
<td>ref ref int id</td>
<td>( p, q, p_1, p_2, \ldots )</td>
</tr>
<tr>
<td>row of int id</td>
<td>( a, a_1, a_2, \ldots )</td>
</tr>
</tbody>
</table>

The rule schemata of the fragment are presented in the same way as the rules presented in section 5. The main difference is that in section 5 we had actual rules, whereas we here have schemata which become actual rules by means of a substitution for mode. Important is that throughout one scheme the same substitution for mode has to be used.
Rule \( B_1 \cdots B_5 \) : int exp \( \times \) int exp \( \rightarrow \) bool exp
\[ \begin{align*}
FB_1 \cdots FB_5 & : \alpha \equiv \beta \quad \text{where} \equiv \text{stands for} <, >, =, \text{or} =.
TB_1 \cdots TB_5 & : a' \equiv \beta' \quad \text{idem for} \equiv.
\end{align*} \]

Rule \( I_1 \cdots I_3 \) : int exp \( \times \) int exp \( + \) int exp
\[ \begin{align*}
FI_1 \cdots FI_3 & : \alpha \oplus \beta \quad \text{where} \oplus \text{stands for} +, \times, \oplus \text{respectively}
TI_1 \cdots TI_1 & : a' \oplus \beta' \quad \text{idem for} \oplus.
\end{align*} \]

Rule \( E_1 \) : mode id \( \rightarrow \) mode unit
\[ \begin{align*}
FE_1 & : \alpha
TE_1 & : \alpha'.
\end{align*} \]

Rule \( E_2 \) : mode unit \( \rightarrow \) mode exp
\[ \begin{align*}
FE_2 & : \alpha
TE_2 & : \alpha'.
\end{align*} \]

Rule \( E_3 \) : ref mode exp \( \rightarrow \) mode exp
\[ \begin{align*}
FE_3 & : \alpha
TE_3 & : \alpha'.
\end{align*} \]

Rule \( E_4 \) : bool exp \( \times \) mode unit \( \times \) mode unit \( \rightarrow \) mode unit
\[ \begin{align*}
FE_4 & : \text{if } \alpha \text{ then } \beta \text{ else } \gamma \text{ fi}
TE_4 & : \text{if } \alpha' \text{ then } \beta' \text{ else } \gamma' \text{ fi}
\end{align*} \]

Comment: The rule is defined for units and not for exp's in order to avoid the problems of 'balancing' (see e.g. Van WIJNGAARDEN 1975).

Rule \( E_5 \) : ref row of mode unit \( \times \) int exp \( \rightarrow \) ref mode unit
\[ \begin{align*}
FE_5 & : \alpha'\beta'[\beta']
TE_5 & : \lambda[\alpha'[\beta']].
\end{align*} \]

Rule \( A_1 \) : ref mode id \( \times \) mode exp \( \rightarrow \) ass
\[ \begin{align*}
FA_1 & : \alpha := \beta
TA_1 & : \lambda \exists z(z/\alpha')^p \land \forall \alpha' = \{z/\alpha'[\beta'] \} \quad \text{where} \ z \in \text{VAR}_{\tau}(\text{mode}).
\end{align*} \]

Rule \( A_2 \) : ref row of mode id \( \times \) int exp \( \times \) mode exp \( \rightarrow \) ass
\[ \begin{align*}
FA_2 & : \alpha[\beta] := \gamma
TA_2 & : \lambda \exists z(z/\alpha')^p \land \forall \alpha' = \{z/\alpha'[\gamma'] \} \quad \text{if} \ n = \beta' \ \text{then} \ \gamma' \ \text{else} \ \text{fi}\n\end{align*} \]

Rule $P_1$ \quad : \text{ass } \rightarrow \text{simple prog}

FP$_1$ \quad : \alpha

TP$_1$ \quad : \alpha'.

Rule $P_2$ \quad : \text{simple prog } \rightarrow \text{prog}

FP$_2$ \quad : \alpha

TP$_2$ \quad : \alpha'.

Rule $P_3$ \quad : \text{prog } \times \text{simple prog } \rightarrow \text{prog}

FP$_3$ \quad : \alpha;\beta

TP$_3$ \quad : \lambda P(a'(\top[\beta'(P)])).

Rule $P_4$ \quad : \text{bool exp } \times \text{ prog } \times \text{ prog } \rightarrow \text{ prog}

FP$_4$ \quad : \text{if } a \text{ then } b \text{ else } c \text{ fi}

TP$_4$ \quad : \lambda P(b'(\top[a' \land \top P])) \lor \top'(\top[\neg a' \land \top P])].

7.1. EXAMPLE. In section 5 I have given several examples of assignment statements. Therefore now as example a somewhat more complex program

\[ p := a[1]; a[1] := 2 \quad \text{precondition } a[1]=1 \land a[2]=2. \]

The postcondition after the first assignment is:

\[ [p := a[1]]'(\top[a[1]=1 \land a[2]=2]) = \]

\[ \lambda Pz(z)'P \land P = (z)'P [\top[a[1]]](\top[a[1]=1 \land a[2]=2]) = \]

\[ \top[a[1]=1 \land a[2]=2] \land P = \top[a[1]]. \]

Then the postcondition after the second assignment is:

\[ \exists z[a]'P \land [\top[a[1]]=1 \land a[2]=2 \land P = \top[a[1]]] \land \]

\[ \top[a = \lambda n \text{ if } n = 1 \text{ then } 2 \text{ else } a[n] \text{ fi}] = \]

\[ = \exists z[z[1]=1 \land z[2]=2 \land P = \top[a[1]]] \land \]

\[ \top[a = \lambda n \text{ if } n = 1 \text{ then } 2 \text{ else } z[n] \text{ fi}]. \]

From this we conclude that $\top P = a[1] = 2$.

7.1. END

7.2. The postulates

In order to formulate the postulates, I will first define the set $\mathcal{AT}$ of achievable types. This set consists of the types which are achievable by translating expressions of categories which have a name obtained from
the name-scheme 'mode \text{ exp}'.

7.2. DEFINITION. The set $\text{AT} \subset \text{Ty}$ is defined by the following clauses:

1. $e \in \text{AT}$
2. If $\tau \in \text{AT}$ then $<s, \tau> \in \text{AT}$ and $<e, \tau> \in \text{AT}$.

7.2. END

The rigidness postulate says that all constants are rigid designators.

7.3. Rigidness Postulate

For all $\tau \in \text{AT}$ and $c \in \text{CON}_{<s, \tau>}$: $\exists! \bar{c}$ [c\bar{c}].

7.3. END

The distinctness postulate says that two states are different only if they give rise to a different extension of some constant.

7.4. Distinctness Postulate

Let $s, t \in \text{ST}$. If for all $\tau \in \text{AT}$ and $c \in \text{CON}_{<s, \tau>}$ we have $V(c)(s) = V(c)(t)$, then $s = t$.

7.4. END

The properness postulate says, roughly, that the extension of a constant has to be a value that can be achieved by executing instructions from the programming language. First we define these sets $\text{AV}_\tau$ of achievable values of type $\tau$.

7.5. DEFINITION. The sets $\text{AV}_\tau (\tau \in \text{AT})$ of achievable values of type $\tau$ are defined as the smallest sets satisfying the following clauses.

I. $AV_\emptyset = \mathbb{N}$
II. $\{V(c) \mid c \in \text{CON}_{<s, \tau>}\} \subset AV_{<s, \tau>}$
III. if $\rho \in AV_{<s, <e, \tau> >}$ and $n \in \mathbb{N}$ then $\lambda s[\rho(s)](n) \in AV_{<s, \tau>}$
IV. $AV_{<e, \tau>} = AV_\emptyset$

7.6. Properness Postulate

For all $s \in \text{ST}, \tau \in \text{AT}, c \in \text{CON}_{<s, \tau>}$ we have $V(c)(s) \in AV_\tau$.

7.6. END
The update postulate says that the model should have such a richness that the value of one identifier can be changed into arbitrary achievable value, without changing the values of other identifiers.

7.8. Update Postulate

For all \( s \in ST, \tau \in AT, c \in \text{CON}_{<s, \tau>}, d \in AV, \) there is a state \( t \in ST \) such that
1. \( V(c)(t) = d \)
2. \( V(c')(t) = V(c')(s) \) for all constants \( c' \neq c \).

7.8. END

The update postulate only requires 'updating' to an achievable value. This means that the interpretation of \( \langle a'/c \rangle \) can be defined as follows.

7.9. DEFINITION.

\[
V_{s, \phi} \langle a'/c \rangle = \begin{cases} 
V_{<c',V_{s, \phi}}>s, \phi & \text{if } V_{s, \phi}(a) \text{ is achievable} \\
V_{s, \phi} & \text{otherwise.}
\end{cases}
\]

7.9. END

7.3. A model

The postulates concerning the model can be distinguished in two groups. Some of the postulates require a certain richness of the model (the distinctness postulate and the update postulate), other postulates limit this richness (rigidness postulate and properness postulate). I will show that it is possible to steer a course between this Scylla and Charibdis by constructing a model which satisfies all these postulates.

The model will be built from the set of natural numbers and a set of states. This set of states should have a certain richness since the model has to fulfill the update postulate (every constant can take every achievable value). In order to obtain this effect one would like to take as set of states the cartesian product of the sets \( AV, \) of achievable values of type \( \tau, \) This method cannot be used since the achievable values themselves are defined using the set of states (clause III of their definition). Therefore we will first introduce a collection of expressions which will turn out to be in a one-one correspondence with the achievable values. The
set of states will be defined as the cartesian product of the sets of these expressions.

7.11. **DEFINITION.** The sets $\text{AE}_\tau$ of achievable value expressions of type $\tau$ are defined as the smallest sets satisfying the following clauses:

1. $\forall i \in \text{CON}_e \quad i \in \text{AE}_e$
2. $\forall c \in \text{CON}_{<s,\tau>} \quad c \in \text{AE}_{<s,\tau>}$
3. $\forall i \in \text{AE}_e \quad \text{and for } \forall \rho \in \text{AE}_{<s,e,\tau>}: \quad \rho[i] \in \text{AE}_{<s,\tau>}$
4. If for all $n \in \mathbb{N}$: $\phi_n \in \text{AE}_\tau$ then $(\phi_n)_{n \in \mathbb{N}} \in \text{AE}_{<e,\tau>}$.  

7.10. **END**

Clause (4) introduced infinite sequences of symbols. They arise since we did not formalize the finiteness of arrays. The above definition has as a consequence that corresponding to each achievable value given by the definition of $\text{AV}$, there is an expression in $\text{AE}$.

A model for IL satisfying the postulates is now constructed as follows. We use the sets $\text{AE}_\tau$ of achievable value denotations and define the set of states by

$$ S = \prod_{\tau \in \text{CON}_{<s,\tau>}} \text{AE}_\tau. $$

For $c \in \text{CON}_{<s,\tau>}$ we denote the projection on the $c$-th coordinate of a state $s$ by $\Pi_c(s)$.

Having chosen the set $S$, the sets $D_\tau$ are determined for each type $\tau$.

To complete the description of the model we must explain how $V(c)$ is defined for constants. This function is defined simultaneously with a mapping $G: \bigcup_{\tau \in \text{CON}_{<s,\tau>}} \text{AE}_\tau \to \bigcup_{\tau \in \text{CON}_{<s,\tau>}} \text{AV}_\tau$.

1. $V(i) = G(i) = i$ for $i \in \text{AE}_e$
   i.e. a number denotations are mapped onto the integers denoted by them.
2. $V(c) = G(c) = \lambda s [G(c)(s)]$ for $c \in \text{CON}_{<s,\tau>}$
3. $G(\rho[i]) = \lambda s [G(\rho)(s)[G(i)]]$ for $\rho \in \text{AE}_{<s,e,\tau>}$
4. $G(\phi_n) = \lambda n [G(n)]$ for $(\phi_n)_{n \in \mathbb{N}} \in \text{AE}_{<e,\tau>}$

Clearly the map $G: \bigcup_{\tau \in \text{CON}_{<s,\tau>}} \text{AE}_\tau \to \bigcup_{\tau \in \text{CON}_{<s,\tau>}} \text{AV}_\tau$ in this way becomes a bijection. So all elements in the model which are of an achievable type, are achievable values. Moreover, the model satisfies all postulates, due to the definition of the set $S$. 
8. CORRECTNESS AND COMPLETENESS

8.1. State transition semantics

In the previous section the meaning a program is defined, and one might expect that the story ends there. But the kind of meanings (predicate transformers) are far removed from the behaviour of a computer while executing a program. One might ask whether we did not loose the connection with a notion of meaning that is more connected with the behaviour of computers. In order to answer this question another kind of semantics will be considered; one in which the meanings of assignments and programs are defined as mappings from states to states, rather than as predicate transformers. I will call it a state-transition semantics; it is related with the standard denotational semantics.

In order to express such a state transition semantics, we need a language in which states can be represented. In the present context the best choice seems to be Ty2: two sorted type theory (see chapter 3, or GALLIN 1975, for a definition). For our purposes this language is extended with state switchers:

8.1. Definition. If $\tau \in AT$, $c \in \text{CON}_{<\beta>}$, $\beta \in \text{ME}_{\tau}$ and $s \in \text{ME}_{\delta}$, then $<c+\beta>s \in \text{ME}_{\delta}$. The interpretation of such an expression is defined by

$$V(\langle c+\beta \rangle s) = V_{\delta}(s),$$

where $g' \sim g$ and $g'(s)$ is the unique state $t$, such that $V(c)(t) = V(\beta)$ and $V(c')(t) = V(c')(s)$ if $c' \neq c$, if such a state exists (the update postulate guarantees uniqueness); otherwise $g'(s) = s$.

8.1. End

The state-transition semantics of the fragment is defined by means of providing for a translation into Ty2. The translation function will be denoted as $\cdot$. For the identifiers the translation into Ty2 is the same as the translation into IL, so $\chi' = \chi''$ for all identifiers $\chi$.

For most of the translation rules into Ty2 the formulation can easily be obtained from the translation rules into IL using the standard formulation of IL in Ty2 (see chapter 3). Therefore I will present here only those rules which are essentially different: the rules concerning assignments and programs.
Rule \( P_1 : \text{Ass} \rightarrow \text{Simple Prog} \)
\[
\begin{align*}
\text{FP}_1 & : a \\
\text{OP}_1 & : a".
\end{align*}
\]

Rule \( P_2 : \text{Simple Prog} \rightarrow \text{Prog} \)
\[
\begin{align*}
\text{FP}_2 & : a \\
\text{OP}_2 & : a".
\end{align*}
\]

Rule \( P_3 : \text{Prog} \times \text{Simple Prog} \rightarrow \text{Prog} \)
\[
\begin{align*}
\text{FP}_3 & : a\beta \\
\text{OP}_3 & : \lambda s[a"(\beta"(s))].
\end{align*}
\]

Rule \( P_4 : \text{Bool Exp} \times \text{Prog} \times \text{Prog} \rightarrow \text{Prog} \)
\[
\begin{align*}
\text{EP}_4 & : \text{i\text{-}f a then } \beta \text{ else } \gamma \text{ fi} \\
\text{OP}_4 & : \lambda s[\text{i\text{-}f a"(s) then } \beta" \text{ else } \gamma" \text{ fi}].
\end{align*}
\]

Rule \( A_1 : \text{Ref mode Id} \times \text{mode Exp} \rightarrow \text{Ass} \)
\[
\begin{align*}
\text{FA}_1 & : a := \beta \\
\text{OA}_1 & : \lambda s[<a"\beta">(s)].
\end{align*}
\]

Rule \( A_2 : \text{Ref Row of mode Id} \times \text{Int Exp} \times \text{mode Exp} \rightarrow \text{Ass} \)
\[
\begin{align*}
\text{TA}_2 & : a[\beta] := \gamma \\
\text{OA}_2 & : \lambda s[<a"\gamma">(s)].
\end{align*}
\]

8.2. **Strongest postconditions**

Our aim is to prove that the predicate transformers we have defined in the previous section, are correct with respect to the operational semantics", and that these predicate transformers give as much information about the final state as possible. The relevant notions are defined as follows.

8.2. **DEFINITION.** A forward predicate transformer \( \pi' \) is called **correct** with respect to program \( \pi \) if for all state predicates \( \phi \) and all states \( s \).

\[
\text{if } s \models \phi \text{ then } s"(s) \models \pi'(\phi).
\]

8.3. **DEFINITION.** A forward predicate transformer \( \pi' \) is called **maximal** with respect to program \( \pi \) if for all pairs of state predicates \( \phi, \psi \) holds:

\[
\text{if for all states } s: s \models \phi \text{ implies } s"(s) \models \psi, \\
\text{then } s \models \pi'(\phi) \rightarrow \psi.
\]
8.4. **Theorem.** Let $\pi$ be a program, and $\pi_1$ and $\pi_2$ be forward predicate transformers which are correct and maximal with respect to $\pi$. Then for all $\psi$:
$$\models \pi_1(\hat{\psi}) \iff \pi_2(\hat{\psi}).$$

**Proof.** Since $\pi_2$ is correct we have:
$$\text{if } s \models \psi \text{ then } \pi''(s) \models \pi_2(\hat{\psi}).$$

Since $\pi_1$ is maximal, from the above implication follows:
$$\models \pi_1(\hat{\psi}) \rightarrow \pi_2(\hat{\psi}).$$

Analogously we prove
$$\models \pi_2(\hat{\psi}) \rightarrow \pi_1(\hat{\psi}).$$

8.4. END

A consequence of this theorem is that all predicate transformers which are correct and maximal with respect to a certain program yield equivalent postconditions. This justifies the following definition.

8.5. **Definition.** Let $\pi$ be a program and $\phi$ an expression of type $t$. Now $sp(\pi, \phi)$ is a new expression of type $t$, called the strongest postcondition with respect to $\pi$ and $\phi$. The interpretation of $sp(\pi, \phi)$ is equal to the interpretation of $\pi'(\hat{\phi})$, where $\pi'$ is a forward predicate transformer which is correct and maximal with respect to $\pi$.

8.5. END

A notion which turns out to be useful for proving properties of predicate transformers is

8.6. **Definition.** A predicate transformer $\pi'$ is called recoverable with respect to program $\pi$ if for all states $t$ and state-predicates $\phi$

if $t \models \pi'(\hat{\phi})$ then there is a state $s$ such that $s \models \phi$ and $\pi''(s) = t$.

8.7. **Theorem.** If $\pi'$ is recoverable, then $\pi'$ is maximal.

**Proof.** Suppose that $\pi'$ is recoverable and assume that $s \models \phi$ implies that $\pi''(s) \models \psi$, but that not $\models \pi'(\hat{\phi}) \rightarrow \psi$. Then there is a state $t$ such that $t \models \pi'(\hat{\psi})$ and $t \models \neg \psi$. Since $\pi'$ is recoverable there is a state $s$ such that $s \models \phi$ and $\pi''(s) = t$. By assumption we also have $\pi''(s) \models \psi$. Contradiction.
8.8. **THEOREM.** The translation function \( \tau' \) defined in section 7 yields strongest postconditions.

**PROOF.** By induction to the structure of the possible programs. We only consider the case \( x := \delta \) because for other cases the proof is straightforward.

**Part 1: Correctness.** Let \( s \models \phi \) and \( t = \pi''(s) \). Thus \( t = <x := \delta'>s \). We have to prove that

\[
(44) \ t \models \exists z' \chi' \phi \land \chi' = [z'/\chi']\delta'.
\]

Let \( h \) be such that \( h(z) = \nu\chi \). Then for every formula \( \psi \):

\[
(45) \ \nu_{t,h}(z/\chi')\psi = \nu_{<x+h(z)>t,h}(\phi) = \nu_{s,h}(\psi).
\]

Therefore

\[
(46) \ t,h \models (z/\chi').\psi.
\]

Moreover

\[
(47) \ \nu_{t,h}(z/\chi')\delta' = \nu_{<x+h(z)>t,h}\delta' = \nu_{s,h}\delta' = \nu_{t,h}\chi'.
\]

This means that \( \tau' \) is correct.

**Part 2: Recoverability.** Let

\[
(48) \ t \models \exists z' \chi' \phi \land \chi' = [z'/\chi']\delta'.
\]

Thus there is a \( g \) such that (49) and (50) hold

\[
(49) \ t,g \models (z/\chi')\phi
\]

\[
(50) \ \nu_{t,g}(\chi') = \nu_{t,g}((z/\chi')\delta').
\]

We define \( s = <x^g(z)>t \), then we immediately conclude that \( s \models \phi \). We prove now that the value of \( \nu\chi' \) 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!)

\[
(51) \ \nu''_{s}(\chi') = \nu_{<x^g(z)>s}(\chi') = \nu_{s}(\delta') = \nu_{<x^g(z)>t}(\delta') = \nu_{t,g}((z/\chi')\delta') = \nu_{t}(\chi').
\]

Notice that this proof also holds in case that \( \delta \) is an \( \lambda \)-expression, or in case \( g(z) \) is not achievable. This means that \( \tau' \) is recoverable, hence \( \pi' \) is maximal.

8.8. END
8.3. Completeness

The notions 'completeness' and 'soundness' of a collection proof rules play an important role in the literature concerning the semantics of programming languages. Such collections are intended to be used for proving properties of programs. Our main aim was not to prove properties, but to define meanings. However, in the discussions about our approach the possibility to prove properties of programs played an important role. In the examples several proofs concerning effects of programs were given, and one of the arguments for using predicate transformers was their usefulness in proofs. Therefore it is interesting to consider our approach in the light of the notions 'soundness' and 'completeness'. First I will informally discuss these notions in their relation to the traditional approach (for a survey see APT 1981), there after I will try to transfer them to our approach.

In the traditional approaches one describes the relation between the assertions (state predicates) \( \phi \) and \( \psi \) and the program \( \pi \) by means of the correctness formula \( \{ \phi \} \pi \{ \psi \} \). This formula should be read as stating that if \( \phi \) holds before the execution of program \( \pi \), then \( \psi \) holds afterwards (for a discussion see section 2). Formula \( \phi \) is called a precondition, and \( \psi \) a postcondition. collection \( C \) of proof rules for such formulas consists of axioms, and of proof rules which allow to derive new formulas from already derived ones. For the basic constructions of the programming language certain formulas are given as axioms (e.g. Floyd's axiom for the assignment statement). An important proof rule is (52); the so called rule of consequence. It allows us to replace a precondition by a stronger statement, and a postcondition by a weaker statement.

(52) If \( \pi \triangleright p \), \( \{ p \} \pi \{ q \} \), \( q \rightarrow q \) are derived, then \( \{ p \} \pi \{ q \} \) follows.

The notion \( \vdash_C \) (derivable in \( C \)) is then defined as usual. Hence (53) means that the formula \( \{ \phi \} \pi \{ \psi \} \) can be derived from the axioms by using only rules from \( C \).

(53) \( \vdash_C \{ \phi \} \pi \{ \psi \} \).

Besides the syntactic notion \( \vdash_C \), the semantic notion \( \models_M \) of satisfaction in a model \( M \) is used. A model \( M \) is defined, in which assertions \( \phi \) and \( \psi \) can be interpreted and in which the execution of \( \pi \) is modelled. Then (54) says that it is true in \( M \) that if \( \phi \) holds before the execution of \( \pi \), then
ψ holds afterwards.

(54) \[ \vdash M (\psi) \rightarrow (\psi). \]

The notions soundness and completeness of a collection \( C \) of proof rules relate the syntactic notion \( \vdash C \) with the semantic notion \( \vdash M \). The collection \( C \) is called sound if for all \( \phi, \psi \) and \( \pi \)

(55) \[ \vdash C (\phi) \rightarrow (\pi) \rightarrow \vdash M (\phi) \rightarrow (\psi). \]

The collection \( C \) is called complete if for all \( \phi, \psi \) and \( \pi \)

(56) \[ \vdash M (\psi) \rightarrow (\pi) \rightarrow \vdash C (\phi) \rightarrow (\psi). \]

Most identifiers in computer programs have to be associated with numbers, and the assertions in correctness formulas may say something about the numerical values of these identifiers. We may consider a trivial program \( a \) (e.g. \( x := x \)) a trivial assertion \( \beta \) (e.g. \( \text{true} \)), and an arbitrary assertion \( \gamma \) from number theory. Then (57) holds.

(57) \[ \vdash M (\beta) \rightarrow (\alpha) \rightarrow \vdash M (\gamma). \]

Suppose now that we had a complete collection \( C \) of proof rules for correctness formulas. Then combination of (56) with (57) would learn us that (58) holds

(58) \[ \vdash C (\beta) \rightarrow (\alpha) \rightarrow \vdash M (\gamma). \]

Thus a complete proof system for correctness formulas would give us a complete proof system for arithmetic. Since arithmetic is not completely axiomatizable, there cannot be such a complete system \( C \) for correctness formulas. Concerning this situation De BAKKER (1980, p. 61) says the following:

"[...] we want to concentrate on the programming aspects of our language, and [...] pay little attention to questions about assertions which do not interact with [assignment] statements (so that even if an axiomatization of validity were to exist, we might not be interested in using it)."

For this reason De Bakker takes all valid assertions as axioms of \( C \), i.e. if \( \vdash M (\phi) \), then by definition \( \vdash C (\phi) \). This notion of completeness, viz. where certain assertions are taken as axioms, is called complete in the sense of Cook. For a formal definition see COOK (1978), or APT (1981). This notion is defined only for logical languages which are expressive: languages in which all strongest postconditions can be expressed (for the class of programs under consideration). From the results in 8.2 follows that our extension of IL is expressive.
In order to define the notions 'soundness' and 'completeness' for our approach, we have to find notions that can be compared with \( \models_C \) and with \( \models \delta \). First I will consider the syntactic notion \( \models_C \). In our approach the logical deductions are performed on the level of intensional logic. So if we would introduce a system \( S \) of proof rules, it would be proof rules of intensional logic. Hence we have to find an expression of IL which corresponds with (52).

We have characterized (in intensional logic) the meaning of a program \( \pi \) by means of a predicate transformer \( \pi' \), and we have proven that this transformer yields strongest postconditions, i.e. \( \mathit{sp}(\pi, \psi) = \pi'(\lambda \psi) \). Consider now (59)

(59) \( \pi'(\lambda \psi) \rightarrow \psi \).

Formula (59) expresses that if \( \psi \) holds before the execution of \( \pi \), then \( \psi \) holds afterwards. So (59) corresponds with the correctness formula \( (\psi)\pi(\psi) \).

An alternative approach would of course be to use the corresponding backward predicate transformers. The discussion below will be restricted to forward predicate transformers; for backward predicate transformers related remarks could be made. Suppose now that we have a system \( S \) of proof rules of intensional logic. The notion \( \models_S \) can be defined as usual. Then (60) says about \( S \) the same as (53) says about \( C \). Therefore I will consider (60) as the counterpart of (53).

(60) \( \models_S \pi'(\lambda \psi) \rightarrow \psi \).

In section 7 we have defined a class of models. Let \( \models \) denote the interpretation in these models. In the light of the above discussion (61) can be considered as the counterpart in our system of (54).

(61) \( \models \pi'(\lambda \psi) \rightarrow \psi \).

A system of proof rules for IL is called sound if for all \( \psi, \psi \) and \( \pi \) (62) holds.

(62) \( \models_S \pi'(\lambda \psi) \rightarrow \psi \) implies \( \models \pi'(\lambda \psi) \rightarrow \psi \).

A system \( S \) of proof rules is called complete if for all \( \psi, \psi \) and \( \pi \) (63) holds

(63) \( \models \pi'(\lambda \psi) \rightarrow \psi \) implies \( \models_S \pi'(\lambda \psi) \rightarrow \psi \).

We might consider again trivial program \( \alpha \), trivial condition \( \beta \), and an arbitrary IL formula \( \delta \). Then (64) holds

(64) \( \models \alpha'(\lambda \beta) \rightarrow \delta \) if and only if \( \models \delta \).
Suppose now that proof system $S$ contains modus ponens. Then (65) holds

$$\models_S \alpha' (\langle \delta \rangle) \rightarrow \delta \text{ if and only if } \models_S \delta.$$  

Suppose moreover that $S$ is complete. Then from (64) and (65) it follows that (66) holds

$$\models_S \delta \text{ if and only if } \models_S \delta.$$  

Thus a complete system of proof rules would give us a complete axiomatization of IL. Such an axiomatization does not exist (see chapter 3). Hence $S$ cannot be complete either. In this situation we might follow De Bakker, and make the notion of completeness independent of the incompleteness of the logic we use. So we might take all formulas of our extension of IL as axioms. But then $S$ is complete (in the sense of Cook) in a trivial way since all correctness formulas are formulas of our extension of IL.

This completeness result is not very exciting, and one might try to find another notion of completeness. A restriction of the axioms to only arithmetical assertions seems me to be unnatural for the fragment under discussion because our programs do not only deal with natural numbers, but also with pointers of different kinds. From a logical viewpoint it is attractive to try to prove for our extension a kind of generalized completeness (see chapter 3). This would require that Gallin's axiom system for IL (see chapter 3) is extended with rules concerning state-switchers. Thus we might show that a system $S$ is generalized complete, i.e. that it is complete with respect to the formulas which are true in all generalized models. The models defined in section 7 constitute a subclass of the set of generalized models. I do not know any reason to expect that the formulas valid in all models of this subclass are the same as those valid in all generalized models (because our subclass does not contain an important class: the standard models). Hence generalized completeness would be an interesting result that proves a certain degree of completeness, but it would not correspond with the traditional completeness results in computer science. I doubt whether computer scientists would be happy with such a completeness result.

Another concept between 'incomplete' and trivially 'complete', is suggested by Peter van Emde Boas. The formula $\pi' (\langle \phi \rangle) \rightarrow \psi$ was intended to be the analogue of the Hoare formula $\{ \phi \} \pi (\psi)$. The language in which we express $\phi$ and $\psi$ contains state switchers, but in most cases a programmer will be interested in cases were $\phi$ and $\psi$ are state-switcher free. However, our
analogue of the Hoare formula, viz. \( \pi' (\langle \psi \rangle) \Rightarrow \psi \), will always contain a state-switcher introduced by the predicate transformer \( \pi' \). Now one might hope for a result which says that this state-switcher can always be eliminated. In the examples we described this was indeed the case. There are however situations where no reduction rule is applicable (if values of pointers are involved, where these values are unknown). This makes it unlikely that it will always be possible to eliminate the state-switcher from a formula obtained by application of a predicate transformer to a state-switcher free formula (i.e. such an expressibility result is not to be expected). It would however, be interesting to know whether the reduction formulas are sufficient to eliminate the state-switchers from those translations of Hoare formulas where elimination is possible. This gives the following intermediate concept of 'completeness'.

If \( \phi \) and \( \psi \) are state-switcher free and \( \models \pi' (\langle \psi \rangle) = \psi \) then \( \models S \pi' (\langle \psi \rangle) = \psi \).

9. THE BACKWARD APPROACH

9.1. Problems with Hoare's rule

Besides the approach discussed up till now, there is the approach based on backward predicate transformers. In section 3 we have already met Hoare's rule for the assignment statement

(67) \( (\delta/v) \psi \) := \( \delta \{ \psi \} \).

Hoare's rule may yield incorrect results when applied to assignment containing pointers or arrays, just as was the case with Floyd's rule. I mention three examples.

De BAKKER (1976) presents for Hoare's rule the following example

(68) \( (1/a[a[2]])(a[a[2]]=2) \) \( a[a[2]] := 1 \ (a[a[2]]=2) \).

The precondition in (68) reduces to \( 1=1 \). That would imply that, for any initial state, the execution of \( a[a[2]] := 1 \) has the effect that afterwards \( a[a[2]] = 1 \) holds. This is incorrect (consider e.g. an initial state satisfying the equality \( a[2]=2 \land a[1]=2 \)).

GRIESS (1977) presents the following example

(69) \( 1=a[j] \) \( a[i] := 1 \ (a[i]=a[j]) \).

Whereas in example (68) the obtained precondition was too weak, in the present example the obtained precondition is too restrictive. The postcondition holds also in case the initial state satisfies \( i=j \).
An example of the failure of Hoare's rule for the treatment of pointers is (JANSSEN & VAN EMDE BOAS 1977b):

\[(70) \{x=x+1\} p := x; \{p=x+1\} x := x+1 \{p=x\}.\]

It is impossible to satisfy the precondition mentioned in (70), whereas for any initial state the postcondition will be satisfied.

Besides the objection that (67) gives incorrect results in certain cases, the same more fundamental problems arise as were mentioned in section 3 for Floyd's rule (e.g. the use of textual substitution).

9.2. Backward predicate transformers

Using a state switcher a formulation can be given for the backward predicate transformers which satisfies our algebraic framework. The transformer corresponding to \( \nu := \delta \)

\[\lambda \varphi[\{\delta'/\nu\}] \nu.\]

The transformer corresponding with \( a[\beta] := \gamma \)

\[\lambda \varphi[\{\lambda n \text{ if } n = \beta' \text{ then } \gamma' \text{ else } \gamma \} a[n] \varphi]/\gamma' \nu] \nu.\]

9.1. EXAMPLE. Assignment \( a[i] := 1 \); Postcondition \( a[i] = a[j] \)

Precondition: \( \{\lambda n \text{ if } n = i \text{ then } 1 \text{ else } a[n]\} i \varphi/a\}(\nu a[i] = a[j]) \)

reducing to:

\( 1 = (\text{if } j = i \text{ then } 1 \text{ else } a[j] \varphi)[j] \)

and further to: \( j = i \lor a[j] = 1 \) (compare this with (69)).

9.2. EXAMPLE. Assignment \( a[a[2]] := 1 \); Postcondition \( \nu a[a[2]] = 1 \). 

Precondition: \( \{\lambda n \text{ if } n = a[2] \text{ then } 1 \text{ else } a[n] \} i \varphi/a\}(\nu a[a[2]] = 1) \).

We have to apply the state switcher to both occurrences of \( \nu a \) in the postcondition. If we apply it to \( \nu a[a[2]] \) then we obtain

\( \text{if } a[2] = a[a[2]] \text{ then } 1 \text{ else } a[a[2]] \) \varphi.

This leads us to consider the following two cases.

I. \( a[2] = a[a[2]]. \)

Then the precondition reduces to

\( \{\lambda n \text{ if } n = a[2] \text{ then } 1 \text{ else } a[n] \} \varphi[a[2]] = 1 \) which reduces to \( a[1] = 1 \).

II. \( a[2] \neq a[a[2]]. \)

Then the precondition reduces to

\( \{\lambda n \text{ if } n = a[2] \text{ then } 1 \text{ else } a[n] \} \varphi[a[2]] = 1 \) which reduces to \( 1 = 1 \).
So the precondition is \( \forall a[2] = 2 \land \forall a[1] = 1 \lor (\forall a[2] \neq 2) \). (Compare this result with (68)).

9.2. END

9.3. Weakest preconditions

We aim at obtaining backward predicate transformers which yield a result that is correct with respect to the operational semantics", and which require assumptions as weak as possible about the initial state (i.e. dual to the requirements concerning the forward predicate transformers). The relevant notions are defined as follows.

9.3. DEFINITION. A backward predicate transformer \( \pi \) is called correct with respect to a program \( \pi \) if for all state predicates \( \phi \) and all states \( s \)

if \( s \models \pi(\beta) \) then \( \pi''(s) \models \phi \).

9.4. DEFINITION. A backward predicate transformer \( \pi \) is called minimal with respect to a program \( \pi \) if for all pairs of state predicates \( \eta \) and \( \phi \), the following holds:

if for all states \( s: s \models \eta \) implies \( \pi''(s) \models \phi \),

then \( \models \eta \rightarrow \pi(\beta) \).

9.5. THEOREM. Let \( \pi \) be a program, and \( \pi_1 \) and \( \pi_2 \) be backward predicate transformers which are correct and minimal with respect to \( \pi \). Then for all \( \phi \):

\( \models \pi_1(\beta) \leftrightarrow \pi_2(\beta) \).

PROOF. Since \( \pi_1 \) is correct, we have:

if \( s \models \pi_1(\beta) \) then \( \pi''(s) \models \phi \).

Since \( \pi_2 \) is minimal, from this implication follows

\( \models \pi_1(\beta) \rightarrow \pi_2(\beta) \).

Analogously we prove \( \models \pi_2(\beta) \rightarrow \pi_1(\beta) \).

9.5. END

A consequence of this theorem is that all backward predicate transformers which are correct and minimal with respect to a certain program, yield equivalent preconditions. This justifies the following definition.
9.6. **DEFINITION.** Let \( \pi \) be a program and \( \phi \) a state predicate. Then \( \wp(\pi, \phi) \) is a new expression of type \( t \), called the weakest precondition with respect to \( \pi \) and \( \phi \). The interpretation of \( \wp(\pi, \phi) \) is equal to the interpretation of \( '\pi(\wedge \phi) \), where \( '\pi \) is a backward predicate transformer which is correct and minimal with respect to \( \pi \). If a state predicate is equivalent with \( \wp(\pi, \phi) \) it is called a weakest precondition with respect to \( \pi \) and \( \phi \).

9.6. END

In 9.2 backward predicate transformers are defined for the assignment statements. We wish to prove that they yield a weakest precondition. This will not be proven in a direct way because it turns out that backward and forward predicate transformers are closely related. The one can be defined from the other, and correctness and maximality of the forward predicate transformers implicate correctness and minimality of the backward predicate transformers. These results will be proven in the next subsections.

9.4. Strongest and weakest

Strongest postconditions and weakest preconditions can syntactically be defined in terms of each other. This connection is proved in the following theorem.

9.7. **THEOREM.** Let \( Q \in \text{VAR}_{<s,t>} \) and let

(I) \hspace{1em} \exists \phi(\forall [\square [sp(\pi, Q) \rightarrow \phi]])

and

(II) \hspace{1em} \forall \phi(\square [\phi \rightarrow wp(\pi, Q)] \rightarrow Q).

Then it holds that

formula (I) is equivalent to \( wp(\pi, \phi) \), and formula (II) is equivalent to \( sp(\pi, \phi) \).

**PROOF.**

part A

I show that (I) is correct \( (A_1) \) and minimal \( (A_2) \) with respect to \( \phi, \pi \) and \( '\). From this follows that (I) is equivalent to \( wp(\pi, \phi) \).

part A

Suppose that \( s \) satisfies (I), so

\[ s \models \exists \phi(\forall [\square [sp(\pi, Q) \rightarrow \phi]]). \]
Then for some \( g \) (71) and (72) holds

(71) \( s, g \models \phi \)

(72) \( s, g \models \Box [sp(\pi, \phi) \rightarrow \phi] \).

By definition of \( sp \), from (71) follows

\( \pi'''(s), g \models sp(\pi, \phi) \).

By definition of \( \Box \) from (72) follows

\( \pi'''(s), g \models sp(\pi, \phi) \rightarrow \phi \).

Therefore

\( \pi'''(s), g \models \phi \), or equivalently \( \pi'''(s) \models \phi \).

This means that (I) is correct.

part \( A_2 \)

Suppose that for all \( s \) holds

(73) \( s \models \eta \) implies \( \pi'''(s) \models \phi \).

By definition of \( sp \) from (73) follows

\( \models sp(\pi, \eta) \rightarrow \phi \).

So for all \( s \)

\( s \models \Box [sp(\pi, \eta) \rightarrow \phi] \).

Let \( g \) be an assignment such that \( g, s \models \phi \) = \( ^\eta \). Then

\( s, g \models \Box [sp(\pi, \phi) \rightarrow \phi] \).

So (for the choice \( \phi = ^\eta \))

\( s \models \eta \models [sp(\pi, \phi) \rightarrow \phi] \).

This means that (II) is minimal.

part \( B \)

I show that (II) is correct (\( B_1 \)) and maximal (\( B_2 \)) with respect to \( \phi, \pi \), and\( '' \).

From this it follows that (II) is equivalent with \( sp(s, \phi) \).

part \( B_1 \)

Let

\( s \models \phi \).

Suppose that for variable assignment \( g \) holds

(74) \( g \models \Box [\phi \Rightarrow wp(\pi, \phi)] \).
Then from (74) follows
\[ s, g \vdash wp(\pi, \gamma) . \]

So
\[ \pi''(s, g) \vdash \gamma . \]

From (74) and (75) it follows that for all \( g \) holds
\[ \pi''(s, g) \vdash \Box [\phi \to wp(\pi, \gamma)] \to \gamma . \]

So
\[ \pi''(s) \vdash \forall g[\Box [\phi \to wp(\pi, \gamma)] \to \gamma] . \]

This means that (II) is correct.

part \( B_2 \)

Suppose that for all \( s \) holds
\[ s \vdash \phi \text{ implies } \pi''(s) \vdash \eta . \]

Then, by definition of \( wp \) it follows that
\[ \vdash \phi \to wp(\pi, \eta) . \]

Suppose that \( t \) satisfies (II), so
\[ t \vdash \forall g[\Box [\phi \to wp(\pi, \gamma)] \to \gamma] . \]

Let \( g \) be an assignment such that \( g, t \vdash Q = \eta \). Then:
\[ t, g \vdash \Box [\phi \to wp(\pi, \gamma)] \to \gamma . \]

Then from (77) and (79) follows
\[ t, g \vdash \gamma . \]

So
\[ \vdash \forall g[\Box [\phi \to wp(\pi, \gamma)] \to \gamma] \to \gamma . \]

This means that II is maximal.

9.7. END

That \( wp(\phi, \pi) \) and \( sp(\pi, \phi) \) are closely connected is also observed by RAULEFS (1978). He gives a semantic connection. Theorem 9.7 goes further, because an explicit syntactic relation is given.
9.5. **Correctness proof**

The theorem 9.7 has as a consequence that weakest preconditions and strongest postconditions can be defined in terms of each other. Now it is unlikely that the formulas with quantification over intensions of predicates are the kind of expressions one would like to handle in practice. The importance of the theorem is that given some expression equivalent with $sp(\pi, \phi)$, it allows us to prove that some expression (found on intuitive considerations) is equivalent with $wp(\pi, \phi)$. From the correctness and maximality of the predicate transformers defined in section 5 and 7, it follows that the backward predicate transformers defined in this section are correct and minimal.

9.8. **Theorem.** The following two statements are equivalent

(I) $sp(\chi := \delta, \phi) = \exists z[(z/\chi)\psi \land \chi' = (z/\chi')\delta']$

(II) $wp(\chi := \delta, \phi) = (\delta'/\chi)\psi$.

**Proof.**

part 1: (I) $\Rightarrow$ (II).

Assume that (I) holds. Then from theorem 9.7 follows:

(82) $\vdash wp(\chi := \delta, \phi) = \exists z[(z/\chi')\psi \land \chi' = (z/\chi')\delta'] \Rightarrow (\exists z[(z/\chi')\psi \land \chi' = (z/\chi')\delta']$.

So we have to prove that for arbitrary assertion $\phi$ and state $s$ holds that

(83) $s \models (\delta'/\chi')\psi$

if and only if

(84) $s \models \exists z[(z/\chi')\psi \land \chi' = (z/\chi')\delta']$.

part 2a: (83) $\Rightarrow$ (84)

Assume that (83) holds. Let $g$ be a variable assignment such that

(85) $g \models \psi = (\delta'/\chi')\psi$.

Then (due to (83)) we have

(86) $s, g \models \psi$.

In order to prove (84) we have next to prove the necessary validity of the formula mentioned after the $\square$ for this choice of $\psi$. So we have to prove that for arbitrary state $t$ (87) implies (88).

(87) $t \models \exists z[(z/\chi')\psi \land \chi' = (z/\chi')\delta']$
Let $h$ be a variable assignment for which (87) holds. Then using the iteration theorem, we find

\[(89) \ t, h \models \{z/\chi'\}\delta' \chi' \land \chi' = \{z/\chi'\}\delta'.\]

The second conjunct gives us information about the value of $\chi'$ in this state. The state switcher says that we have to interpret $\chi'$ with respect to the state where $\chi'$ precisely has that value. So the state switcher does not change the state! This means that

\[(90) \ t \models \phi.\]

So (87) implies (88), and therefore (84) holds.

**part 2b:** (84) $\Rightarrow$ (83)

Assume (84) holds. Then there is a variable assignment $g$ such that (91) and (92) hold

\[(91) \ s, g \models \phi\]

\[(92) \ s, g' \models \square [\exists z (z/\chi') \chi' \land \chi' = \{z/\chi'\}\delta'] \rightarrow \phi].\]

In (92) it is said that a certain formula is necessarily valid. Application of this to state $\langle \chi' + \delta' \rangle s$ gives

\[(93) \ \langle \chi' + \delta' \rangle s, g \models \exists z (z/\chi') \chi' \land \chi' = \{z/\chi'\}\delta'] \rightarrow \phi.\]

Let $g' \preceq g$ be such that $g'(z) = V_s(\chi')$ so $\langle \chi' + z \rangle \langle \chi' + \delta' \rangle s = s$. Since (91) holds, we have

\[(94) \ \langle \chi' + z \rangle \langle \chi' + \delta' \rangle s, g' \models \phi.\]

Consequently

\[(95) \ \langle \chi' + \delta' \rangle s, g' \models \{z/\chi'\} \phi.\]

Moreover

\[(96) \ \langle \chi' + \delta' \rangle s, g' \models \chi' = \{z/\chi'\}\delta'.\]

because $V_{\langle \chi' + \delta' \rangle s} (\chi') = V_s (\delta') = V_{\langle \chi' + z \rangle \langle \chi' + \delta' \rangle s} (\delta') = V_{\langle \chi' + \delta' \rangle s} (z/\chi') \delta'.\]

From (94) and (95) follows that the antecedent of the implication in (93) holds. Therefore the consequent of the implication holds

\[(97) \ \langle \chi' + \delta' \rangle s, g' \models \phi\]

so

\[(98) \ s, g' \models \delta' \chi'.\]
This means that (83) holds, so (84) \(\Rightarrow\) (83). And this completes the proof of (I) \(\Rightarrow\) (II).

**Part 2:** (II) \(\Rightarrow\) (I)

The proof of (II) \(\Rightarrow\) (I) uses a related kind of arguments. Therefore this proof will be presented in a more concise way. Assume that (II) holds. Then we have to prove that for arbitrary \(s\) and \(\phi:

\[(99) s \models \forall \phi[\exists \delta' (\delta' \mapsto \{\delta'/\chi'\}) \models \forall \phi]\]

if and only if

\[(100) s \models \exists \phi[\{\delta'/\chi'\} \models \exists \delta' \models \{\delta'/\chi'\}]\]

**Part 2a**

Assume (99). Take for \(\phi\) in (99) the assertion in (100). We now prove that the antecedent of (99) holds, then (100) is an immediate consequent. Suppose \(t \models \phi\). We have to prove that

\[(101) t \models \exists \phi[\{\delta'/\chi'\} \models \exists \delta' \models \{\delta'/\chi'\}]\]

or equivalently

\[(102) t \models \exists \phi[\{\delta'/\chi'\} \models \exists \delta' \models \{\delta'/\chi'\}]\]

This is true for \(g(z) = \forall \chi'(\chi')\), so the antecedent of (99) holds, and from this follows that (100) holds.

**Part 2b**

Assume (100). Let \(g\) be arbitrary and assume

\[(103) s, g \models \phi \Rightarrow \{\delta'/\chi'\} \models \forall \phi\]

This is the antecedent of (99). We now prove that the consequent holds, so that

\[(104) s, g \models \forall \phi\]

From (100) follows that for some \(g' \sim g\)

\[(105) \langle \chi' + z \rangle s, g' \models \phi\]

Using (103), from (105) follows

\[(106) \langle \chi' + z \rangle s, g' \models \{\delta'/\chi'\} \models \forall \phi\]

Consequently

\[(107) s, g' \models \{\{\delta'/\chi'\} \models \forall \phi\}

From (100) also follows
(108) \( s, g' \vdash \forall x' = \{z/z'\}\delta' \).

From (108) and (107) we may conclude

(109) \( s, g' \vdash \forall Q \).

This proves (104), so (99) follows from (100).

9.8. END

9.9. THEOREM. The following two statements are equivalent

(I) \( \vDash sp(a[b]) = \delta, \phi = \exists z([z'/a']\phi \land \forall\ a' = \{z'/a'\}\lambda n \text{ if } n = b' \text{ then } \delta' \text{ else } \forall a[n] fi \) \]

(II) \( \vDash wp(a[b]) = (\lambda n \text{ if } n = b' \text{ then } \delta' \text{ else } \forall a'[n] fi'/a)\phi \).

PROOF. The expressions at the right hand side of the equality signs are a special case of the corresponding expressions in the previous theorem. So theorem 9.9 follows from theorem 9.8.

9.9. END

From theorems 9.9 and 9.10 it follows that the predicate transformations for the assignment as defined in section 9.2, yield weakest preconditions.

10. MUTUAL RELEVANCE

In this section I will mention some aspects of the relevance of the study of semantics of programming languages to the study of semantics of natural languages, and vice versa. Most of the remarks have a speculative character.

The present chapter constitutes a concrete example of the relevance of the theory of semantics of natural languages to the study of programming languages. Montague's framework was developed for natural languages, but it is used here for programming languages. The notions 'opaque' and 'transparent', well known in the field of semantics of natural languages, turn out to be useful for the study of semantics of programming languages, see section 1. And the logic developed for the semantics of natural languages turned out to be useful for programming languages as well.

In the semantics of natural languages the principle of compositionality is not only the basis of the framework, but also, as will be shown later, a valuable heuristic tool. It helped us to understand already existing
solutions. It gives rise to suggestions how to deal with certain problems, and it is useful in finding weak points in proposals from the literature. I expect that the principle can play the same role in the semantics of programming languages. The treatment of arrays in this chapter (see section 6) is an example of the influence of the principle. Below I will give some further suggestions concerning possible relevance of the principle.

Consider the treatment of 'labels' and 'goto-statements' by A. de Bruyn (chapter 7 in De BAKKER 1980). The treatment is rather complex, and not much motivation for it is given. I expect, however, that these phenomena are susceptible to the technique explained in chapter 1: if the meaning of some statement seems to depend on certain factors, then incorporate these factors into the notion of meaning. In this way the notion of 'continuation' (used by de Bruyn) might be more easily explained, and thus the proposal more easily understood.

In De BAKKER 1980, the proof rules for certain constructions make use of devices which are, from a compositional point of view, not attractive. These constructions are assignments to subscripted array identifiers, procedures with parameters, and declarations of identifiers at the beginning of blocks. In the proof rules for these constructions mainly syntactic substitution is used. From a compositional point of view it is not surprising that the semantic treatment of these phenomena is not completely satisfactory. For assignments to array elements an alternative was proposed in section 6, and for blocks a suggestion was made in section 4. A compositional approach to the semantics of procedures with parameters would describe the meaning of a procedure-call as being built from the meaning of the procedure and the meaning of its argument. If this argument is a reference parameter (call by variable), then the argument position is opaque. This suggests that the meaning of such a procedure should be a function which takes as argument an intension.

In the semantics of natural language ideas from the semantics of programming languages can be used. The basic expression in a programming language is the assignment statement. For the computer the assignment statement is a command to perform a certain action. I have demonstrated how the semantics of such commands is dealt with by means of predicate transformers. Inspired by this approach, we might do the same for commands in natural language. Some examples (taken from Van EMDE BOAS & JANSEN 1978) are given below. Consider the imperative
(110) John, drink tea.

Its translation as a predicate transformer would become something like

(111) \( \lambda x (\mathcal{B}(x, y) \land \text{drink-tea}(x, y)) \).

This expression describes the change of the state of the world if the command is obeyed. The operator \( \mathcal{B} \) is a kind of state-switcher, it indicates the moment of utterance of the command. A similar approach can be used to describe the semantics of actions. One might describe the semantics of performative sentences like

(112) We crown Charles emperor

by means of a predicate transformer.

Often a sequence of sentences is used to perform an action rather than to make a some assertions: sentences can be used to give information to the hearer. Consider the text

(113) Mary seeks John. John is a unicorn.

These sentences might be translated into the predicate transformers (114) and (115).

(114) \( \lambda x (\mathcal{V} y \land \text{seek}_y(x, y)) \)

(115) \( \lambda x (\mathcal{V} y \land \text{unicorn}_y(x)) \).

Suppose that the information the hearer has in the beginning is denoted by \( \phi \). Then by the first sentence this information is changed into

(116) \( \phi \land \text{seek}_x(y, x) \)

and by the second sentence into

(117) \( \phi \land \text{seek}_x(y, x) \land \text{unicorn}_y(x) \).

From the final expression the hearer may conclude that Mary seeks a unicorn.

Also on a more theoretical level the semantics of programming languages can be useful for the study of semantics of natural languages. In the study of natural languages the need for partial functions often arises. In the semantics one wants to use partially defined predicates in order to deal with sortal incorrectness and presuppositions, and in the syntax one wishes to have rules that are not applicable to every expression of the category for which the rule is defined. In the field of programming languages phenomena arise for which one might wish to use partial functions. In this field techniques are used which make it possible to use nevertheless total
functions. The basic idea is to introduce in the semantic domain an extra element. Since this approach is, from an algebraic point of view, very attractive, I would like to use this technique in the field of natural languages as well (see chapter 7).