Linking Programs and Specifications in Z

Semantic Basis and Proof Structures

A THESIS SUBMITTED FOR THE DEGREE OF MSC.

James Power BSc.

School of Computer Applications
Dublin City University
Dublin 9

August 12, 1991.

Supervisor: Dr. Tony Moynihan

This thesis is based on the candidate's own work, and has not previously been submitted for a degree at any academic institution.

CONTENTS

ABSTRACT

Linking Programs and Specifications in Z

by James Power

This thesis uses the Z specification language notation to discuss the relationship between programs and specifications. We give a brief introduction to formal semantics, and the Z specification language. We provide an adaptation of the Z schema calculus which is based on predicates rather than propositions: this is constructed from the standard calculus and is thus intended as an enhancement rather than a replacement.

We discuss the process of refining a specification towards a program as an exposition of the relationship between the Z language (in particular) and standard programming languages. We formulate the standard refinement proof obligations in our calculus, and briefly examine some applications.

The focal point of the thesis is the provision of the semantics for a small language in Z. We believe our approach to be unique, in that we have not only described the semantic mapping functions using Z, but we are also taking Z to be our semantic domain; that is, our semantics will map a program into a corresponding Z specification. We assert the usefulness of such a specification as a basis for the analysis of the program.

The semantics are analyzed in detail, and further work is done towards the unification of various aspects of program analysis under the Z notation, and in particular, under our predicate version of the schema calculus; the use of this notation and calculus to explore the link between programs and specifications is a dominant theme throughout the thesis.

Using the same structures to describe programs and specifications is becoming increasingly common; in the light of this we claim that constructing a link between one and the other should not be considered solely in directional terms; the techniques in moving from the former to the latter are likely to be just as useful when moving in the opposite "direction".

We present a number of small examples to illustrate the above concepts.

CHAPTER 1 - DESCRIBING PROGRAMS

1.1 Importance of the Program-Specification Link

Let us define, for the moment, a specification as being some formal description of the properties of a program, or of a system which that program is supposed to implement. We can identify three main reasons why we would want to study the link between such descriptions and their corresponding programs: if we have a specification and want a corresponding program, if we have a program and need a corresponding specification, or if we have both, but are unsure that they actually correspond.

1. Developing programs

The task of developing a program to solve some problem will conventionally start with some description of what that program is expected to do. Any program we develop will be a formal, unambiguous, and hopefully correct description of the solution to the problem. Characterising the specification in equally unambiguous terms will be of obvious benefit for the analysis of that specification, but also as a starting point for the development of a program: if we have the start point (the specification) and the end point (the program) of the development process expressed formally, then it makes sense to adhere to such formality as we attempt to develop the latter from the former. Such a formalised description of the development can then be examined to ensure that it has proceeded correctly: the program is thus verified by asserting the correctness of the development steps.

2. Documenting Programs

If a program can be seen as a formal description of the solution to the problem, then it can also be seen as the documentation which explains that solution; unfortunately this is often the case. The nature of an explanation of a program will naturally depend on why that explanation was required, but it is a safe bet that a description other than the program itself will be useful. A formal specification provides a "summary" of the program, from which some the detail has been abstracted away; this form of description can be used to assess the usefulness of the program for a particular task, or as a basis for altering the program. We would hope that studying the link between programs and specifications will provide a "ladder" which we might use to ascend from programs to their specifications.

3. Certifying programs

Even if we have been given a description of what the program is supposed to do, there is no guarantee that this description is correct and does not contain inaccuracies (accidental or otherwise). The idea of software certification envisages some sort of independent body which would be able to "approve" the program as one which fully realises the claims made for it in its documentation. If a formal specification can be constructed from whatever form of documentation is provided, then the program-specification link can be seen as a common platform, from which the relative features of the program and the putative specification can be compared.

1.2 The Purpose of Formal Semantics

If we wish to analyze a program, then we will want to have a fairly definite idea of its exact meaning; in order to have this, we must fully understand the programming language in which it is written. Since we will be exploring the connection between programs and formal specifications we will want a description of the programming language in formal mathematical terms which we can reason about. This type of formal description of a language is referred to as the formal semantics of that language; it usually involves describing elements of the syntax of the language in terms of some set paradigm.

One of the earlier semantic techniques was operational semantics, which involved describing an abstract machine, and then characterising the program in terms of some sequence of actions which would be performed by the machine. This method was prominent in the early days of semantics (based on the success of the idea for dealing with program complexity issues), but will be less of interest to us than its more "abstract" counterparts. We will be considering two of these approaches - the denotational and axiomatic method.

1.3 Denotational Semantics

The denotational method (also known as mathematical semantics) is based on a significant amount of mathematical theory, investigation of which was pioneered by Dana Scott; for our purposes we will only deal with this theory superficially, but a fuller description of the underlying principles can be found in Scott's article in [BrSc82], or in chapter 5 of [Mann74] (or in books dealing with denotational semantics such as [Schm86]).

We will be describing program constructs by means of their actions modelled in some group of semantic domains, which we can regard as being sets with particular properties We usually start out with some basic group of domains, such as the Integers, Booleans, Natural Numbers, etc. We can then build on these domain by defining functions, tuples, unions, sequences and so on over existing domains.

Providing a semantics for a language will consist of giving a number of functions which map constructs of (the syntax of) the language into elements of some group of pre-defined domains. These functions are usually defined recursively over the language; the semantics of some construct is defined as some form of composition of the semantics of its sub-components. This emphasis on the denotation of the constructs gives the semantics its name; the style of depicting a component in terms of its sub-components is the hallmark of denotational semantics.

We would usually start the definition of the semantics of a language with some sort of formal definition of the syntax; these syntactic domains will then provide the basis for our mapping. The simplest way to describe this process is to take a small example of a programming language[1], and to attempt to formulate its semantics.

[1]Based on the descriptions in [Gord79] chapters 4 and 6, [Paga81] § 4.2, and [Stoy77] chapter 9.

1.3.1 Denotational Semantics for a (very) small language

Syntactic Domains

We will need consider just five syntactic domains:

Name	Description	Sample Members	Type
Ident	Identifiers	I	Primitive
Bas	Basic constants	B	Primitive
Opr	Binary Operators	O	Primitive
Cmd	Commands (Statements)	C, C_1, C_2	Compound
Exp	Expressions	E, E_1, E_2	Compound

If we assume that our language operates over the natural numbers, then the domain *Bas* will consist of the constants {0, 1, 2, ...}, and the operators in *Opr* will be +, -, =, < and so on.

The domains *Ident*, *Bas*, and *Opr* are primitive domains, whereas *Cmd* and *Exp* are compound domains, and are defined by the equations:

$$C ::= \text{skip} \mid I := E \mid \text{if } E \text{ then } S_1 \text{ else } S_2 \mid \text{while } E \text{ do } S \mid S_1;S_2$$
$$E ::= \text{True} \mid \text{False} \mid \text{not } E \mid I \mid B \mid E_1 \; O \; E_2$$

Semantic Domains

We will need just two primitive semantic domains: ℕ the Natural Numbers, and *Bool* the Boolean values *TRUE* and *FALSE*. To simplify things we will use the same identifiers for both the syntax and the semantics, and so we need define just two compound semantic domains, *Dval* and *State*.

Name	Meaning	Sample Members	Type
ℕ	The Natural Numbers		Primitive
Bool	Boolean Values		Primitive
Dval	Denotable values	d	Compound
State	Machine States	s	Compound

The compound domains are defined by the equations:

$$\text{Dval} = [\mathbb{N} + \text{Bool} + \{\omega\}]$$

$$\text{State} = \text{Ident} \rightarrow \text{Dval}$$

Thus the program state is simply a mapping from identifiers to their current value, or to ω if they are undefined.

Semantic Functions

Before we define the semantic functions, we will first need two expressions which will prove useful. For some function $f : [A \rightarrow B]$ and domain elements $a : A$ and $b : B$, the update expression $f \oplus (a \mapsto b)$ returns a function which is the same as f except that a is mapped to b. If a and b are elements of the same domain, then the conditional expression $t \rightarrow a, b$ evaluates to a if t is true, or b if t is false[2].

For any two functions f and g, we use $f \circledcirc g$ to denote the application of f followed by the application of g; if the result of applying f was undefined, then so will $f \circledcirc g$.

Now that we have defined the syntactic and semantic domains, we can define a mapping between them using just three semantic functions, $\mathfrak{R}$, $\mathfrak{C}$ and $\mathfrak{E}$ which are defined as follows:

$\mathfrak{C}$: Cmd $\rightarrow$ [State $\rightarrow$ State]

$$\mathfrak{C}\,[\text{skip}] = \lambda\, s.\, s$$

$$\mathfrak{C}\,[I := E] = \lambda\, s.\, (s \oplus (I \mapsto \mathfrak{E}[E]s))$$

$$\mathfrak{C}\,[\text{if } E \text{ then } S_1 \text{ else } S_2] = \lambda\, s.\, (\mathfrak{E}[E]s \rightarrow \mathfrak{C}[S_1]s, \mathfrak{C}[S_2]s)$$

$$\mathfrak{C}\,[\text{while } E \text{ do } S] = \lambda\, s.\, (\mathfrak{E}[E]s \rightarrow \mathfrak{C}[S]s \circledcirc \mathfrak{C}[\text{while } E \text{ do } S], s)$$

$$\mathfrak{C}\,[S_1;S_2] = \mathfrak{C}[S_1] \circledcirc \mathfrak{C}[S_2]$$

$\mathfrak{E}$: Exp $\rightarrow$ [State $\rightarrow$ Dval]

$$\mathfrak{E}[\text{True}] = \lambda\, s.\, \mathit{TRUE},$$

$$\mathfrak{E}[\text{False}] = \lambda\, s.\, \mathit{FALSE}$$

$$\mathfrak{E}[\text{not } E] = \lambda\, s.\, (\neg\, \mathfrak{E}[E]s)$$

$$\mathfrak{E}[I] = \lambda\, s.\, I$$

$$\mathfrak{E}[B] = \lambda\, s.\, \mathfrak{R}\,[B]$$

$$\mathfrak{E}[E_1\ O\ E_2] = \lambda\, s.\, \mathfrak{R}[O]\,(\mathfrak{E}[E_1]s, \mathfrak{E}[E_2]s)$$

---

[2]We make the additional assumption that if any of the arguments to the update or conditional expression are undefined, then the result will also be undefined.

We shall not describe the function $\mathfrak{R}$ in detail; it is a trivial mapping from the constant and operator symbols of the program into the corresponding symbols of the semantic domain.

Thus, a command is regarded as being a state to state mapping (ie. something which can possibly change the values that the identifiers are mapped to), while an expression simply represents some value which can be calculated from the current state.

One of the main points of interest here is the definition of the while loop by recursion. If we let:

$$W = \mathfrak{C}[\text{while E do S}]$$
$$F = \lambda\, w.\, (\lambda\, s.\, (\mathfrak{E}[E]s \rightarrow \mathfrak{C}[S]s \circ w,\ s))$$

then the definition of the while loop becomes the equation $W = F(W)$, which asserts that the meaning of the while loop is the fixpoint of the function F. To examine this definition in greater detail would involve going further into the mathematical background of denotational semantics than we will need; suffice to say that the uniqueness of the solution is guaranteed by choosing the least fixpoint, while the existence of a solution will depend on the monotonic properties of the functions.

1.3.2 Dealing with scope and parameters

One obvious omission from the above language is a provision for the declaration of identifiers representing variables, as is common in most languages. The notion of the scope of the variable is closely connected with how we deal with its declaration, and will be fundamental to parameter passing and dealing with procedure calls in general. We will just look briefly at a common approach.

If we wish to treat the variables which an identifier represents as being dependent on the current context, then we must first of all make the link between identifiers and the values which they denote a little bit more flexible. We can do this by choosing to define this link in terms that would perhaps be somewhat close to the way in which a machine might view it; in terms of memory locations. Let us we have some arbitrarily large domain of memory locations called *Loc*; the we can define the memory, or store, by the function:

$$\text{Store} = \text{Loc} \rightarrow \text{Dval}$$

Each identifier is then associated with a particular "address" in the store; we get the corresponding value by examining the contents of this address. We use an environment function *Env* to define this mapping:

$$\text{Env} = \text{Ident} \rightarrow \text{Loc}$$

Thus, in order to find the value of any identifier, we will have to first work out the corresponding location using the environment, and then use the store to map this to a value; this means in practice that we will now have to introduce *Env* as an extra parameter to our semantic functions. We can now distinguish between an identifier's appearance on the left-hand-side and on the right-hand-side of an assignment statement; the former will represent a location, the latter will represent a value.

The environment allows us to model issues concerning scoping during a procedure call.

Each time the procedure executes, the environment will be augmented with mappings for the local variables for that procedure. However, we will also be able to represent the situation where two identifiers both reference the same location (as in the case of the call-by-name parameter mechanism); we can just map both identifiers to the same element of *Loc*. Thus the environment is a many-to-one mapping from identifiers to locations, while the store will be a many-to-one mapping from locations to values.

1.3.3 Meaning Functions

The last topic we will consider in our discussion of denotational semantics is the idea of a meaning function. We have defined a program in terms of functions over domains; the purpose of a meaning function is to pick out some aspect of this definition which we will regard as being the "meaning" of the program as far as we are concerned. We choose to highlight three such aspects[3]:

1. $\mathrm{M}_{\mathrm{ALL}}$

Each command will involve some operation which can change the value of any identifier which is currently in scope. In a manner reminiscent of § 1.3.1 we could characterise the state of the program at any instant as a set of pairs of identifiers and values. Commands can change the values, while declarations and procedure calls can change the identifiers (and the number of identifier-value pairs). One way of defining the meaning of a program therefore would be simply to list the sequence of states which would correspond to an execution of the program. This type of meaning is perhaps closest to the operational approach, and we shall denote it as M_{ALL}, since it effectively captures all

[3]We do not claim, of course, that these are the only three aspects, or even that they must be exclusive.

of the information that the program has to provide; the level of abstraction here is thus at a minimum.

2. M_{END}

A description which lacks in abstractness to the degree of M_{ALL} has the disadvantage that it may distinguish between programs which we would like to regard as identical. We might only want to know what a program "does", and choose to characterise this in terms of the values of the identifiers upon termination; we will call this M_{END}. What we are really concentrating on here is what the program has done, rather than how it has done it. This approach is more abstract that the preceding one, since M_{END} will not distinguish between two programs which use different methods to achieve the same results, while M_{ALL} would.

3. M_{IO}

We did not deal with input and output commands in our discussion so far, but it is clear that they are fundamental to any program that we might hope to execute on a machine. One natural way to characterise a program is on its observable behaviour; that is, on how its outputs relate to its inputs - we call this M_{IO}. From the point of view of the semantics, dealing with input and output may involve introducing special I/O domains, and passing these between semantic functions along with the store; we could then pick out some properties of these domains as being a suitable characterisation of the input/output behaviour of the program.

1.4 Axiomatic Semantics

Axiomatic semantics are usually regarded as operating at a more abstract level than either denotational or operational semantics. A language is defined by including its constructs in a proof theory, based on mathematical logic; this involves adding structures for describing the language components, and of adding new deduction rules to standard logic. One of the first methods of achieving this was by means of Hoare triples in [Hoar69], with a definition of the Pascal language following in [HoWi73]. A variation of this method, using a predicate transformer called *wp*, was detailed in [Dijk76]; we will give a brief outline of both methods.

1.4.1 Hoare's Rules

The basis of these rules is the triple *{P} S {Q}*, where *P* and *Q* are predicates, and *S* is a statement, or group of statements[4]. The predicates assert some property of the program variables, and correspondingly, the triple asserts some property of the statements *S*. We regard *P* as describing the relationship between the variables before the execution of *S*, and *Q* as describing the relationship after *S* has been executed. When we assert that *{P} S {Q}*, we are saying that if the statements *S* are executed starting from a situation when the variables satisfy *P*, then the termination of *S* will result in the variables being given values which satisfy *Q*.

We do not insist that the statements *S* will terminate; nothing is asserted about the situation which occurs when they don't. The assertion that *{P} S {FALSE}* states that *S*, when executed from a situation where *P* holds, will never terminate (since no assignment of values to the program variables can ever satisfy *FALSE*). If we claim that

[4]This triple is also sometimes written as *P {S} Q* with the same meaning for *P*, *S* and *Q*

{TRUE} S {Q} holds, then we are effectively characterising the final values of the variables after any terminating execution of *S* by the predicate *Q*. For the triple *{P} S {Q}*, let us refer to *P* and *Q* as the pre- and post-condition respectively, and regard the assignment of values to the program variables as constituting a program state.

If we assert *{P} S {Q}*, then we require from *Q* that any post-condition resulting from a pre-condition characterised by *P* can be characterised by *Q*; we do not care if extra states, even if they do not correspond to any pre-conditions, are also characterised by *Q* so long as it contains the ones we want. Therefore we can feel free to either strengthen *P* (and possibly reduce the set of possible corresponding post-conditions), or we can weaken *Q*, without changing the validity of the assertion. Using the turnstile symbol $\vdash$ to represent deduction, we can thus formulate rules for the system such as:

$$\{P\}\ S\ \{R\},\ R \Rightarrow Q \quad \vdash \quad \{P\}\ S\ \{Q\}$$

and

$$P \Rightarrow R,\ \{R\}\ S\ \{Q\} \quad \vdash \quad \{P\}\ S\ \{Q\}$$

Various other rules exist which allow the formulae to be manipulated in this manner.

We can now define the statements of the program using these triples, by describing the effect that the statements have on arbitrary predicates. We define the most basic command, the assignment statement *x := t* by means of an axiom:

$$\{P[t/x]\}\ x := t\ \{P\}$$

We use *P[t/x]* to denote the predicate *P* which has had all free occurrences of the variable *x* replaced by the variable *t*. The assignment statement is thus regarded as a means of taking the properties which were true for *t*, as represented by *P*, and asserting that they now hold true for *x*.

Compound statements are represented by deduction rules, whose prerequisites usually assert properties of the component statements.

The composition rule allows us to assert properties of the sequential composition of two statements based on a common "intermediate" state:

$$\{P\}\ S_1\ \{R\},\ \{R\}\ S_2\ \{Q\}\ \vdash\ \{P\}\ S_1;S_2\ \{Q\}$$

The if-then-else rule is equally straightforward, picking out a pre-condition for the statement which is a specific weakened form of the pre-condition for the two branches of the statement:

$$\{P \wedge E\}\ S_1\ \{Q\},\ \{P \wedge \neg E\}\ S_2\ \{Q\}\ \vdash\ \{P\}\ \textbf{if}\ E\ \textbf{then}\ S_1\ \textbf{else}\ S_2\ \textbf{fi}\ \{Q\}$$

The rule for the while statement depends on selecting a suitable strengthening of the negation of the guard which will remain invariant for the execution of the loop body:

$$\{P \wedge E\}\ S\ \{P\}\ \vdash\ \{P\}\ \textbf{while}\ E\ \textbf{do}\ S\ \textbf{od}\ \{P \wedge \neg E\}$$

The rules for procedure statements is not unlike that for the assignment in that it also depends on variable substitution; we will not discuss here the various different cases which correspond to the different types of parameter mechanisms. Suffice to say that most of the detail involved in describing the rules involves ensuring that there are no name clashes brought about by the replacements; a number of rules are discussed in [Apt81].

1.4.2 Dijkstra's wp

A more commonly-used variant of the axiomatic method is the *wp* predicate transformer (references include [Dijk75], [Dijk76], [Grie81], [Drom89]) which defines the relationship between a terminating program, a post-condition, and their weakest precondition. We write *wp(S,Q)* for statements *S* and post-condition *Q* to specify the weakest pre-condition that must be true in order for the program to terminate and establish *Q*. Note that we are interested in partial correctness, since we deal only with programs which terminate, unlike the Hoare rules which can be applied to non-terminating programs. Thus, if we know that *S* is guaranteed to terminate for every input, the assertion that *{P} S {Q}* is equivalent to saying that $P \Rightarrow wp(S,Q)$ (since any valid pre-condition will always be contained in the weakest pre-condition).

Since we will meet *wp* again in later chapters, it will be useful to elaborate on four of its characteristic properties:

1. For any statements *S* we have *wp(S,FALSE) = FALSE*. This is known as the Law of the Excluded Miracle. In the previous section we identified the post-condition *FALSE* with non-termination; what we are saying here is that the *wp* of any group of statements will never include a state that causes the statements not to terminate.

2. For any *S* and post-conditions *Q* and *R* such that $Q \Rightarrow R$, we have $wp(S,Q) \Rightarrow wp(S,R)$, ie. *wp* is monotonic with respect to implication (or, as we weaken the post-condition,

we also weaken the corresponding weakest pre-condition). The weakest possible post-condition is *TRUE*, (since for any R, $R \Rightarrow TRUE$), and so $wp(S,TRUE)$ will contain all those states which, for any R, satisfy $wp(S,R)$ (in other words, it characterises the set of states for which S will halt).

3. The transformer *wp* is distributive with respect to conjunction and disjunction; ie. for S, Q and R as above we have: $wp(S,Q) \wedge wp(S,R) \Leftrightarrow wp(S,Q \wedge R)$ and also that: $wp(S,Q) \vee wp(S,R) \Leftrightarrow wp(S,Q \vee R)$. Note that the equivalence of the latter assertion depends on the determinicity of the statements S.

4. If we have some non-deterministic set of statements N, then we may be able to talk about the weakest pre-condition for some post-condition $(Q \vee R)$ if this predicate represents the aggregate results from choosing any of the non-deterministic routes. However, there is absolutely no guarantee that we can divide those pre-conditions into two groups, one of which will be guaranteed to establish Q and the other of which will be guaranteed to establish R, since the notion of "guaranteeing" the result may infringe upon the non-determinicity of the choice. Thus for any non-deterministic statements N, we have $wp(N,Q) \vee wp(N,R) \Rightarrow wp(N,Q \vee R)$

Now that we have an alternative to using the Hoare triples, we can begin to give a semantics for parts of a programming language. We can define the assignment statement in a corresponding manner to the definition of the previous section, ie:

$$wp(x := t,P) = P[t/x]$$

Sequential composition is equally straightforward:

$$wp(S_1;S_2,R) \;=\; wp(S_1,\, wp(S_2,R)$$

The language commonly used with the wp calculus utilises the idea of guarded commands, where a sequence of statements is executed only if their preceding guard is true; for guard B and statements S, this is usually written $B \rightarrow S$. This can then be used to construct the familiar conditional statement, but this is further generalised to the non-deterministic version[5],

$$\text{if } B_1 \rightarrow S_1 \lozenge B_2 \rightarrow S_2 \lozenge \ldots \lozenge B_n \rightarrow S_n \text{ fi}$$

where the statement to be executed is chosen non-deterministically from those statements whose guard is true (execution "halts" if none of them are true). Let *IF* denote the above statement and let *BB* denote $(B_1 \vee B_2 \vee \ldots\ldots \vee B_n)$, so that we can define a semantics for the conditional statement as:

$$wp(IF,R) \;=\; BB \wedge (B_1 \Rightarrow wp(S_1,R)) \wedge (B_2 \Rightarrow wp(S_2,R)) \wedge \ldots \wedge (B_n \Rightarrow wp(S_n,R))$$

The pre-condition *BB* is necessary to ensure that at least one of the guards will be true, and we also assert that the truth of any guard must imply that the corresponding statement produces the required result.

The iterative command is structured similarly:

$$\text{do } B_1 \rightarrow S_1 \lozenge B_2 \rightarrow S_2 \lozenge \ldots \lozenge B_n \rightarrow S_n \text{ od}$$

[5]We use the symbol $\lozenge$ to represent the choice: the standard rectangular symbol does not appear to be in our character set!

where the loop body is repeatedly executed until all the guards are false; a single iteration corresponds to the *if* .. ◊ .. *fi* statement above. Let us denote the above statement by *DO*, and define the functions $H_i(R)$ as:

$$H_0(R) \;=\; (R \wedge \neg BB)$$

$$H_i(R) \;=\; H_0(R) \vee wp(IF, H_{i-1}(R)) \qquad \text{for any } i > 0$$

Each $H_i(R)$ corresponds to the situation where the loop iterates *i* times and then terminates. The while loop will then terminate if we can find just one such *i* for which this holds:

$$wp(DO,R) \;=\; (\exists\, i : \mathbb{N} \cdot H_i(R))$$

In practice it is easier to describe a loop by asserting that some predicate *P* is invariant for the loop body, and to prove termination by showing that some bound function β is decreased by each loop iteration and reaches a minimum when the loop guard is false. Let us assume for simplicity that β is a function over the program variables which returns an element of ℕ. If for each guard B_i and corresponding statement S_i in the loop body, we can show that:

$$P \wedge B_i \;\Rightarrow\; \exists\, t : \mathbb{N}_1 \cdot (\beta = t) \wedge wp(S_i, P \wedge (\beta < t))$$

then this implies that $P \Rightarrow wp(DO, P \wedge \neg BB)$.

1.5 Program Annotation

1.5.1 What is involved

The major benefit in axiomatising a programming language is that it facilitates the formal derivation of programs from a given specification. The ideal is that we should manipulate the initial specification into some form which is then amenable to expression in terms of one of the programming commands; various techniques, examples and heuristics are given in text such as [Grie81], [Back86] and [Drom89]. The end result of such a process would be a program whose derivation has been verified, and which has been annotated accordingly with various predicates which are deemed to hold true at a certain point in the program.

However, if a program has not been developed using formal derivation techniques (or if details of this derivation are no longer available) then in order to deal with it in an axiomatic framework, we are faced with the task of providing predicates to annotate the program based solely on the actual program code. This involves using the semantics of the programming language to deduce some predicate which describes the relationship between the variables at a particular point in the program, and attempting to work out what effect ensuing commands will have on this predicate. The best description of the program will be the one which uses the strongest possible (valid) predicates to describe the program state.

We essentially have three groups of rules to deal with when attempting to analyze a program in this manner. First of all, we must consider the rules of the semantic framework in which we are working; for our purpose these rules will be the familiar rules of standard mathematical logic. Other situations might involve different proof structures: for example, if we were attempting to analyze a system with concurrency facilities, we might choose to work in the realm of temporal logic.

Secondly, we will have the rules associated with the particular programming language that we are using. As we have seen, this may involve an augmentation of the basic rules of the semantic framework to incorporate the individual commands of the language. We can thus regard the program as an "assertion" in some form of logic, from which we can draw various consequences, depending on each of the individual statements, and groups of statements. When dealing with a proof in standard logic, we will use certain conventions to document our progress. Usually we will start off with the facts that we are given, and then we will write any deductions that we may make one after the other, perhaps giving some clue as to how they were arrived at. Thus we assume that an assertion at any point in the proof depends on what has gone before. In the same way a program can be regarded as such a structure, since we may write an assertion at a particular point in the program text, and feel justified in deducing that a some other assertion will hold further on in the program text, based on the semantic rules that we have defined for the relevant programming language.

Our third set of rules involves the data types used in the programs. When these are modelled in the semantic domain, we do not consider them to be just a set of named elements and operations, but we also assume that ceratin properties hold true for them. This may be done implicitly if we are using familiar domains (such as Integers or Natural numbers), or we may choose to provide some form of axiomatisation to aid in the description of the domain. When we come to prove properties of any program which uses a type corresponding to one of these domains, we will want to exploit these properties; for example, if we were considering the Integers, then we will expect to be able to use the transitivity property of equality in our proofs (that is, for integers a, b

and c, we have $(a = b) \wedge (b = c) \Rightarrow (a = c)$). Some algebraic specification techniques provide methods for representing such properties formally - very often in terms of abstractions such as groups and rings.

We can thus regard the program annotation process of being one where we are provided with the above three groups of rules, and an assertion based on these rules (ie. the program), and asked to deduce some set of facts; the particular emphasis that we will place on some facts over others will depend on how we represent the meaning of a program (as in § 1.3.3 above).

1.5.2 Annotation Rules

It is worthwhile to illustrate the above discussion (and to motivate some of the work we do in chapter 4 of this thesis) by considering an attempt to systematise the annotation process described in [Mann80] and [Ders83].

The types of annotations that we may write in a program text are divided into three categories. For some predicate *P*, the annotation *assert P in Prog* is called a global invariant, and it describes some properties of the variables which are invariant throughout the section of the program which corresponds to *Prog*. This can be used to characterise some property which has been established by another section of the program and upon the validity of which the operation of *Prog* has been based; for example, after an input statement with some form of input-validation procedure, we might assert that the input will consequently possess the properties established by this validation.

The second type of annotation is the local invariant, which is of the form *L: assert P* where *L* is some position (or label) in the program. This annotation asserts that the predicate *P* is true whenever execution reaches that point in the program. We could use this type of assertion to describe a loop invariant, by annotating the program with the predicate which represents the invariant after the initialisation and after the execution of the loop body.

The third type of annotation is the candidate invariant *L : suggest P*. This simply says that it is possible for the assertion *P* to be true at point *L*, but that we have not formally verified it as yet. We would use this type of invariant in the course of the annotation process, and we would hope that all such invariants would have been verified or eliminated by the end of that process.

In practice, these rules are closely associated with the axiomatic semantics for the given language. We will describe (a small number of) the rules by giving a fragment of possibly annotated program text, and writing in any new assertions that we feel entitled to make in italics. All of the rules involve local invariants, so we shall not worry about prefixing them with a label.

There is one obvious condition that we can establish after an assignment:

x := a *{assert x = a}*

We can also move invariants forwards through an assignment:

{assert P(u,y)} x := u *{assert P(x,y)}* and, of course, backwards:

{assert P(u,y)} x := u {assert P(x,y)}

Rules for the conditional statement include the immediate observation:

{assert P} **if** B **then** *{assert $P \wedge B$}* S_1 **else** *{assert $P \wedge \neg B$}* S_2 **fi**

and rules such as:

if B **then** S_1 {assert P} **else** S_2 {assert Q} **fi** *{assert $P \vee Q$}*

An important rule for the while statement would be:

{assert P} **while** B **do** *{assert B}* S {assert P} **od** *{assert $P \wedge \neg B$}*

A number of rules such as these are presented in Appendix 4 of [Ders83], along with heuristics such as:

if B **then** S_1 {assert P} **else** S_2 {assert Q} **fi** *{suggest $P \wedge Q$}*

We will not be considering such rules and heuristics in any great detail. The purpose of this thesis is to provide a framework within which rules such as these have been implicitly incorporated; this framework will be the semantics (in Z) which we will construct for a sample programming language. We are not attempting for one moment to dismiss the usefulness of such guidelines; indeed, we argue that one effect of our approach will be to facilitate their use. We will be returning to this theme in chapter 6, when we attempt to analyze some sample programs, and consider the nature of the techniques which must augment our framework in order to do this.

CHAPTER 2 - SPECIFYING PROGRAMS

2.1 The Nature of a Specification

The purpose of a specification of any system is to describe, in some way, what that system is supposed to do. The exact nature of the specification will depend on the composition and operation of the system - we will build our specification so as to highlight those aspects of the system that we have deemed to be important. As with the process of describing the semantics of some language, we will build a specification by describing the system in terms of structures which are familiar to us. A specification language provides a standard notation for describing these structures.

One critical aspect of a specification is that we will usually not wish to describe how the system is to be implemented. This "abstract" view allows us to reason about the specification, to consider aspects such as its correctness and completeness, without being constrained by having to work in an environment defined by some programming language. Presumably (although not necessarily) we will want to implement the specification at some stage; this will involve a process of adding commitments to the specification which prejudice it in favour of some programming language or paradigm.

Since a specification only describes what a system will do, it is possible to have specifications which can never be implemented - the Halting Problem would constitute such a specification. It is also possible to have specifications where we do not know if they can be implemented (for example, to generate solutions to an unsolved existence theorem such as Fermat's Last Theorem). As we refine a specification we may make implementation choices which will not represent a full implementation, but which we will deem to be acceptable compromises: in most cases we could use a programming language representation for natural numbers in place of the infinite set of natural numbers mentioned in a specification.

Another important feature of a specification is non-determinism. This occurs when we specify the range for an acceptable result of some operation, rather than distinguishing a unique result for each case. This may be due to the fact that we are not interested in the actual result, or because we have deliberately built in a "don't care" situation to the specification. In all instances, if we are attempting to implement the specification in a deterministic language, we will be forced to make some kind of decision as to a single acceptable result. Thus the process of implementation can be seen as one of adding information to the specification, or of constraining it to a greater degree.

When we talk of a specification possessing a degree of abstraction, we are, to a large extent, making an essentially subjective decision. The abstractness of a specification depends on what you are comparing it against: usually we would have some informal notion of the problem domain and some idea of a target programming environment upon which to base such a decision, but, in the absence of these, the notion of abstraction is entirely arbitrary. When we have two or more specifications we can begin to make comparisons and possibly formulate some hierarchy of abstractness, with each specification being regarded as being at a particular level of abstraction when compared to the others. This is central to the idea of stepwise refinement, where an implementation will, in turn, become a specification for future implementation steps.

2.2 Specification Methods

As a prelude to our discussion of Z we will take a (very) brief look at the two principal specification paradigms. Examples of the property-oriented method (also known as algebraic, or equational, specification) include CLEAR ([BuGo77]), Larch ([GHW85]) and OBJ ([EhMa85]), while the main examples of the model-oriented method are Z ([Spiv87]) and VDM ([Jone90]).

2.2.1 Property-Oriented Methods

The property-oriented specification method seeks to describe a system by stipulating how the operations interact with each other. The method therefore concentrates on a description of the properties of the system (as described by these operations).

A property-oriented specification is composed of building blocks which we will call theories; these theories usually consists of two parts - a signature, and a set of axioms (or equations). The signature gives the names of the types being defined and the names of the functions along with the types of their domain and range. Individual elements may be distinguished as being of a certain type. The set of axioms describe the properties of the types, functions and elements by describing how they interact with each other. These equations typically would involve some combination of the applications of some of the functions being equated with an application of different functions, or possibly the same functions in a different order. Common properties such as inverses, transitivity etc. can be described in this manner.

A specification of a system can be constructed by combining theories; various strategies exist for the combination methods. A specification will start from some basic set of theories, and will then combine and extend these to describe the operation of the system. It is thus possible to build up a library (or "heritage") of theories which express basic properties that may be useful in describing a system. For example, the Larch Handbook [GuHo86b] starts with definitions of basic properties such as associativity, transitivity, partial and total orderings, and builds up to specifications of sets, queues, trees, rings, lattices etc., most of which will have parameterised types and operations.

2.2.2 Model-Oriented Methods

The model-oriented method regards a system as consisting of a group of states, each of which corresponds to some configuration of the components of the system. An operation in the system is regarded as something which moves that system from one state to another. The building blocks of a model-oriented specification are sets, and the associated ideas of tuples, functions, sequences etc.; these are used to build a model of the system by constructing a model of a state of that system. Operations are modelled by specifying the states before the operation, and the corresponding states after the operation has taken place.

2.3 The Specification Language Z

2.3.1 A Brief Introduction

The basic unit of specification in Z is the schema, which consists of a declaration part and a predicate part. The declaration part introduces some group of variables (giving their name and the set to which they belong), while the predicate part constrains their values by presenting a predicate in which they appear. A schema is written as:

$$\mathit{Name} \mathrel{\hat{=}} [\, x_1 : t_1; \ldots; x_n : t_n \mid P(x_1, \ldots, x_n) \,]$$

where $x_1,\ldots,x_n$ are variable names, $t_1,\ldots,t_n$ are their corresponding type (ie. the set, or tuple, function etc., to which they belong), and $P(x_1,\ldots,x_n)$ is some predicate ranging over the variables. We can, if needed, associate some unique *Name* to a schema, and future references to *Name* are taken to be references to the variables of the declaration part of that schema (suitably constrained by the predicate part, of course). Every variable, when first introduced, must be given a type which associates it with a set (or function, tuple...) of some previously defined type. We take sets such as $\mathbb{Z}$, $\mathbb{N}$, $\mathbb{R}$ etc. as our basic types.

The most fundamental type of schema is that which describes what it means to be a state of the system; this is called the state invariant schema, and it effectively specifies a set, each member of which will be a valid state. We will also designate a subset of this set of states as being the valid set of start-states for the system as a whole; these are usually called the initial states of the system.

Operations are also modelled using schemas, except that these will be different from the state-invariant schema in that they must specify two sets of states - the valid before- and after-states of the operation. To distinguish between the value of some variable x before and after an operation, it is conventional to write the latter as x' in a schema.

2.3.2 The Schema Calculus

Since Z schemas consist basically of a predicate, we can construct new schemas from old by using standard logical operators such as $\neg$, $\wedge$, $\vee$, $\Rightarrow$ and $\Leftrightarrow$. Thus for some schemas named *A* and *B* we can construct the schema $A \wedge B$ whose declaration part is the combination of the declaration parts of *A* and *B*, and whose predicate part is the conjunction of the predicate parts of *A* and *B*. Schemas for the other operators are defined similarly.

We can also use the quantifiers $\forall$ and $\exists$ with schemas; the phrase $\forall B$ for some schema *B* is formed by universally quantifying all the variables in *B*, and similarly for $\exists B$. The concept of "hiding" some of the variables of a schema simply means to existentially quantify those variables in the schema. Thus for the schema *Name* defined above, we could hide some variable, x_i say, in *Name* by writing:

$$[x_1 : t_1; \ldots x_{i-1} : t_{i-1}; x_{i+1} : t_{i+1}; \ldots x_n : t_n \mid \exists x_i : t_i \cdot P(x_1,\ldots x_n)]$$

This schema would usually be denoted $Name \setminus (x_i)$. We can hide a group of variables by projecting one schema onto another; *A* projected onto *B*, written $A \upharpoonright B$ consists of $A \wedge B$ from which all of those variables in *A* not shared by *B* have been hidden.

2.3.3 Schema Binding and θ

When we use the schema name *A* in some logical sentence, we are introducing all the variables of that schema, and any logical operation such as conjunction, quantification etc. involving the schema will be over all the variables of that schema. We can talk about an individual variable of A by writing $A.x_i$ where x_i has been declared in the predicate part of A; however, this will not always be exactly what we want.

Special variables in Z can be decorated; for instance variables representing the after-state of an operation are decorated by priming them (as in x'), while output and input variables would be distinguished by decorating them with an exclamation mark and question mark respectively (as in $x!$ and $x?$). It will be useful for us to be able to talk about these variables in certain circumstances without wanting to drag in all the other variables of the schema; we can use the binding operator for this purpose.

Suppose we have some schema S containing the variables $x_1,\ldots,x_n$ and $x_1',\ldots,x_n'$, and suppose we want to be able to assert some extra condition about the primed variables of S. What we do is to use some other schema, call it B, which contains the variables $x_1,\ldots,x_n$. Assuming that the variables $x_1',\ldots,x_n'$ have been brought into scope in the sentence (by mentioning the schema S, presumably), then we also can use the schema $\theta B'$ which will refer, whenever used, to all the primed variables of S; that is the θ operator has bound all the variables in B' with the corresponding variables which are currently in scope.

In a similar manner, if the variables in some schema O had the same names and types as the output variables of S, then we could use the schema $\theta O!$ whenever we wanted to assert some attribute of these variables. Note that schema binding only works when the variables of the decorated schema (ie. B' or $O!$) have previously been introduced by some other schema.

2.3.4 Scoping Rules

Since we will be hoping later on to model a programming language using Z, it will be worthwhile to take some time to investigate the scoping rules of Z, and how they compare with standard programming language.

For our purposes we will regard a variable in a programming language as being associated with some block. Each variable has to be declared, along with its type, and may be used from then on until the end of the block. If another block is declared from within that block, then we will expect to be able to use all the variables which were in scope for the original block. Thus we envisage a situation where variable declarations correspond to blocks which are arranged in a nested structure, being added and removed from scope in a stack-like (last in, first out) manner.

Variable names could be used for two different variables. In this case, a reference to a particular variable name is deemed to be a reference to the variable which was most recently declared; that is, the declaration of a variable will hide any variable with the same name which is currently in scope. When we move back out of the block in which the new variable was declared, the old variable will be "restored", and all references will again be taken as being to that variable. We could therefore envisage something like a tree-structure with blocks as nodes, with a sub-block definition inside a block being regarded as a child of that block. To find the declaration corresponding to a variable reference, we start in the current block, and then move back through the block's ancestors, stopping at the first declaration we find.

Z, on the other hand, is flat. The closest replica we have of blocks in Z are schemas which, in a specification, are presented sequentially. Once a variable has been declared, it stays in scope, and is available to all the schemas which follow. There is some correspondence, however, since the re-introduction of a variable name will "mask out" any previous references; thus variable names refer to the most recently introduced declaration of that variable. Basically, once introduced, a variable stays in scope until its name is re-introduced by the declaration of (what is effectively) some other variable.

It would be surprising if Z schemas did not operate in this manner, since they are basically predicates, and the scoping and binding rules are those of any variable in a sentence of predicate logic. What it means for us is that we are going to have to be fairly careful when dealing with scoping rules in a language definition. Issues involving the introduction of a variable, and having two variables with the same name should not cause us much trouble, since we will endeavour to ensure that we exploit these features in Z. However, we will expend most of our energy when dealing with scoping in ensuring that we can get the variables back out of scope when appropriate. We will deal with this matter further in chapter 4.

2.4 Models and Programs

Before going on to our schema calculus, let us pause for a moment to consider a possible relationship between models of systems and programs. We have identified three basic components of a model: the state invariant, the set of initial states, and a set of operations (over states satisfying the state invariant). In this light, let us examine a simple program consisting of just a *while* loop, as we would define it using the axiomatic method of § 1.4. We would formulate some predicate, call it I, which would be an invariant for the loop. This would mean that I would be established before the first iteration of the loop, and the execution of each subsequent iteration would also establish *I*.

Both specifications and programs essentially work with variables, so we can use this as a starting point for relating them. A state in a specification is effectively defined by a predicate which constrains the values those variables may take; we now assume the same description for a program state. Thus, regarding commands as predicate transformers means that we are effectively regarding them as state-to-state mappings; in a specification we would describe such mappings as operations. A state invariant is some schema whose properties hold for the before and after states of all the operations; in programming terms, a predicate is invariant for a command if whenever it is true before execution of the command, it will be true afterwards. Also, given the right "conditions" (ie. the existence of intermediate states), the composition of two operations may itself be an operation; the same is true for program statements.

Going back to our simple *while* loop - we identify two groups of statements which are of particular interest: the initialisation statements (basically, all those before the actual *while* command), and the loop body. We have asserted that the invariant *I* must be true before the loop is executed, so the initialisation commands must establish it. If use specification parlance to describe the loop invariant as the "state invariant", then we could regard the set of states which are established by the initialisation commands as our set of initial states. Similarly, since *I* will be true before and after an iteration of the loop, we can regard the loop body as an "operation" over the state-space defined by *I*. In order to ensure loop termination, some subset of the after-states will not satisfy the loop guard; this subset, or rather the predicate which defines them, is the post-condition of the program.

For larger programs, which possibly include a number of such loops, we could envisage a hierarchy of invariants. At the highest (or "outermost") level the invariant will simply specify the values that we would expect the variables to have during the entirety of the program (this corresponds to the idea of a global invariant in § 1.5.2). This will basically be a definition of the role of those variables in the program, and it could be anything from just the declaration of the variable (ie. confining it to membership of some set), or some assertion telling us that we need only consider values of the variables within a certain range. (The facility provided by some languages for declaring an identifier to be a constant with a specific value would be a particularly strong invariant). If we wished to formulate some (loop) invariant later on in the program, then we would use any invariant which had been defined at a higher level as a starting point, and strengthen it to get the new invariant[1].

The purpose of this discussion is to tie together from the outset the notion of a program and a specification as both being "models" of some process. We will specify in chapter 4 a process which will transform a program-model into a Z-model, and which will cater for the preservation of invariants as we have discussed. Before we attempt this however, we have a little work to do on that schema calculus...

2.5 A Predicative Basis for the Schema Calculus

The Z schema language allows us to deal with specifications as though they were propositions; however using a propositional-like calculus become quite confusing and it can often be difficult to discern the relationship between individual schemas in a sentence. We could of course use the schema binding operator which can provide some

[1]Note that there are two types of invariants here: the global invariant is true at the beginning and end of the program, and at all points in between, whereas we require of a loop invariant only that it is true of the before- and after-states.

quite elegant methods of expression. We have found it useful to introduce a predicate calculus to deal with schemas; we believe that this notation can often provide a more familiar basis for studying a statement involving a Z schema, especially to someone who is not that familiar with the nature of a Z schema, and who wishes to regard schemas as just a pre-condition/post-condition pair[2].

We will only consider two types of schema variable - those representing "before" values (unprimed) and those representing "after" values (primed). Since the semantics to be introduced in chapter 4 does not make much use of input and output variables, we feel justified in not considering them for the moment; besides, we can always adopt the strategy of regarding input/output operations as reference and assignment to designated I/O variables.

What we want to be able to do is to distinguish the primed and unprimed variables in a schema so that we can illuminate the correspondence between these variables in some assertion. Effectively what we are looking for is a description of a state; we defined this as being just a schema consisting of variable declarations; for our purposes here we will have to constrain this a little further.

2.5.1 Meaning of S(x,y)

Suppose x and y' are sets of variable declarations (ie. schemas consisting solely of declarations), and let S be some schema: then we will write $S(x,y')$ to denote the schema S which has had its unprimed variables replaced by the variables in x and its

[2]This is somewhat similar to the way specifications can be treated in VDM: see for example [Jone90] § 3.2

primed variables replaced by those in y'. We therefore make the assumption that the signatures of x and y' are compatible with the signature of S.

Thus if S consists of the variables $\{s_1 \ldots s_k, s_{k+1}', \ldots, s_n'\}$, and x and y' have variables $\{u_1, \ldots u_k\}$ and $\{v_1', \ldots v_{n-k}'\}$ respectively then $S(x,y')$ would be the same as

$$S\ [u_1 / s_1, \ldots, u_k / s_k, v_1' / s_{k+1}', \ldots v_{n-k}' / s_n']$$

Although each of S, x and y' are schemas (the latter two having no predicate part), we will maintain a convention of using lower-case letters to refer to schemas representing groups of replacement variables, and use upper-case letters for the names of the schemas in which this replacement occurs. When we refer to some schema z^+, where + is any decoration (or none), we will be indicating that all the variables in the schema have been decorated with +, as for y' above. In statements where the primed variables in a schema are bound by a quantifier outside the schema, we will not feel obliged to ensure that our replacement variables are primed; when it adds to the clarity of the sentence we will do so. Note that if $S(x,y)$ appears in a sentence, then the names of the variables contained in x and y are always assumed to be unique.

On occasion we will take the liberty of referring to schemas such as x and y as "states".

2.5.2 A replacement mechanism

However, one problem immediately arises - which variables should be replaced for which? The variables which occur in the declaration part of a schema are not in any particular order, yet we will want to ensure that the replacement strategy we use is unique; ie. that *$S(x,y')$* will mean the same thing if it is mentioned twice in the same assertion (for the same S, x and y' of course). Thus we need to indicate which variables are to be replaced by which. We get around this problem by demanding that each variable in x and y' contain some unique subscript as part of its name which will indicate the variable that it will replace.

Let us investigate the feasibility of such a replacement. Suppose the declaration part of S looks like:

$$[s_1 : t_1 ; \dots ; s_k : t_k ; s_{k+1}' : t_{k+1} ; \dots ; s_n' : t_n]$$

Now each declaration in S occurs only once; therefore the declaration of a variable is unique for each variable. For $1 \leq j \leq n$, let *dj* represent the result of applying some mapping function J to the characters in the declaration $s_j : t_j$. We require of J that each *dj* is distinct, and that we can distinguish the corresponding declaration uniquely from it. (That is both J and J^{-1} are injective - such a mapping is central to the idea of Gödel numbers; see eg. [Herm69] § 1.3) We will thus expect the variables in x and y' to be indexed by the set of *dj*s; we will want x to be something like $[u_{d1} : t_1 ; \dots u_{dk} : t_k]$ and y' to be $[v_{d(k+1)}' : t_{k+1} ; \dots v_{dn}' : t_n]$. These conditions will ensure that our replacement mechanism will maintain the meaning of assertions involving schemas.

Note that an important consequence of this strategy is that we do not need to worry about issues such as the order or decoration of variables. Thus we can write something like *$S(x,y)$* without priming the y in situations where it is useful; we will of course attempt to keep the convention of priming the after-state variables when possible. Also we can technically write *$S(y',x)$* instead of *$S(x,y')$* with the same meaning; strictly speaking this depends on the names given to the individual variables, so such a change is unlikely to lead to greater clarity; for the moment we will just note that it is possible.

We do not need to investigate the issue of replacement strategies further; it is sufficient that we know that the replacement paradigm we desire is possible. From now on when we speak of some *S(x,y)* we assume that a replacement strategy equivalent to the one described above has been adopted.

2.5.3 Schema pre- and post-conditions as predicates

We can immediately start to use our this notation. If a schema *S* is represented as a two-place predicate, then *pre-S* will be represented as a one-place predicate, defined as:

$$\forall\ x \ \cdot\ \text{pre-S}(x) \Leftrightarrow \exists\ y' \cdot S(x,y')$$

We assume that the set of variables *x* have already been brought into scope.

In a similar manner, we could also define:

$$\forall\ y \ \cdot\ \text{post-S}(y) \Leftrightarrow \exists\ z \cdot S(z,y)$$

2.5.4 Schema composition as a predicate

Next take the schema composition *S;T*, which we can define (as per [Spiv89]), for some appropriate *State* as

$\exists$ State$''$ $\cdot$

$$(\exists\ \text{State}' \cdot [S \mid \theta\text{State}' = \theta\text{State}'']) \wedge (\exists\ \text{State} \cdot [T \mid \theta\text{State} = \theta\text{State}''])$$

Using our notation, this definition becomes:

$$\forall\, x,z' \cdot S;T(x,z') \Leftrightarrow (\exists\, y \cdot S(x,y) \wedge T(y,z'))$$

Because we can use *y* with both schemas we do not need to introduce two more states representing the "before" state of *T* and the "after" state of *S* and assert the appropriate equality. Also, we highlight the fact that *S;T* is a schema by writing it as a two-place predicate in *x* and *z*′.

In [Spiv89], § 5.4 a rule is given which will ensure that the specification *S;T* is correctly implemented by the program *S;T* (that is, we are replacing schema composition, represented by ";", with program statement composition, represented by ";"). The rule insists that every valid after-state of *S* will be a valid before-state of *T*, or, for an appropriate *State*, in standard Z notation we can write:

$$\forall\, State'' \cdot (\exists\, S \cdot \theta State' = \theta State'') \Rightarrow (\exists\, T \cdot \theta State = \theta State'')$$

We can now write this condition as:

$$\forall\, y \cdot (\exists\, x \cdot S(x,y)) \Rightarrow (\exists\, z \cdot T(y,z))$$

or, using the predicates defined above, as:

$$\forall\, y \cdot \text{post-}S(y) \Rightarrow \text{pre-}T(y)$$

2.5.5 Other schema operations

Schema composition insists that the variables in both schemas have the same name and type. However, we can use $\wedge$, $\vee$ or $\Rightarrow$ between schemas without any such restriction. For schemas $S(x,x')$ and $T(y,y')$ we will form their conjunction by merging the declaration parts and conjoining the predicate parts (and similarly for disjunction and implication). Thus we can write

$$\forall\ x, x', y, y' \cdot S(x,x') \wedge T(x,x') \equiv S{\wedge}T(x;y,x';y')$$
$$\forall\ x, x', y, y' \cdot S(x,x') \vee T(x,x') \equiv S{\vee}T(x;y,x';y')$$
$$\forall\ x, x', y, y' \cdot S(x,x') \Rightarrow T(x,x') \equiv S{\Rightarrow}T(x;y,x';y')$$

We may also wish to quantify some of the schema variables. Suppose the variables that we wish to quantify are listed in q and q' - we can simply write $\forall\ q,q' \cdot S(x,x')$ as you would expect. If we wish to indicate that all the variables in q and q' are variables from the schema $S(x,x')$, we can write $q \Rightarrow x$ and $q' \Rightarrow x'$. (This is using the fact that our "lists" of variables are really schemas containing the declarations of these variables.) To emphasise this point we could describe the remaining variables using some schemas r and r' (such that $(q \wedge r) \Leftrightarrow x$ and $(q' \wedge r') \Leftrightarrow x'$), and we can then write $\forall\ q, q' \cdot S(q;r,q';r')$.

2.5.6 Predicates defining a state

Up to now we have dealt with schemas which specified an operation - thus they were defined over two states. However, we can also have schemas which operate over just one state: the state invariant for any specification is an example. As you might expect, if P was such a schema, we would represent it as $P(x)$.

For a particular specification, we can represent the state invariant by some schema $I(x)$ (or by convention in Z, we might write $I(x')$). Then, if $O(x,x')$ is any operation in the specification, we would expect its valid before- and after-states to be a subset of those states defined by the invariant - we would thus have $\forall\, x, x' \cdot O(x,x') \Rightarrow (I(x) \wedge I(x'))$.

One-state schemas may also be used for constraining the valid before- or after-states of some schema by conjoining them with that schema. Thus we could restrict the before-states of some specification $S(x,x')$ by writing $P(x) \wedge S(x,x')$, which we will usually write $P{\wedge}S(x,x')$. In a similar manner we can write $S;T(x,x') \wedge P(x')$ as $(S;(T{\wedge}P))(x,x')$ and $S;T(x,x') \wedge P(x)$ as $((S{\wedge}P);T)(x,x')$.

CHAPTER 3 - REFINING SPECIFICATIONS

We have thus far considered formal descriptions of programs and specifications; we now turn to describing the link between them. The job of a specification is to formally define the set of valid states of the system being modelled, and to describe the effects of the operations over these states. A program will have a greater burden: it must realize (or implement) valid states in terms of the constructs available to it, and it must describe a method of establishing the required results of each operation.

In this chapter we provide a general description of the process of developing a program from a specification, known as the *refinement* process, and look at some of the principal ideas. We give the Z proof rules in terms of the calculus introduced in the preceding chapter, and we demonstrate the application of these rules to specific types of program statements.

3.1 The Idea of a Refinement Calculus

Rather than regard programs and specifications as being two entirely different entities which are to be eventually reconciled through some sort of formal link, there is a growing trend (as in [HoHe87], [MoVi90], [Morr90a] etc.) towards using the same framework which can handle both. The view taken is that a program is simply a specification which can be expressed using only certain operations and certain data types; refinement then becomes a matter of moving between specifications, until we eventually reach one which is "implementable".

Since we have introduced the *wp*-calculus, we will follow [Morr90a] and use this as our method of defining programs and specifications. We discussed earlier the meaning of $P = wp(S,Q)$ in the situation where S is a statement; we now extend this to cover the situation where S can also be a specification. In chapter 5 we will provide a formal meaning for *wp*, but for now we will just regard the statement as saying that S will map any before-state satisfying P onto an after-state which satisfies Q. This definition of *wp* will allow us to use specifications and programs interchangeably.

The refinement process will involve starting with some specification S_1, and constructing a list of specifications $S_1, \ldots, S_n$ such that S_n is implementable, and each S_i is a refinement of the preceding S_{i-1}. Each refinement step can involve operation or data refinement, or both. We will be discussing the proof-obligations necessary for showing that one specification S_i refines some other specification S_{i-1}; if these obligations are discharged correctly, then we write $S_{i-1} \sqsubseteq S_i$, where $\sqsubseteq$ means "is refined by". From our discussion of refinement, it will be evident that $\sqsubseteq$ must be reflexive and transitive, and thus defines a partial ordering over specifications (and programs). Transitivity is important, since we will want to be able to assert that for our list of specifications $S_1 \sqsubseteq \ldots \sqsubseteq S_n$, we thus have $S_1 \sqsubseteq S_n$, which corresponds to the assertion that the program S_n correctly refines the specification S_1.

3.2 Refinement as Interpretations between Theories

If we wish to build up an axiomatic system in some logical framework, we must start off with a set of symbols (an alphabet) and a group of axioms. The symbols will be used for variable, constant, predicate and function names; the axioms will be formed from combinations of these symbols. An *interpretation* gives a semantics to these symbols (and therefore to the axioms) by mapping them into constants, variables, predicates and functions from some chosen set. The result is termed a *theory*. We can form the *extension* of a theory by adding in new symbols and axioms. We can also form a theory by combining other theories; the new theory consists of the union of the languages and the axiom sets.

We can regard the basic given sets (such as $\mathbb{N}$, $\mathbf{Z}$ or Bool) of a specification language as theories: writing a specification then involves constructing new operations over these theories. Defining an operation entails introducing a new function name and a number of axioms to specify its properties. Adding a number of such operation definitions to a set of theories is equivalent to constructing an extension to that set. For some specification Q, we can say that the theory which Q represents, denoted TH[Q], is an extension of the combination of some base-set of theories; we will denote this base set as BS[Q]. Thus one measure of the "abstractness" of a specification might be the degree to which this base-set can be implemented; as we refine the specification, we would hope to replace each member of the base-set with an implementable equivalent.

Suppose Q is refined to some specification R. First of all we will have to refine the theories involved in the base-set of Q to some "more concrete" equivalent. If the base set of theories for specification R is BS[R], then we will want to extend this so as to represent the theories in BS[Q]; let us call this extension the base-image in R, or BI[R]. We must also declare which elements of BS[Q] correspond to BI[R] - this is done by means of an interpretation, call it $\mathbb{I}_B$, between the two theories.

Once we have modelled the base set, we can construct an equivalent to TH[Q] by extending BI[R] by the concrete operations; the result of this extension will be TH[R].

We will also have to specify which concrete operations refine which abstract operations; again this is done using an interpretation, call it $\mathbb{I}_T$ between the languages of the theories of Q and R.

We can represent the above refinement from Q to R diagrammatically, using horizontal arrows to represent interpretations, and vertical arrows to represent extensions, as:

$$\begin{tikzcd}
TH[Q] \arrow[r, dash, "\mathbb{I}_T"] \arrow[d, dash] & TH[R] \arrow[d, dash] \\
BS[Q] \arrow[r, dash, "\mathbb{I}_B"] & BI[R] \arrow[d, dash] \\
 & BS[R]
\end{tikzcd}$$

The basic building block of the whole process is the interpretation between one theory and the extension of another; this is what [TuMa87] describe as the "canonical step".

Once this interpretation has been defined, we can then set about verifying our refinement by showing that for each axiom α in TH[Q], the corresponding version, $\mathbb{I}_T<\alpha>$, holds true in TH[R]. This gives us a basis for demonstrating that there exists a *morphism*, or property-preserving transformation, between the theories; that is for each abstract operation *A*, and for every state *a* in its domain (satisfying its pre-conditions), we will want to assert that $\mathbb{I}_T<A(a)> = \mathbb{I}_T<A>(\mathbb{I}_T<a>)$. If we look at the interpretation from the concrete to abstract theories, then the morphism which best represents an abstraction would be one which is surjective; this many-to-one map is called an *epimorphism*. A special case occurs where the mapping is a bijection - this is known as an *isomorphism*. Two theories which are isomorphic are essentially the same from the perspective of some higher level of abstraction.

Since refinement will be proceeding step-by-step, if R is not concrete enough we would have to refine it in a similar manner, starting with BS[R], and so on, until eventually we reach some theory S whose base set is implementable directly in the target programming language.

3.3 The Refinement Relation and Specification Bias

We will usually want our first model of any system to be as abstract as possible, with the proviso that it should contain a full and useful description of all its important features. In some situations however, the model may contain more information than is apparently necessary - we say that the specification is *biased* if this is the case. When we specify the state invariant, we are identifying a set which will contain the before and after states of all the subsequently described operations. Each operation will induce some equivalence classes over these states, where one state is deemed to be equivalent to another with respect to a given operation if they cannot be distinguished from each other by that operation. An operation cannot distinguish between two states if (i) neither are valid before-states, or both are mapped to the same (set of) after states and (ii) for each valid before state neither are valid corresponding after-states, or they both are.

We need not worry if two states cannot be distinguished by one particular operation; usually they will have been introduced because they are needed by some other operation in the specification. However, when two (or more) states cannot be distinguished by *any* operation then we regard our specification as being biased. Our justification for distinguishing between such states could be that their existence will make it easier to work towards some particular implementation that we have in mind; thus these extra states form a sort of implicit specification which has placed extra constraints on our model. We could also use these states to bound the range of acceptable after-states for some non-deterministic operation; realisation of this operation (which includes a move towards greater determinism) may pick just one of these as being acceptable.[1]

What effect will specification bias have on the refinement process? When attempting to refine a biased specification we are faced with the situation where some abstract states may not have any corresponding concrete representation, or several different abstract states may not be distinguished in the concrete model (they are mapped to the same concrete state). If the specification is not biased then we are faced with a simpler situation: no longer do we have a many-to-many relation, but instead a much more manageable one-to-many relation whose domain is the whole of the abstract state-space. The inverse of this relation is thus a function from concrete states into abstract states which is surjective; this will correspond to the notion of an epimorphism as mentioned above. (This surjectivity is the *adequacy* condition of [Jone90] and corresponds to the notion of *functional refinement* described in [Spiv89] § 5.7) Our proof rules will be simpler in this case since we will not have to introduce the condition of the existence of an abstract state for each corresponding concrete state each time. In what follows we will deal with rules which cover the more general case; it should be apparent which constraints can be deleted when dealing with an unbiased specification.

[1]Some examples of biased specifications can be found in [Jone90] § 9.3

3.4 Refinement Proof Obligations in Z

We have already introduced the idea of a morphism as being a suitable representation of the refinement process, since it involves preserving the properties which held for the abstract operations even after they have been translated into their concrete counterparts. We now need to address the problem of proving that our mapping is indeed a morphism - we do this by ensuring that a set of proof obligations are discharged by the concrete representation.

We consider a single step between an abstract and concrete specification (in future steps this concrete specification will become itself the "abstract" specification as we set about refining it). Each specification will consist of schemas specifying the state-invariant, a set of initial states and a number of operations. Let us name the state-invariant and initial-state schemas respectively as *Abs* and *AbsInit* for the abstract specification and *Con* and *ConInit* for the concrete specification. Thus, for any abstract state *a* and concrete state *c*, we can assert that $AbsInit(a) \Rightarrow Abs(a)$ and $ConInit(c) \Rightarrow Con(c)$.

We will also presume that we have constructed a schema which describes the connection between the concrete and abstract states; let this relation be described by the schema *Rel*, which will have *Abs* and *Con* as its declaration part. We can thus assert that $\forall\ a,c \cdot Rel(a,c) \Rightarrow Abs(a) \wedge Con(c)$. In certain situations we will want to regard the relation as being directional (from abstract to concrete states, or vice-versa). If we have some abstract state a' and concrete state c' then $Rel(a,c')$ is a mapping from abstract into concrete states, and similarly $Rel(a',c)$ is a mapping from concrete into abstract states (this is the retrieval relation).

3.4.1 Refining Initial States

Our first proof obligation is to ensure that each concrete initial state has a corresponding abstract state, and that this abstract state is an initial state in the abstract specification. This can be expressed straightforwardly as:

$$\forall\, c \cdot \mathrm{ConInit}(c) \;\Rightarrow\; \exists\, a \cdot \mathrm{AbsInit}(a) \wedge \mathrm{Rel}(a,c)$$

3.4.2 Proof of Applicability

Next we must prove that each concrete operation correctly models the corresponding abstract operation. Suppose that we are checking a concrete operation, called *ConOp* against an abstract operation *AbsOp*. We would certainly hope that the concrete operation is at least as likely to terminate as the abstract operation - this is known as *applicability*. What we are asserting is that whenever the abstract operation is guaranteed to terminate for a state, then the concrete operation will also terminate for the corresponding concrete states. An operation terminates for a given state if that state satisfies the pre-conditions of the operation; thus the applicability condition is:

$$\forall\, a,c \cdot \text{pre-AbsOp}(a) \wedge \mathrm{Rel}(a,c) \Rightarrow \text{pre-ConOp}(c)$$

This is also known as the *domain* condition, since we are basically asserting that the domain of the abstract operation is mapped into a subset of the domain of the concrete operation.

3.4.3 Proof of Correctness

Our main proof obligation (known as the *correctness* condition) is obviously to ensure that the concrete operation produces the correct results for all the abstract states to which it is applicable. For each of these states the concrete operation must produce an after-state that can be mapped back to an after-state of the abstract operation when applied to the corresponding abstract before-state.

$$\forall\ a, c, c' \cdot$$
$$\text{pre-AbsOp}(a) \wedge \text{Rel}(a,c) \wedge \text{ConOp}(c,c') \Rightarrow \exists\ a' \cdot \text{Rel}(a',c') \wedge \text{AbsOp}(a,a')$$

This is a formal statement of the homomorphism property: that is, the result of mapping any valid before-state of *AbsOp* into a corresponding concrete state, carrying out *ConOp* on this state, and mapping the resulting concrete state back into its abstract counterpart, is the same as the result we can get by applying *AbsOp* to *a*.

3.4.4 Functional and Operational Refinement

As we mentioned earlier, functional refinement is a special case of this, and we would be allowed to drop the existential quantifier from the proof-obligation for initial states, and from the right-hand-side of the correctness condition. A special case of functional refinement involves the situation where the state-space is the same in the abstract and concrete models, and the refinement step only involves the operations. This case, known as *operational refinement*, has similar proof rules to the applicability and correctness conditions given above, except that now we can also drop all the references to *Rel*. The two conditions will now be:

Applicability: $\forall\ s \cdot \text{pre-AbsOp}(s) \Rightarrow \text{pre-ConOp}(s)$

Correctness: $\forall\ s, s' \cdot \text{pre-AbsOp}(s) \wedge \text{ConOp}(s,s') \Rightarrow \text{AbsOp}(s,s')$

where *s* is a state (we do not need to distinguish between abstract and concrete states since they are now both described by the same state invariant). These can then be combined[2] into one condition:

$$\forall\ s,s' \cdot \text{pre-AbsOp}(s) \ \Rightarrow\ (\text{pre-ConOp}(s) \wedge (\text{ConOp}(s,s') \Rightarrow \text{AbsOp}(s,s')))$$

3.5 The Refinement Rules in Practice

We have seen how specifications may be refined step-by-step, so that each time we move closer to a more "implementable" version. The last step of this process will involve translating suitably refined specifications into individual program statements; we will take look at the form of the proof obligations for refinement from Z specifications into sequential, alternative and iterative statements.

For these three situations we are assuming that we have reached a stage where the state-space has been expressed in terms used by the programming language, so that we will only consider operational refinement. Each of these refinements will involve deterministic statements only, so a choice may be involved when actually using them if the specification being refined was non-deterministic.

[2]Using the rules that for any propositions A, B, C and D, we have

$$\begin{array}{l} A \Rightarrow B \\ A \wedge C \Rightarrow D \end{array} \ \vdash \ \begin{array}{l} A \Rightarrow B \\ A \Rightarrow (C \Rightarrow D) \end{array} \ \vdash \ A \Rightarrow (B \wedge (C \Rightarrow D))$$

3.5.1 Sequential Composition

Suppose we have some schema *A* which we would like to refine to *S;T*, where *S* and *T* can be refined directly to program statements. The proof obligations demand that we must show:

$$\forall\ x \cdot \text{pre-A(x)} \ \Rightarrow\ \text{pre-(S;T)(x)} \wedge (\forall\ x' \cdot \text{S;T}(x,x') \Rightarrow \text{A}(x,x'))$$

But since *S* and *T* may be represented by program statements, we can assume that they are deterministic, and so we can simplify this to:

$$\forall\ x \cdot \text{pre-A(x)} \ \Rightarrow\ \exists\ z \cdot \text{S;T}(x,z) \wedge \text{A}(x,z)$$

3.5.2 The Alternative Statement

We assume that we want to refine the specification *A* to a (deterministic) alternative statement of the form:

if B_1 **then** S_1 **elseif** B_2 **then** S_2 **elseif ... elseif** B_n **then** S_n **else** S_0 **fi**

Let us also assume that each of the B_1 ... B_n are disjoint (ie. $\forall\ i,j : N \mid 1 \leq i < j \leq n \cdot \neg(B_i \wedge B_j)$) - this is not much of a restriction since the guards are checked in strict sequence, and we could always conjoin the negation of the disjunction of all the previous guards to each of them.

We regard each B_i as a one-state schema which constrains the before-states of S_i; thus we can form the schema $(B_i(x) \Rightarrow S_i(x,x'))$ which we will denote $BS_i(x,x')$. Let B_0 be defined as $\neg(B_1 \vee \ldots \vee B_n)$, so that we can also form BS_0.

The *if .. fi* statement will terminate if one of its branches terminates; thus its pre-condition is the disjunction of the pre-conditions for each of the branches

$$\forall\, x \cdot \text{pre-A}(x) \;\Rightarrow\; ((\text{pre-BS}_0(x) \vee \ldots \vee \text{pre-BS}_n(x)) \wedge$$
$$(\forall\, x' \cdot \text{BS}_0(x,x') \Rightarrow \text{A}(x,x') \vee \ldots \vee \text{BS}_n(x,x') \Rightarrow \text{A}(x,x')))$$

We can tidy this up a little by letting $IF(x,x') \mathrel{\hat{=}} BS_0(x,x') \vee \ldots \vee BS_n(x,x')$, and using some of the rules[3] from chapter 2 to get the condition

$$\forall\, x \cdot \text{pre-A}(x) \;\Rightarrow\; \text{pre-IF}(x) \wedge \forall\, x' \cdot \text{IF}(x,x') \Rightarrow \text{A}(x,x')$$

3.5.3 The Iterative Statement

We will want to refine the specification *A* to a statement of the form

while B **do** S **od**

Analogous to what we did for the alternative statement above, we can define a schema *WH(x,x')* to describe an iteration of the loop body:

$$\forall\, x, x' \cdot \text{WH}(x,x') \;\Leftrightarrow\; ((\text{B}(x) \wedge \text{S}(x,x')) \vee (\neg\text{B}(x) \wedge \text{Id}(x,x')))$$

[3]And also the fact that for any *A*, *B* and *C*, we have $(A \Rightarrow C) \wedge (B \Rightarrow C) \dashv\vdash (A \wedge B) \Rightarrow C$

As we discussed earlier, the standard approach here is to formulate some invariant, call it *I(x)* which effectively describes the operation of the loop for us. We will want this invariant to be true before and after the execution of the loop, and we can define a schema which asserts this property:

$$\forall\ x, x' \cdot \mathrm{WHI}(x,x') \Leftrightarrow (\mathrm{I}(x) \wedge \mathrm{WH}(x,x') \wedge \mathrm{I}(x'))$$

We are now ready to define the proof obligation for the iterative statement, remembering that we will only want the equivalence with *A* to be established upon termination of the loop (when we have $\neg B(x')$):

$$\forall\ x \cdot \text{pre-A}(x) \Rightarrow \text{pre-WHI}(x) \wedge (\forall\ x' \cdot \mathrm{WHI}(x,x') \wedge \neg \mathrm{B}(x') \Rightarrow \mathrm{A}(x,x'))$$

3.6 Motivation for a semantics in Z

Up to now in this chapter we have concentrated on describing various features of the link between program and specifications, and attempting to unify some of the notation using Z. However, it is fundamental to any attempt to refine a specification into a programming langauge that we have a formal description of what a program in that language actually means; we have discussed some such formal definitions in chapter 1. The basic assumption is therefore that a specification is regarded as being already expressed in "formal" terms, and we must make the link between these terms and suitable equivalents in the programming language.

The starting point in the move from an existing program back towards a formal description of its operation or properties will obviously be the sequence of statements in the programming language which constitute the actual program. We will presumably utilise some form of formal semantics in order to provide a basis for our work, and we will then begin to work back towards a specification. The main concern of this paper is to provide such a basis using the Z notation.

The idea of applying formal specification techniques to the field of formal semantics is not new; this was the original motivation behind the VDL language, an early ancestor of VDM. Much of the early work on VDM has concentrated on describing the denotational semantics of programming languages, including comprehensive definitions of full-scale languages such as Pascal and Algol (as in [BjJo82]). The basis of any denotational description is a set of functions over a domain, and a specification language such as VDM can be used to formalise the definitions of these functions. The mapping functions from the program statements into the semantic domain can be described in the same manner; VDM also allows us to address separately the issues of the semantic well-formedness and the meaning of phrases in the language.

We do not seek here to just echo these techniques in Z. The result of our semantic mapping will not be a set of well-defined functions (as such), but an actual Z specification which will describe the program. This specification will involve the sequential composition of a number of smaller schemas, each of which will represent a statement from the program. We will thus be regarding a program statement as specifying an operation over those variables of the program which are currently in scope. Although much of this work is denotational in style, we will not be relying on (or establishing) the existence of the properties of certain types of domains (such as continuity and monotonicity) which are central to denotational semantics in the style of

Scott. Our semantics are intended to be reconcilable with specifications, which are expressed in terms of predicates, and so whenever we look for a label, we shall claim that the semantics presented in the following chapter belong to the axiomatic category.

Our task then will be to take a program and to construct a specification which represents in Z. To do this we will need to define a mapping from individual statements of the language into Z schemas; we will do this using the Z notation. Chapter 4 will thus refer to two types of specifications. First of all its main task will be to specify the mapping from programs into specifications, which will be done explicitly by the presentation of a number of functions for this purpose. Secondly, this mapping will involve describing and constructing elements of the target specification; we will therefore also be implicitly illustrating of the nature of such a specification.

The purpose of this is to provide structures by which a program can be converted directly into a Z specification, and this specification can then be used as a basis for further study of the program. Since our specification will be relying on the composition of schemas as its basic element, there will obviously be much scope for the simplification of such a specification. The rules for such simplification depends on the properties of the data types (such as integers, natural numbers), are largely heuristic in nature, and will be one of the topics for discussion in chapter 6. However, the essential feature of the next chapter is that it constructs a framework within which this work can take place, based around the Z notation.

CHAPTER 4 - SEMANTICS OF A SMALL LANGUAGE

4.1 A Simple Programming Language

Describing the formal semantics of any language involves constructing a mapping from elements of the syntactic domain into particular elements in the semantic domain. The syntactic domain consists of those phrases which describe the syntax of the programming language, from which the non-essential details (such as statement delimiters) have been abstracted away; this is the ***abstract syntax*** of the language. The abstract syntax of the language which we will be using is given below.

4.1.1 Informal description of the language

It is a simple block-structured language; the main program block is distinguished by the keyword *Program*. Each block may contain procedure definitions as sub-blocks; such a definition may optionally include formal parameters. There are two types of parameters - ***value*** and ***variable*** parameters. Value parameters take an expression as argument; their initial value is the value of that expression at the time of procedure invocation. The value of any variable in the expression is not changed by passing it as an argument in this way. Variable parameters take an identifier (ie. variable name) as argument; references to a variable parameter in the procedure body are then treated exactly as if they were references to the actual argument. As a result of this, the value of a variable passed as argument to a variable parameter may be changed.

Basic data types are the Natural numbers, Integers and Booleans. Composite data types may be (multi-dimensional) arrays or records. An identifier may be declared directly as an array; for an identifier to be of type record, the record must already have been declared and named.

The basic statements are the (concurrent) assignment and the skip (or null) statement. Communication to an external environment is via the read and write commands. The call command invokes the named procedure with a list of arguments. Composite statements are the conditional (if-then-else-fi) and iterative (while-do-od).

We distinguish terminal symbols by writing them in **bold** print; also, for any non-terminal N, we use N^* to denote zero or more occurrences of N, and N^+ to denote one or more occurrences. When dealing with these (as types) later on in this chapter, we may, on occasion, take the liberty of regarding N^* and N^+ as being equivalent to *seq N* and seq_1 *N* respectively.

ABSTRACT SYNTAX

program	::=	[**Program** ident decl* block* stat$^+$]
block	::=	[**Proc** ident (param*) decl* block* stat$^+$]
decl	::=	sdecl | rdecl
sdecl	::=	ident **:** stype
rdecl	::=	ident **: record** sdecl$^+$ **endrec**
stype	::=	**Bool** | **Int** | **Nat** | ident | **array** const **to** const **of** stype | ↑ st
param	::=	**val** sdecl | **var** sdecl
stat	::=	vref$^+$ **:=** expr$^+$ |
		skip |
		if expr **then** stat$^+$ **else** stat$^+$ **fi** |
		while expr **do** stat$^+$ **od** |
		read (ident) |
		write (expr) |
		call ident (expr*) |
		new (vref st)
expr	::=	vref |
		const |
		expr binop expr |
		unop expr |
		(expr)
vref	::=	ident | vref subsc | ↑ (vref)
subsc	::=	[expr] | **.** ident
binop	::=	**+** | **-** | ***** | **/** | **mod** | **≠** | **=** | **<** | **≤** | **>** | **≥** |
		and | **or**
unop	::=	**+** | **-** | **not**
const	::=	**true** | **false** | **NULL** | number
number	::=	*natural or integer numbers*
ident	::=	*identifiers*

4.1.2 Context Conditions

The above syntax does not fully describe the acceptable class of programs. Further conditions could be introduced by using a more complicated grammar to restrict the valid set of programs, or by imposing them as preconditions on the semantic functions (so that some syntactically correct programs would be denied a semantics). However, it is more usual, and more convenient, to augment the syntax of a language with *context conditions*. For our language these conditions will include:

1) Procedure, variable and record field names at the same level must be unique
2) After their use in a declaration, record and procedure names may only be subsequently used in other declarations and the *call* statement respectively.
3) Arguments for procedure calls must exactly match the number and type of the corresponding formal parameters. Arguments matching variable parameters must be identifiers. We do not allow procedure names to be passed as arguments.
4) Input and output is restricted to integer variables and expressions only
5) We do not allow procedures to call themselves - ie. no recursion
6) We reserve the use of the identifiers *in*, *out* and *mem* (we will use them in § 4.4.6)

4.2 The Semantic Domain

The semantic domain consists of elements whose meaning is deemed to be "understood"; thus, constructing a function from the abstract syntax into this domain allows us to give a meaning to any phrase of the language. Two major criteria for the choice of a domain are that it should be useful for our ultimate purposes, and that *its* semantics should be unambiguously defined.

The semantic domain which we have chosen is that of Z schemas which, thanks to [Spiv89] is unambiguously defined. In his description of the semantics of Z, Spivey gives a syntax for the schema language, which he uses as his starting point. Since this language is therefore formally defined, we will use it as the target for our mapping function; that is, Spivey's syntax will define our semantic domain.

The parts of Spivey's syntax which we shall need are reproduced below, with an informal explanation. Our main use for this syntax is to use its components in the signatures of variables and functions. We regard our semantic domain as consisting of the following sets:

SPEC is the set of all Z specifications. This can consist of given set names, global variables or functions, or schema expressions.

SEXP contains all schema expressions. These may be individual schemas, or the disjunction, conjunction, quantification etc. of schemas.

SCHEMA is the set of all schemas, each of which consists of a declaration part and a predicate part.

DECL describes the declaration part of a schema. Its elements may be schema designators or identifier declarations. The latter consists of an identifier followed by a term.

PRED describes the predicate part of a schema

TERM is the set of all terms which are used in the signature of a variable declaration.

This includes other identifiers, schema designators, power sets, tuples etc.

IDENT is the set of all identifiers

We will assume that basic sets such as **N**, **Z** and **R** are available to us. It will prove useful to take as given the set *[WORDS]* which consists simply of any sequence of characters. In order to distinguish between (constant) elements of this set and identifiers, we will enclose the former in inverted commas: that is, *word* will be regarded as an identifier (ie. a variable name), whereas *"word"* is a constant, and a member of the set *WORDS* (in the same way that 1 is a constant, and member of the set **N**).

4.3 Strictness

There are two main approaches to dealing with undefinedness in programs - ie. programs which are syntactically correct and fulfil the context conditions, but which could be rendered meaningless depending on the values taken by variables at run time (exceeding array bounds is an example). We could add in checks to the semantics to rule out such situations - this would mean that our mapping was now partial over syntactically correct programs - or we could ensure that such programs are not mapped to anything meaningful. The latter approach would mean that our mapping is *strict* - meaningless programs are mapped into undefined elements of the semantic domain (possibly by introducing a distinguished "undefined" element.)

Our approach is similar to using a strict mapping in that a meaningless program will be mapped into a similarly meaningless specification. For example, an array reference with an index value outside the declared range is translated into a function application to a value which is not in the domain of that function. We will find it useful to introduce a special element, denoted $\perp$, which is distinguished by the fact that it is not an element of any set that we will be using to represent a type: this is the undefined element. We will also make use of a shorthand notation[1]: for any set T we will write $T^{\perp}$ when we mean $T \cup \{\perp\}$. We will also insist that all the operations which we will need over the integers, natural numbers and booleans are strict, in that if one of the arguments is undefined, then the result will be undefined.

4.4 Semantic Functions

The syntax of a programming language was described above by giving a set of basic elements (or terminal symbols), and then describing how phrases of the language may be constructed from these. Giving a semantics to the language mirrors this process: we first map the basic elements into the semantic domain, and then use this to construct mappings for the phrases from which they are built. In order to differentiate between pieces of the programming language and pieces of Z semantics, we will adopt the convention of enclosing syntactic elements (fragments of the program) in Strachey brackets - $[\![$ and $]\!]$.

The mapping for the operators and constants is trivial since, in most cases, the syntax is identical in both domains. However, the meaning which we give to the identifiers of the syntactic domain will serve as a basis for the rest of the semantics. The most

[1]Note that in some of the literature on semantics the undefined element is denoted by ω. Also, the union of some set T with this element is usually denoted T^{+}; we do not use this notation in order to avoid confusion with earlier notation for a non-empty sequence.

common approach is to construct a function, known as the environment, which describes the association between identifiers and elements of Z. Indeed, the environment function is often one of the best clues towards understanding a particular denotational description.

4.4.1 The environment

The environment function which we are about to describe will have three main uses:

* It will provide a mapping from identifiers representing program variables into appropriate Z identifiers
* It will be used for assigning the correct scope to these program variables
* It will be used to store the meaning of procedure blocks which are currently in scope

4.4.1.1 Definition of the environment

We will start with the third of these. The main part of our representation of procedures will be a schema which will correspond to the body of that procedure. However, if we wish to use it later on with a list of arguments, we will also need to have a list of the formal parameters (in the correct order, so that we can match them one for one with the arguments). We define `ProcEnv` to hold this information:

```
┌─ ProcEnv ──────────
│ formals : {VAL,VAR} ↔ ℕ → IDENT
│ body : SEXP
└──────────────────────
```

The *formals* are simply a sequence of the identifiers which represent the formal parameters, along with a tag saying whether they are call-by-value or call-by-variable.

The environment function will map identifiers into either DECL or ProcEnv, depending on whether they represent variable or procedure names respectively:

$$\text{ENV} \ == \ \text{ident} \rightarrow (\text{DECL} \cup \text{ProcEnv})$$

A declaration in Z (something of type DECL) is composed of a variable name (identifier) and a description of the set to which the variable belongs. An environment therefore provides a means of mapping a program identifier into the corresponding Z identifier, or to a declaration which, when introduced in a schema, will bring the variable into scope.

4.4.1.2 Functions dealing with the environment

We will next define some functions over environments which will come in useful later. The function ι retrieves the corresponding Z identifier to a program variable from a given environment

$$\iota : \text{ident} \rightarrow \text{ENV} \rightarrow \text{IDENT}$$

$$\iota = \lambda \ \text{id} : \text{ident}, \text{E} : \text{ENV} \cdot$$
$$\quad \mu \ \text{Id} : \text{IDENT} \mid \exists \ \text{Tm} : \text{TERM} \mid \text{E(id)} = (\text{Id} : \text{Tm})$$

MakeSch constructs a schema which consists of a declaration of all variables in the range of some environment (we assume that the mapping which is passed represents variables only and not procedures). This will be used as a signature when we are constructing schemas from statements in the function σ defined later.

| MakeSch : (ident $\rightarrow$ DECL) $\rightarrow$ SEXP
|―――――――――――
| $\forall$ id,i_1,...i_n : ident, D_1,...,D_n : DECL, Id: IDENT, Tm : TERM, Pr : ProcEnv $\cdot$
|
| MakeSch($\{i_1 \mapsto D_1, \ldots, i_n \mapsto D_n\}$) = MakeSch($\{i_1 \mapsto D_1\}$) ; ... ; MakeSch($\{i_n \mapsto D_n\}$)
| MakeSch($\{$id $\mapsto$ Id : Tm$\}$) = Id : Tm

4.4.1.3 Memory Allocation

Since we will be using pointers in our language, we will need to be able to distinguish between an identifer (representing a variable) and a "memory location". First of all, we will name the set of memory locations as *MLoc*; it will not be necessary to define this set further. We will let *GIVEN* be the set of all those sets which are formable using the integers, natural numbers and Booleans; basically, any identifier in our program will represent a member of this set. We justify the use of such a set by noting that it would be possible to statically analyse any given program and work out exactly the required sets; for our purposes, we will just assume it as a given set of our specification.

Now we can define the function:

$$\text{AT} : \text{MLoc} \rightarrow \text{GIVEN}$$

This function will "de-reference" a pointer - it will return the element that is stored at the "address". Note that we have defined the domain as *MLoc* and not *MLoc*$^{\perp}$, so that it is not possible to dereference *NULL* (which is mapped to $\perp$).

4.4.2 Declarations:

Now that we have defined a mapping for identifiers, our next step is to define a meaning for the parts of the program which operate on this mapping, the declarations. First of all, we need an auxiliary function to deal with the right-hand side of declarations (except records). We will assume that the set *Boolean* has been defined somewhere (we will attend to this later), and we will extend the basic sets **Z** and **N** by the undefined element. Also, we use the standard definition of arrays as functions from the elements of the index set into the array type.

τ : stype $\rightarrow$ ENV $\rightarrow$ TERM

$\forall$ E : ENV, id : ident, st : stype, a,b : const | a < b $\cdot$

τ [Bool] E = *Boolean*$^{\perp}$

τ [Int] E = $\mathbf{Z}^{\perp}$

τ [Nat] E = $\mathbf{N}^{\perp}$

τ [id] E = ι [id] E

τ [array a to b of st] E = {ε [a] E .. ε [b-1] E} $\rightarrow$ (τ [st] E)

τ [$\uparrow$ st] E = MLoc$^{\perp}$

We have defined anything which has been declared as a pointer to be of type *MLoc*; we will deal with the exact type of the corresponding Z identifer (as given by *AT*) when we deal with the *new* statement.

Next, using the function δ, we interpret declarations (of variables or parameters) as functions mapping environments to environments, where the output is the input environment plus the new declaration. We will want to ensure that each program identifier is mapped to a "new" Z identifier - one that has not been used for anything else. It is not simply enough to test that the new identifier is not in the environment, since it may have been overwritten by the definition of some local variable; what we want is to check that the variable is not, and has never been in the environment. We could define a set containing all the Z identifiers which have been used to date, but it would be cumbersome to carry around this extra "baggage" from function to function. Instead, let us define an ordering on identifiers: we'll call it $\prec$

$$\_ \prec \_ : \mathrm{IDENT} \times \mathrm{IDENT} \rightarrow \{\mathit{TRUE}, \mathit{FALSE}\}$$

$$\forall\ \mathrm{Id}_1, \mathrm{Id}_2, \mathrm{Id}_3 : \mathrm{IDENT} \cdot$$
$$\neg(\mathrm{Id}_1 \prec \mathrm{Id}_1)$$
$$((\mathrm{Id}_1 \prec \mathrm{Id}_2) \wedge (\mathrm{Id}_2 \prec \mathrm{Id}_3)) \Rightarrow (\mathrm{Id}_1 \prec \mathrm{Id}_3)$$

Thus $\prec$ is irreflexive and transitive; just what we need to ensure that if identifers are used in a sequence which is monotonic with reference to $\prec$, we will not use the same identifier twice. We are now ready to define δ:

$\delta : (\text{decl} \cup \text{param})^* \rightarrow \text{ENV} \rightarrow \text{ENV}$

$\forall\ \text{E} : \text{ENV},\ \text{dl} : \text{decl},\ \text{dls} : \text{decl}^*,\ \text{id}, \text{id}_1, \ldots \text{id}_n : \text{ident},\ \text{st}, \text{st}_1, \ldots \text{st}_n : \text{stype}\ |$

$\quad \exists\ \text{Id} : \text{IDENT}\ |\ (\forall\ (\text{Idt:Tm}) : \text{DECL}\ |\ (\text{Idt:Tm}) \in \text{ran(E)} \cdot \text{Idt} \prec \text{Id})$

$\delta\ [\![\,]\!]\ \text{E} = \text{E}$

$\delta\ [\![\text{dl dls}]\!]\ \text{E} = \delta\ [\![\text{dls}]\!]\ (\delta\ [\![\text{dl}]\!]\ \text{E})$

$\delta\ [\![\text{VAL id : st}]\!]\ \text{E} = \text{E} \oplus \{[\![\text{id}]\!] \mapsto \text{Id} : (\tau\ [\![\text{st}]\!]\ \text{E})\}$

$\delta\ [\![\text{VAR id : st}]\!]\ \text{E} = \text{E} \oplus \{[\![\text{id}]\!] \mapsto \text{Id} : (\tau\ [\![\text{st}]\!]\ \text{E})\}$

$\delta\ [\![\text{id : st}]\!]\ \text{E} = \text{E} \oplus \{[\![\text{id}]\!] \mapsto \text{Id} : (\tau\ [\![\text{st}]\!]\ \text{E})\}$

$\delta\ [\![\text{id : record id}_1 : \text{st}_1 \ldots \text{id}_n : \text{st}_n\ \text{endrec}]\!]\ \text{E} =$

$\quad \text{E} \oplus \{[\![\text{id}]\!] \mapsto (\text{Id} : \{\text{"id}_1\text{"}\} \rightarrow (\tau\ [\![\text{st}_1]\!]\ \text{E}) \cup \ldots \cup \{\text{"id}_n\text{"}\} \rightarrow (\tau\ [\![\text{st}_n]\!]\ \text{E}))\}$

We use the ordering imposed by $\prec$ above to ensure that Id does not already form a part of some declaration in the environment.

Note that the definition of (δ *[dl dls]* E) ensures that a declaration operates over an environment to which the preceding declarations have already been added. We deal with ordinary declarations by simply finding a suitable Z identifer, working out the Z equivalent of the type, and adding this new Z declaration to the environment as the image of the associated program identifier. The function overriding operator $\oplus$ ensures that we will only have *one* mapping for each program identifier - thus a local declaration of a variable will overwrite prior declarations (at a higher level) of variables with the same name.

We regard record declarations as introducing a new type: all records are treated as if they were functions from the field names into the respective field type. Thus we make use of the Z set *WORDS* which we introduced earlier in order to allow us to use the field names (exactly as they appear in the program) in the corresponding specification. Let us take a definition of a tree node as an example: we might use the record

```
tnode : record
        contents : Nat
        left : ↑ tnode
        right : ↑ tnode
endrec
```

Assuming that we select the Z identifier TNODE to represent the program identifier [tnode], we would get the following definition:

$$\mathrm{TNODE} : \{\text{"contents"}\} \rightarrow \mathbb{N}^{\perp} \cup \{\text{"left"}\} \rightarrow \mathrm{MLoc}^{\perp} \cup \{\text{"right"}\} \rightarrow \mathrm{MLoc}^{\perp}$$

Thus we regard *TNODE* as being a function whose domain consists of the three words "contents", "left" and "right", and whose range is $\{\mathbb{N} \cup MLoc\}^{\perp}$, with the added restriction that it will only map "contents" to something of type $\mathbb{N}^{\perp}$, and "left" and "right" to something of type $MLoc^{\perp}$.

4.4.3 Blocks

The next element for definition is a block. Throughout this discussion, let us assume that we have some function σ which will map statements and an environment into an appropriate schema (we will be defining this function later).

We defined *ProcEnv* above as consisting of two parts - *formals* and *body* which between them provide the information needed to describe a block in the environment. We will now describe a function which will give the appropriate semantics for a block in *ProcEnv*.

Firstly, we will need a list of the formal parameters of the block; the function π maps a list of parameters into a form suitable for inclusion in the *formals* part of a *ProcEnv*. For simplicity, we will treat something of type $param^*$ as though it were a sequence of *param*.

$$\pi : param^* \rightarrow ENV \rightarrow (\{VAL, VAR\} \leftrightarrow \mathbb{N} \rightarrow IDENT)$$

$$\begin{aligned}
\pi = \lambda\ & pms : param^*, E : ENV \cdot \\
& \mu\ plist : \{VAL, VAR\} \leftrightarrow \mathbb{N} \rightarrow IDENT \mid \\
& \quad \forall\, i : \mathbb{N} \cdot \ (pms(i) = [\![var\ id\ st]\!]) \Leftrightarrow \{VAR \mapsto i \mapsto (\iota\, [\![id]\!]\, E)\} \in plist \\
& \quad\quad\quad \wedge\ (pms(i) = [\![val\ id\ st]\!]) \Leftrightarrow \{VAL \mapsto i \mapsto (\iota\, [\![id]\!]\, E)\} \in plist
\end{aligned}$$

Each Z identifer representing a parameter is thus identified with its position in the sequence and whether it is either *VAR* or *VAL*. Note that the above function assumes that we have already added the parameters to the environment (presumably using the δ function defined above).

We regard a block in much the same way as declarations above: as representing a mapping from one environment to another. The output environment is just the input environment with the definitions of all the procedures in the block added to it.

When giving the semantics for blocks, it is essential that we respect the scoping rules of the language. We ensure that everything in scope for a particular block is in scope for its sub-blocks by making each block pass its environment (after the declarations have been added) to its sub-blocks. Also, we nest the definition of β for sequences of blocks to ensure that each block is in scope for succeeding blocks at the same level.

$\beta : \text{block}^* \rightarrow \text{ENV} \rightarrow \text{ENV}$

$\forall\ \text{E} : \text{ENV},\ \text{blks}_N,\ \text{blks} : \text{block}^*,\ \text{pms} : \text{param}^*,\ \text{dls} : \text{decl}^*,\ \text{sts} : \text{stat}^* \cdot$

$\beta\, [\![\,[\,]\,]\!]\ \text{E} = \text{E}$

$\beta\, [\![[\text{Proc N (pms) dls blks}_N\ \text{sts}]\ \text{blks}]\!]\ \text{E} =$

$\quad \beta\, [\![\text{blks}]\!]\ (\beta\, [\![[\text{Proc N (pms) dls blks}_N\ \text{sts}]]\!]\ \text{E})$

$\beta\, [\![[\text{Proc N (pms) dls blks}_N\ \text{sts}]]\!]\ \text{E} = \text{E} \oplus \{[\![\text{N}]\!] \mapsto \text{PE}\}$

where $\text{PE} : \text{ProcEnv}\ |$

$\qquad \text{PE.formals} = \pi\, [\![\text{pms}]\!]\ \text{E}_2$

$\qquad \text{PE.body} = (\Omega\, [\![\text{dls}]\!]\ \text{E}_2) \wedge (\sigma\, [\![\text{sts}]\!]\ (\beta\, [\![\text{blks}]\!]\ \text{E}_2))$

and $\text{E}_2 : \text{ENV}\ |$

$\qquad \text{E}_2 = \delta\, [\![\text{dls}]\!]\ (\delta\, [\![\text{pms}]\!]\ \text{E})$

The net result of applying β to a block and an environment is to add a mapping for that block to the environment. Note that the same environment E_2 is passed to both σ and π which are used to create the procedure's representation in the new environment.

4.4.4 Initialisation

We made use of a function Ω in the previous definition - the purpose of this function is to initialise the local variables of the block to the appropriate value. We define it as:

$\Omega : \mathrm{decl}^* \rightarrow \mathrm{ENV} \rightarrow \mathrm{SEXP}$

$\forall\ E : \mathrm{ENV},\ dl : \mathrm{decl},\ dls : \mathrm{decl}^*,\ id, id_1, \ldots id_n : \mathrm{ident},\ st, st_1, \ldots st_n : \mathrm{stype}\ \cdot$

$\Omega\ [\![\ ]\!]\ E\ =\ \mathit{TRUE}$

$\Omega\ [\![dl\ dls]\!]\ E\ =\ (\Omega\ [\![dl]\!]\ E) \wedge (\Omega\ [\![dls]\!]\ E)$

$\Omega\ [\![id : st]\!]\ E\ =\ (\omega\ (\tau\ st\ E)\ E)$

$\Omega\ [\![id : \mathrm{record}\ id_1 : st_1 \ldots id_n : st_n\ \mathrm{endrec}]\!]\ E\ =\ \mathit{TRUE}$

The function ω simply works out whether the initialisation that corresponds to a particular type is the undefined element or the empty set, and is defined as:

$\omega : \mathrm{TERM} \rightarrow \mathrm{ENV} \rightarrow \{\bot, \{\}\}$

$\forall\ Tm : \mathrm{TERM},\ E : \mathrm{ENV}\ \cdot$

$Tm \in \{N^\bot, Z^\bot, \mathit{Boolean}^\bot, \mathit{MLoc}^\bot\}\ \Rightarrow\ (\omega\ Tm\ E) = \bot$

$\exists\ Tm_1, Tm_2 : \mathrm{TERM} \cdot Tm = Tm_2 \rightarrow Tm_2\ \Rightarrow\ (\omega\ Tm\ E) = \{\}$

$Tm \in \mathrm{IDENT} \Rightarrow (\exists\ Tm_1 : \mathrm{TERM}\ |\ Tm : Tm_1 \in \mathrm{ran}(E) \cdot (\omega\ Tm\ E) = (\omega\ Tm_1\ E))$

The net result of this is to ensure that any program which attempts to use variables on the right-hand-side of an assignment statement before they have been *explicitly* initialised (by assigning them a value) will be meaningless.

4.4.5 Expressions

The next step from the semantics of identifiers is to define the semantics for expressions containing them. This is done fairly routinely using the following function:

$$
\begin{array}{|l}
\varepsilon : \text{expr} \rightarrow \text{ENV} \rightarrow \text{TERM} \\
\hline
\forall\, \text{E} : \text{ENV},\ \text{ex}, e_1, e_2 : \text{expr},\ \text{id} : \text{ident},\ \text{vr} : \text{vref},\ \text{bop} : \text{binop},\ \text{uop} : \text{unop},\ \text{c} : \text{const} \,\cdot \\
\\
\varepsilon \llbracket e_1 \text{ bop } e_2 \rrbracket\, \text{E} \;=\; (\varepsilon \llbracket e_1 \rrbracket\, \text{E})\ (\sigma \llbracket \text{bop} \rrbracket)\ (\varepsilon \llbracket e_2 \rrbracket\, \text{E}) \\
\varepsilon \llbracket \text{uop ex} \rrbracket\, \text{E} \;=\; (\sigma \llbracket \text{uop} \rrbracket)\ (\varepsilon \llbracket \text{ex} \rrbracket\, \text{E}) \\
\varepsilon \llbracket (\text{ex}) \rrbracket\, \text{E} \;=\; (\varepsilon\ \text{ex}\ \text{E}) \\
\varepsilon \llbracket \text{vr}[\text{ex}] \rrbracket\, \text{E} \;=\; (\varepsilon \llbracket \text{vr} \rrbracket\, \text{E})\ (\varepsilon \llbracket \text{ex} \rrbracket\, \text{E}) \\
\varepsilon \llbracket \text{vr.id} \rrbracket\, \text{E} \;=\; (\varepsilon \llbracket \text{vr} \rrbracket\, \text{E})\ (\text{"id"}) \\
\varepsilon \llbracket \uparrow (\text{vr}) \rrbracket\, \text{E} \;=\; \text{AT}(\varepsilon \llbracket \text{vr} \rrbracket\, \text{E}) \\
\varepsilon \llbracket \text{id} \rrbracket\, \text{E} \;=\; (\iota \llbracket \text{id} \rrbracket\, \text{E}) \\
\varepsilon \llbracket \text{NULL} \rrbracket\, \text{E} \;=\; \bot \\
\varepsilon \llbracket \text{true} \rrbracket\, \text{E} \;=\; \mathit{TRUE} \quad \textit{... and so on for the other constants}
\end{array}
$$

We have assumed the existence of some (trivial) function σ, which maps unary and binary operators into their Z counterparts; we shall not bother to define it further. Similarly, we will not elaborate on the mapping for constants. Array and record references have been defined as function application, as discussed earlier. We make use of the function *AT* to give a meaning for the dereferencing of a pointer.

4.4.6 Statements

We are now in a position to define a function σ which gives a semantics to individual statements. Since this function is central to the whole definition of a program, we divide our discussion into five parts: semantics for the assignment statement, the *new* statement, input/output statements, compound statements, and the procedure call. For reference, we will give the definition of the whole function first, and then explain how it works afterwards (we will also define the functions γ, ξ and α later).

4.4.6.1 The basic concept behind σ

The central idea of what we are trying to do above goes something like this: At the start of every block of statements we have a set of variables, say $x_1...x_n$, which are introduced by *Sch*. Each statement introduces a new set of variables $x_1'...x_n'$; the relationship between each x_i and the corresponding x_i' depends on the statement. Any variables which are not affected by a statement are deemed to be equal to the corresponding variable before the statement - this is indicated above by the presence of Ξ *Sch* as a basis for some of the definitions. The assignment, read and write statements replace old equalities with new ones, while the conditional and iterative statements provide assertions about the variables.

We introduce the relevant variables into the schema by the inclusion of *Sch*, which is formed from those variables currently listed in the environment. We can use the range restriction operator $\triangleright$ to make sure that only variable declarations are passed to *MakeSch* (and not procedure definitions). Thus for the simplest statement, [skip], is just mapped to the identity operation over the environment, Ξ *Sch*.

$\sigma : \mathrm{stat}^{+} \rightarrow \mathrm{ENV} \rightarrow \mathrm{SEXP}$

$\forall\, \mathrm{E} : \mathrm{ENV},\ st : stat,\ sts, sts_1, sts_2 : stat^{*},\ v, v_1..v_n : vref,\ e, e_1..e_n : expr,\ args : expr^{*},$
$\quad Sch : SEXP \mid Sch = MakeSch(DECL \blacktriangleright \mathrm{E}) \cdot$

$\sigma \llbracket st\ sts \rrbracket\, \mathrm{E} = (\sigma \llbracket st \rrbracket\, \mathrm{E})\ ;\ (\sigma \llbracket sts \rrbracket\, \mathrm{E})$
$\sigma \llbracket\ \rrbracket\, \mathrm{E} = \Xi\, Sch$

$\sigma \llbracket skip \rrbracket\, \mathrm{E} = \Xi\, Sch$

$\sigma \llbracket v_1,..,v_n := e_1,..,e_n \rrbracket,\ \mathrm{E}) =$
$\quad \Xi\, Sch \setminus ((\xi \llbracket v_1 \rrbracket\, \mathrm{E})',\ \ldots\ (\xi \llbracket v_n \rrbracket \mathrm{E})') \wedge$
$\quad [\Delta\, Sch \mid {}^{\wedge}/\ \gamma\langle \alpha(\llbracket v_1 \rrbracket, (\varepsilon \llbracket e_1 \rrbracket\, \mathrm{E}))\, \mathrm{E},\ \ldots,\ \alpha(\llbracket v_n \rrbracket, (\varepsilon \llbracket e_n \rrbracket\, \mathrm{E}))\, \mathrm{E} \rangle]$

$\sigma \llbracket read\ (v) \rrbracket\, \mathrm{E} =$
$\quad \Xi\, Sch \setminus ((\iota \llbracket v \rrbracket\, \mathrm{E})',\ inseq') \wedge [(\iota \llbracket v \rrbracket\, \mathrm{E})' = head(inseq)] \wedge [inseq' = tail(inseq)]$

$\sigma \llbracket write\ (e) \rrbracket\, \mathrm{E} = \Xi\, Sch \setminus (outseq') \wedge [outseq' = outseq \frown (\varepsilon \llbracket e \rrbracket\, \mathrm{E})]$

$\forall\, vr : vref,\ ty : stype,\ Id : IDENT \mid Id = (\xi \llbracket vr \rrbracket\, \mathrm{E}) \wedge Ty = (\tau \llbracket ty \rrbracket\, \mathrm{E}) \cdot$
$\sigma \llbracket new\ (vr\ ty) \rrbracket\, \mathrm{E} =$
$\quad \Xi\, Sch \setminus (AT', Id') \wedge [\Delta\, Sch \mid (Id' \notin dom(AT)) \wedge \alpha\, (\llbracket \uparrow vr \rrbracket, \omega\ Ty\ \mathrm{E})\ \mathrm{E}]$

$\sigma \llbracket if\ e\ then\ sts_1\ else\ sts_2 \rrbracket\, \mathrm{E} =$
$\quad [Sch \mid (\varepsilon \llbracket e \rrbracket\, \mathrm{E}) \wedge (\sigma \llbracket sts_1 \rrbracket\, \mathrm{E})] \vee [Sch \mid \neg(\varepsilon \llbracket e \rrbracket\, \mathrm{E}) \wedge (\sigma \llbracket sts_2 \rrbracket\, \mathrm{E})]$

$\sigma \llbracket while\ e\ do\ sts\ od \rrbracket\, \mathrm{E} =$
$\quad [Sch \mid (\varepsilon \llbracket e \rrbracket\, \mathrm{E}) \wedge (\sigma \llbracket sts \rrbracket\, \mathrm{E})]^{*} \wedge [Sch' \mid \neg(\varepsilon \llbracket e \rrbracket\, \mathrm{E})']$

$\forall\, P : ident,\ args : expr^{*}, \mid P = \mathrm{E}(\llbracket P \rrbracket) \cdot$

$\sigma \llbracket call\ P\ (args) \rrbracket\, \mathrm{E} = \quad ([Sch \mid a_1 = f_1 \wedge \ldots\ldots\ a_k = f_k]$
$\qquad\qquad \wedge\ P.body\ [f_{k+1}/a_{k+1},\ \ldots\ldots\ f_m/a_m,\ f_{k+1}'/a_{k+1}',\ \ldots\ldots\ f_m'/a_m'])$
$\qquad\qquad \upharpoonright\ \Delta\, Sch$

where

$\quad k, m : \mathbb{N},\ f_1,\ldots,f_m : IDENT,\ a_1,\ldots,a_m : TERM \mid$
$\quad \{f_1 \ldots f_k\} = ran\ (P.formals(\!|\{\mathit{VAL}\}|\!)) \wedge$
$\quad \{f_{k+1} \ldots f_m\} = ran\ (P.formals(\!|\{\mathit{VAR}\}|\!)) \wedge$
$\quad \forall\, i,j : \{1..m\} \cdot (P.formals(\!|\{\mathit{VAL}, \mathit{VAR}\}|\!)(i) = f_j) \Leftrightarrow ((\varepsilon \llbracket args \rrbracket(i)\ \mathrm{E}) = a_j)$

4.4.6.2 The Assignment Statement

Basically, what we want to do is to assert the equality of the (post-execution) left- and (pre-execution) right-hand-side of the assignment statement; that is, we wish to assert that for a simple assignment to an identifier such as **[x := e]**, in some environment E, we will have the assertion (ε **[x]** E)′ = (ε **[e]** E) holding after the statement.

If the assignment consists of just variable names, then this assertion will suffice as it stands, but if array or record references are involved, we must be a little more careful. Take as an example the assignment **[**a s_1 ... s_n := z**]** where *a* is an array or record name, and each of $s_1,...,s_n$ are subscripts, and *z* is a suitable expression, and suppose that these are mapped to *A* and $S_1,...S_n$ and *Z* respectively. After the assignment has taken place we will want to assert that $A'(S_1) ... (S_n) = Z$. If only one subscript is involved (ie. n = 1), we can simply state that $A' = A \oplus \{S_1 \mapsto Z\}$. If we have two subscripts, then we must assert something like $A' = A \oplus \{S_1 \mapsto (A(S_1) \oplus \{S_2 \mapsto Z\})\}$. For larger numbers of subscripts this notation could quickly become cumbersome, so we will define a shorthand version called ⊗, which, for the assignment with *n* subscripts, will allow us to write $A' = A \otimes \{(S_1,...S_n) \mapsto Z\}$.

$$[X_1,...X_n,Y]$$

$$\_\otimes\_ : (X_1 \rightarrow ... \rightarrow (X_n \rightarrow Y)) \times ((X_1 \times ... \times X_n) \rightarrow Y) \rightarrow (X_1 \rightarrow ... \rightarrow (X_n \rightarrow Y))$$

$$\begin{aligned}
&\otimes = \lambda\, R : (X_1 \rightarrow ... \rightarrow (X_n \rightarrow Y)),\ N : ((X_1 \times ... \times X_n) \rightarrow Y) \bullet \\
&\quad \mu\, R' : (X_1 \rightarrow ... \rightarrow (X_n \rightarrow Y)) \bullet \\
&\qquad \forall\, i, (i_1,...i_n) : (X_1 \times ... \times X_n) \mid i = (i_1,...i_n) \bullet \\
&\qquad\quad ((i \in \mathrm{dom}\, N) \Rightarrow (R'(i_1)...(i_n) = N(i))) \wedge ((i \notin \mathrm{dom}\, N) \Rightarrow (R'(i_1)...(i_n) = R(i)))
\end{aligned}$$

Thus the $\otimes$ function simply builds a new R' from the old R, by replacing some of the mappings of R with new mappings from N. We could not use the function overriding operator $\oplus$ here since R and N do not have exactly the same domain (and are thus not of the same type), and the definition of $\oplus$ (as in [Spiv89, § 4.3) is based on domain corestriction. The domain of R (and of R') is just X_1, while the domain of N is the n-tuple $(X_1 \times \ldots \times X_n)$.

The purpose of the function α (as used in the function σ) is to take what is basically the right- and left-hand-side of a single assignment, and build an equation that represents the situation which we would like to hold after that assignment.

$\alpha : (\text{vref} \times \text{TERM}) \rightarrow \text{ENV} \rightarrow \text{TERM}$

$\forall\ \text{id} : \text{ident},\ \text{tm} : \text{TERM},\ \text{vr} : \text{vref},\ s_1,\ldots s_n : \text{subsc},\ E : \text{ENV}\ \cdot$

$\alpha\ ([\![\text{id}]\!],\text{tm})\ E\ =\ (\iota\ [\![\text{id}]\!]\ E)' = \text{tm}$

$\alpha\ ([\![\text{id}\ s_1 \ldots s_n]\!],\text{tm})\ E\ =\ (\iota\ [\![\text{id}]\!]\ E)' = (\iota\ [\![\text{id}]\!]\ E) \otimes \{(S_1,\ldots S_n) \mapsto (\text{tm})\}$

$\alpha\ ([\![\uparrow(\text{vr})]\!],\text{tm})\ E\ =\ AT' = AT \oplus \{(\varepsilon\ [\![\text{vr}]\!]\ E) \mapsto \text{tm}\}$

$\alpha\ ([\![\uparrow(\text{vr})\ s_1 \ldots s_n]\!],\text{tm})\ E\ =\ AT' = AT \otimes \{((\varepsilon\ [\![\text{vr}]\!]\ E),S_1,\ldots S_n) \mapsto \text{tm}\}$

where, for i : 1 .. n, we have the n equations:

$((\exists\ \text{id}_1 : \text{ident} \mid s_i = [\![.\text{id}_1]\!]) \Rightarrow (S_i = \text{"id}_1\text{"})) \wedge$

$((\exists\ \text{ex}_1 : \text{expr} \mid s_i = [\![[\text{ex}_1]]\!]) \Rightarrow (S_i = \varepsilon\ [\![\text{ex}_1]\!]\ E))$

We are now at the stage where we can map an assignment statement incorporating n concurrent assignments, $[\![v_i := e_i]\!]$, into n separate equations looking like $l_i' = r_i$. If we were to use the sequential composition operator to combine these equations then we would be treating them as sequential assignments, so evidently we will wish to conjoin them.

However, consider for some array B, the assignment $[\![B[i], B[j] := x, y]\!]$. As it stands, our definition would produce something like: $B' = B \otimes \{I \mapsto X\} \wedge B' = B \otimes \{J \mapsto Y\}$, whereas we really would want to combine this as: $B' = B \otimes \{I \mapsto X, J \mapsto Y\}$. We therefore will define a "grouping" function γ to do this for us. To make things a little easier, we will form a sequence from our equations, and thus γ will operate over sequences of TERMs:

$$\gamma : \text{seq TERM} \rightarrow \text{seq TERM}$$

$$\forall\, S : \text{seq TERM},\ l : \text{IDENT},\ t_1, t_2 : \text{TERM} \mid \text{head } S = (l' = l \otimes \{t_1\}) \cdot$$

$$(\#S = 1) \;\Rightarrow\; (\gamma(S) = S)$$
$$\exists\, i : \mathbb{N} \cdot (i \neq 1) \wedge (S(i) = (l' = l \otimes \{t_2\})) \Rightarrow$$
$$\quad \gamma(S) = \gamma(\text{tail } (S \oplus \{i \mapsto (l' = l \otimes \{t_1, t_2\})\}))$$
$$\nexists\, i : \mathbb{N} \cdot (i \neq 1) \wedge (S(i) = (l' = l \otimes \{t_2\})) \Rightarrow$$
$$\quad \Rightarrow\; \gamma(S) = (\text{head } S) \frown \gamma(\text{tail } S)$$

The result of applying γ will be a sequence of equations (where no term l will appear on the left-hand-side of two separate equations), so we will want to convert this into a conjunction of the equations. To do this we need to distribute the conjunction through the sequence; we thus use the conjunctive equivalent to distributed concatenation $\frown/$, which we write as $\wedge/$, and could define (as per [Spiv89] § 4.5) as

$\hat{}/ : \text{seq TERM} \rightarrow \text{TERM}$

$\hat{}/\langle\,\rangle = \langle\,\rangle$

$\forall\ t : \text{TERM} \cdot \hat{}/\langle t\rangle = t$

$\forall\ q,r : \text{seq TERM} \cdot \hat{}/(q \frown r) = (\hat{}/q) \wedge (\hat{}/r)$

We now have a set of equations which specify the effect of the assignment statement on the variables concerned; the last step is then to add in an assertion stating the invariance of all the other variables. This assertion is just Ξ *Sch* which has had all the "changing" variables hidden: we use the function simple ξ to extract the variable which is to be hidden, where ξ is defined as:

$\xi : \text{vref} \rightarrow \text{ENV} \rightarrow \text{IDENT}$

$\forall\ \text{id} : \text{ident},\ \text{vr} : \text{vref},\ \text{ss} : \text{subsc},\ \text{E} : \text{ENV} \cdot$

$\xi\, [\![\uparrow(\text{vr})]\!]\, \text{E} \;=\; \text{AT}'$

$\xi\, [\![\text{vr ss}]\!]\, \text{E} \;=\; \xi\, [\![\text{vr}]\!]\, \text{E}$

$\xi\, [\![\text{id}]\!]\, \text{E} \;=\; (\iota\, [\![\text{id}]\!]\, \text{E})'$

4.4.6.3 Input and Output

We regard input and output as operations on the special variables *inseq* and *outseq*, which will both be declared (in § 4.4.6) as sequences of integers. One reason for this is that it models the idea of a "program input" and a "program output" which can both be dealt with by a meaning function (such as M_{IO} as mentioned in chapter 1). This also corresponds to the idea of a "standard input" and "standard output" as being actual entities which can be changed by the program itself (with a larger set of commands, of course) or by its calling environment. We might choose to allow some sort of redirection by changing the variables *inseq* or *outseq*.

There are also less high-minded reasons than this for not allowing individual schemas have access to specific identifiers representing input sources and output destinations. First of all, when we come to analyze this semantic definition in terms of the calculus of chapter 2, it will simplify things if we only have to consider schemas as consisting of primed and unprimed variables, and can disregard the possibility of there being variables with other (special) decorations. Also, we will be making frequent use of schema composition, since this will model statement composition in the program, and if inputs and outputs were involved, then the I/O variables (those decorated with *?* and *!*) would be all lumped in together in the resulting schema. As we have defined it we will only have one special variable for each, and there can be no ambiguity as to the sequence of input or output of values.

The definitions of both statements are fairly routine. The read statement is similar to a single assignment, except that we must also note the fact that we have removed the front element from the input sequence, *inseq*. For the write statement, we must assert that we have added a new element to the sequence *outseq*.

4.4.6.4 The *new* statement

The easiest way to explain the *new* statement is to look at an example. Suppose we had processed the declaration [*p* : ↑ *ptype*] which had added the declaration *P* : *MLoc*$^\perp$ to the environment. Then the statement [*new(p ptype)*] would result in two values changing: the value of *P* itself, and the value of the function *AT* at *P*. We do not specify the value of *P* exactly, except to say that it is some "new" location that was not previously in *AT*; the new value of the function *AT* at *P* will be the initialisation value corresponding to *P*. Thus the statement will produce the schema:

$$\Xi\ \mathrm{Sch} \,/\, (\mathrm{AT}', \mathrm{P}') \ \wedge\ [\Delta\ \mathrm{Sch} \mid (\mathrm{P}' \notin \mathrm{dom}(\mathrm{AT})) \wedge (\mathrm{AT}' = \mathrm{AT} \oplus \{\mathrm{P}' \mapsto \perp\})]$$

4.4.6.5 Compound Statements

Sequential composition in the programming language is modelled using ;, the schema composition operator. This simply has the effect of identifying the post state of the first schema with the pre state of the second, forming a schema whose pre and post states are the pre and post states of the first and second schema respectively. In order for this schema to be defined in general, we must know that the variables in the first and second schema can be matched up - we know this is so above, since all schemas will have *Sch* as their declaration part.

The conditional, or if-then-else, statement is represented by the disjunction of two schemas, each representing one of the alternative branches of the statement. We add an assertion that the guard holds to the schema representing the first branch, and an assertion that the negation of the guard holds to the schema for the second branch.

In defining a meaning for the while loop, we apply the * operation to schemas. This denotes the reflexive-transitive closure of the composition operation, ;. That is, if S is a schema, then $S^* = S^0 \vee S^1 \vee S^2 \vee .....$ where S^0 is the identity schema and each $S^n = S^{n-1} \, ; S$.

In the above situation, S^0 is Ξ *Sch* - this corresponds to the situation where no iterations take place. Each S^i in the disjunction represents the possibility of the loop iterating i times and then terminating. Since we do not impose any upper bound on i, the reflexive-transitive closure describes all iterations to infinity, thus including the situation where the loop does not terminate at all.

4.4.6.6 Procedure Calls

We now need to take care of the one remaining type of statement, the procedure call. Just as a reminder, we dealt with the procedure call in the definition of σ as follows:

$$
\begin{array}{|ll}
\sigma [\![\text{call P (args)}]\!] \, \mathrm{E} \; = & ([\text{Sch} \mid a_1 = f_1 \wedge ..... \, a_k = f_k] \\
 & \wedge \;\; \text{P.body} \, [f_{k+1}/a_{k+1}, ... \, f_m/a_m, f_{k+1}{}'/a_{k+1}{}', ... \, f_m{}'/a_m{}']) \\
 & \upharpoonright \;\; \Delta \, \text{Sch} \\
 & \\
\text{where} & \\
\quad k, m : \mathbb{N}, \;\; f_1,...,f_m : \text{IDENT}, \;\; a_1,...,a_m : \text{TERM} \mid & \\
\quad \{f_1 \, ... \, f_k\} = \text{ran (P.formals} (\!|\{\mathit{VAL}\}|\!)) \;\; \wedge & \\
\quad \{f_{k+1} \, ... \, f_m\} = \text{ran (P.formals} (\!|\{\mathit{VAR}\}|\!)) \;\; \wedge & \\
\quad \forall \, i,j : \{1..m\} \cdot (\text{P.formals}(\{\mathit{VAL},\mathit{VAR}\})(i) = f_j) \;\; \Leftrightarrow \;\; ((\varepsilon \, [\![\text{args}]\!](i) \, \mathrm{E}) = a_j) &
\end{array}
$$

The basis of this definition is the schema which represents P in the environment, namely E([P]).body. However, we then need to allow for the substitution of variable and value parameters, and to take the "internal" variables of the procedure back out of scope.

The predicate in the second part of the definition tells us that $f_1 \ldots f_k$ are the call-by-value formal parameters, $f_{k+1} \ldots f_m$ are the call-by-variable parameters and for each parameter f_i, the corresponding argument is a_i.

We deal with the value parameters by simply introducing a sequence of predicates before the schema which equate the start value of the parameters with their respective arguments. Variable parameters are easily taken care of by replacing them with the appropriate argument throughout the body of the schema.

We then take all the variables of the procedure which are not global out of scope by projecting the schema onto *Sch* (which has the effect of hiding any variables which are not in *Sch*).

4.4.7 Programs:

We now have all the tools necessary to provide a semantics for any program written in the language defined earlier - all that is left is to define a function tying it all together.

We first need to define the starting environment, *GlobEnv* for the program, which has just three elements. The identifiers *in*, *out* and *mem* are just "dummy" identifiers, whose only purpose is to get *inseq*, *outseq* and *AT* into the range of the environment - this ensures that when we form some schema *Sch* from the environment we will know that Ξ *Sch* will include $(inseq' = inseq) \wedge (outseq' = outseq) \wedge (AT' = AT)$.

$$\mathrm{GlobEnv} : \mathrm{ENV}$$

$$\mathrm{GlobEnv} = \{ [\![\mathrm{in}]\!] \mapsto \mathrm{inseq} : \mathrm{seq}\ \mathbb{Z},\ [\![\mathrm{out}]\!] \mapsto \mathrm{outseq} : \mathrm{seq}\ \mathbb{Z},\ [\![\mathrm{mem}]\!] \mapsto \mathrm{AT} : \mathrm{MLoc} \rightarrow \mathrm{GIVEN} \}$$

We can define the initial values of these elements using the following schema:

INIT

$\mathrm{input?}, \mathrm{output!} : \mathrm{seq}\ \mathbb{Z}$

$\mathrm{inseq}, \mathrm{outseq} : \mathrm{seq}\ \mathbb{Z}$

$\mathrm{AT} : \mathrm{MLoc} \rightarrow \mathrm{GIVEN}$

$\mathrm{inseq} = \mathrm{input?}$

$\mathrm{outseq} = \langle\ \rangle$

$\mathrm{AT} = \{\ \}$

We identify the start value of *inseq* with the input to the whole program, represented by the variable *input?*. The other two initialisations assert that we start with an empty output, and no "memory locations" have been assigned to program identifiers.

We are now ready to define the function ϕ, which maps a program into a specification, as was the original purpose of the exercise. Its definition is somewhat similar to that of β earlier, except that the environment passed to δ in this case is *GlobEnv*.

$\phi : \text{program} \rightarrow \text{SPEC}$

$\forall\ \text{P} : \text{ident},\ \text{blks} : \text{block}^{*},\ \text{dls} : \text{decl}^{*},\ \text{sts} : \text{stat}^{*},\ \text{E} : \text{ENV}\ |$

$\text{E} = (\delta\ [\text{dls}]\ \text{GlobEnv}) \cdot$

$\phi\ [[\text{Program P dls blks sts}]] =$

[SPEC, SEXP, SCHEMA, DECL, PRED, TERM, IDENT]

[GIVEN] [WORDS] [MLoc]

Boolean == {*TRUE*, *FALSE*}

INIT

$(\Omega\ [\text{dls}]\ \text{E}) \wedge (\sigma\ [\text{sts}]\ (\beta\ [\text{blks}]\ \text{E}))$

$[(\text{input?} = \text{inseq}') \wedge (\text{output!} = \text{outseq}')]$

CHAPTER 5 - ANALYSIS OF THE SEMANTICS

In this chapter we study issues relating to the semantic definition given in chapter 4, with the purpose of demonstrating how we can reason about programs using those semantics. We justify some of the decisions made by relating our definition to familiar concepts in the realm of axiomatic semantics using the notation for dealing with Z schemas which was introduced in chapter 2. We also examine some of the issues from chapter 3 in the light of our semantics, and discuss their implications.

5.1 What was specified?

The most familiar way of looking at a specification is as a description of something which we would hope to implement somewhere down the line. We did not write the specifications given in the previous chapter with an implementation in mind, but merely to provide a formal, unambiguous description of how a program could be mapped into the Z notation. However, it may help our understanding of the specification if we ask what an "implementation" of the specification would involve.

First of all, let us consider what the preceding chapter actually deals with. We gave at the start an abstract syntax for a simple programming language, and we gave a sketchy outline of the syntactic classes which go to make up a Z specification. We then presented a specification of a number of functions to map the programming language into what we claimed was a corresponding specification. Therefore we are dealing with two specifications: the "transformation" specification which defined the mapping functions, and the "target" specification which was what we were trying to map the programming language to.

An implementation of the transformation specification would be a program which took a program (text) as input, and produced a specification (text) as output. Implementation details would thus involve worrying about parsing the program, and about presenting the resulting specifications in an orderly manner. Those elements of the specification which we have not described in full detail (such as the procedure for choosing the Z identifier to correspond to a newly-declared program variable) would also have to be thrashed out. Ambitious implementations might even try to tidy up the resulting specification by simplifying some of the schemas.

We already have an implementation of the target specification - the program that was the source of the mapping; we constructed the mapping so that this would be the case. (Accordingly we could even view the abstract syntax as a language for expressing implementations.) The target specification describes a mapping (or function, since the language is deterministic) from the initial values of the input, output and outermost variables of the program to their final values - an implementation is any program which also correctly performs this mapping. The target specification thus describes a set of programs which are semantically equivalent to the program that we started with.

However, we could also take the view that the target specification was just a model of the abstract syntax. Suppose someone had constructed a new language without giving it a formal semantics, and wanted us to provide one. We would expect to be given a definition of the syntax of the language (using BNF or an equivalent), and some sort of informal description of how the language works. From this description we could then construct the appropriate mapping functions, as in the previous chapter, and provide a semantics for the language. This process is analogous to constructing a model of a system given an informal description of how that system works; this would be the starting point for any refinements towards an implementation. If we did not have the original programming language available to us, then perhaps the ultimate implementation from the specification would be an interpreter for the language.

5.2 The Iterative Statement

Usually in denotational semantics recursion is dealt with by treating the meaning of recursively defined functions as the fixpoint of an equation. In fact, this approach is central to the definition of all parts of the language in this manner, since the mathematical basis for the use of fixpoint equations imposes certain constraints on the type of domain over which they may be used. By eliminating recursion from our small language given in chapter 4 we have managed to side-step this issue and simplify our presentation of the semantics; for our purposes the language will be adequate, since we will also endeavour to side-step the use of recursion in our examples. Some work has been done on relating the fixpoint approach with program specifications (eg. in [Hoar87]), but we will not reflect on the matter any further at this point.

Also, denotational semantics tends to regard iteration as a special case of recursion. Since we do not allow recursion, it seems proper that we should consider further the definition of iteration in our language at this point. The definition in terms of the reflexive-transitive-closure operation is unlikely to cause any intuitive difficulties, but there are still some issues which merit a closer look.

It is usual in axiomatic-type semantics to provide a meaning for the iterative statement in terms of an invariant. Although we have not defined it as such in our semantics, we will show later in this chapter how the two concepts can be reconciled. We chose not to introduce the invariant in the previous chapter in order to provide a semantics that was, to some degree, mechanisable. That is, the "translation" involved in the previous chapter does not involve any real decision-making; the only choices involve routine matters such as keeping track of the names of identifiers. To introduce the invariant at this point would have meant that we would have to do much of our reasoning about the program while still dealing with it as expressed in the programming language: this clearly was not the intention of providing the semantics in the first place. Also, if we had chosen an invariant that was too weak at this point, then any backtracking would have to go right back to the original program, even though all the intervening work would have involved reasoning about the specification in Z. Our definition captures the full meaning of the iterative statement in a manner that is directly relatable to the original program, even if it is a little difficult to work with. When we come to choosing an invariant for the loop, we will be working with Z schemas, and we will have all the information expressed by that loop available to us; this was the aim of providing the semantics in the first place.

5.2.1 Uniqueness of *while* loop

We have defined the *while* loop as an infinite disjunction of schemas. It seems natural to ask whether or not the execution of these schemas are mutually exclusive; that is, is is possible for more than one of the terms in the disjunction to be true for the same values of the before-state. Intuitively we expect only one of the terms to be true - it should be the term which corresponds exactly to the number of iterations of the loop when started in that particular before-state.

Let us first of all expand the definition of the *while* statement. We have been given that:

$$\forall\, \mathrm{E} : ENV,\ sts : stat^*,\ e : expr,\ Sch : SEXP \mid Sch = MakeSch(DECL \blacktriangleright \mathrm{E}) \cdot$$

$$\sigma\,([\![\text{while } e \text{ do } sts \text{ od}]\!],\ \mathrm{E}) = [Sch \mid \varepsilon([\![e]\!],\mathrm{E}) \wedge \sigma([\![sts]\!],\mathrm{E})]^* \wedge [Sch' \mid \neg\varepsilon([\![e]\!],\mathrm{E})']$$

Since the environment E is constant throughout, we abbreviate the schemas Ξ *Sch* as *Id*, $\varepsilon([\![e]\!],E)$ as *E*, and $\sigma([\![sts]\!],E)$ as *S*. If we use these abbreviations in the right-hand side of the definition, then we have (for some n : N):

$$(Id \vee (E \wedge S) \vee ((E \wedge S);(E \wedge S)) \vee \ldots\ldots \vee (E \wedge S)^n \vee \ldots\ldots) \wedge \neg E'$$

which is equivalent to:

$$(Id \wedge \neg E') \vee (E \wedge S \wedge \neg E') \vee ((E \wedge S);(E \wedge S) \wedge \neg E') \vee \ldots \vee ((E \wedge S)^n \wedge \neg E') \vee \ldots$$

We want to show that for some $m,n : N \mid m \neq n$ that it is not possible for the *m*th and *n*th term above to hold true for the same start-state. This can be expressed as:

$$\nexists\, n,m : N \mid m \neq n \cdot$$
$$\forall\, x, y' \cdot (E \wedge S)^n(x,y') \wedge \neg E(y') \Rightarrow \exists\, z' \cdot (E \wedge S)^m(x,z') \wedge \neg E(z')$$

We can make things a little easier by noting that $m \neq n$ can be replaced by $m > n$ without loss of generality. Let us assume that we have some *n*, *m* such that $m > n$, and attempt to derive a contradiction from assuming that for some *x*,

$$\forall\, y' \cdot (E \wedge S)^n(x,y') \wedge \neg E(y') \qquad \text{(A)}$$

and

$$\exists\, z' \cdot (E \wedge S)^m(x,z') \wedge \neg E(z') \qquad \text{(B)}$$

Since $m > n$, (B) can be expressed as:

$$\exists\, z' \cdot (E \wedge S)^n;(E \wedge S)^{n-m}(x,z') \wedge \neg E(z')$$

which by the definition of ; is the same as:

$$\exists\, z'\ \exists\, u \cdot (E \wedge S)^n(x,u) \wedge (E \wedge S)^{n-m}(u,z') \wedge \neg E(z')$$

But $(E \wedge S)^n(x,u)$ coupled with (A) gives us $\neg E(u)$, and

$$\begin{aligned}
(E \wedge S)^{n-m}(u,z') \quad & \Rightarrow \text{pre-}(E \wedge S)^{n-m}(u) \\
& \Rightarrow \text{pre-}(E \wedge S)(u) \qquad \textit{(by the defn. of ;)} \\
& \Rightarrow \text{pre-}E(u) \wedge \text{pre-}S(u) \\
& \Rightarrow E(u) \qquad \textit{(since E is a schema over just one state)}
\end{aligned}$$

- *contradiction of* $\neg E(u)$.

This contradicts our assumption which was that two terms of the disjunction could be true for the same input state. This verifies that the definition of the *while* loop corresponds with our intuitive notion of how it is executed.

5.3 Equivalence with Hoare Rules

When we were discussing axiomatic semantics in Chapter 1 we have a series of rules for constructing proofs about programs. In order to further justify the semantics given in Chapter 4, and to relate them to the conventional framework, we will demonstrate that these rules apply to the constructs as we have defined them.

5.3.1 Two states for the price of one

We have seen that the basis of the Hoare rules is a triple such as *{P} S {Q}* where *P* and *Q* are predicates and *S* is a statement; this means that for our purposes, *P* and *Q* can be considered as a schemas over one state, and *S* as a schema over two states. After our discussion of Z schema post-conditions in terms of two-state predicates, this choice of characterisation for *Q* might seem a little strange. Strictly speaking, if a predicate is to express some property of a schema post-condition then it will need to be able to refer both to attributes of the after-state, and of its relationship with the before-state.

On the other hand if we do decide to make *Q* a two-state schema, then we run into difficulties when we are dealing with conditions (such as the rule for statement composition) which would need *Q* to be both a before- and after-state (for different schemas, of course), and thus would lead us towards the description of schema pre-conditions in terms of two-state schemas. We will avoid this issue for the moment by noting that Hoare triples will always be just that - triples - in that they will always consist of a predicate for the before- and after-state, and a schema for the statement. We can thus contend that anything that we might express using one- and two-state schemas *P* and *Q* could just as well be expressed by using a one-state schema for *Q*, and constraining it with respect to the schema for *P*.

To elaborate on this point: the variables of both *P* and *Q* may be initialised by giving specific values which satisfy them; let us call such an initialisation a *configuration.* Thus any predicate will specify a set of such configurations, and the assertion that *{P} S {Q}* is really an assertion involving a set of *pairs* of configurations. For each such configuration we can present a predicate which specifies that configuration uniquely; thus a configuration which satisfies *P* can itself be described uniquely by some predicate which is a strengthened version of *P*. Therefore, when we write *{P} S {Q}* we could just as well be writing $\{P_1\}\ S\ \{Q_1\} \wedge ... \wedge \{P_n\}\ S\ \{Q_n\}$ where $(P_1 \wedge ... \wedge P_n) \Leftrightarrow P$ and $(Q_1 \wedge ... \wedge Q_n) \Leftrightarrow Q$, except that now each Q_i is just specifying a configuration, and is thus a one-state schema.

Consequently, for the rest of this discussion we will feel justified in treating *Q* as a one-state schema specifying a configuration, and presume that we will use a number of these should we ever want to assert something about the relationship between before- and after-states. This approach, albeit a little circuitous, has the advantage of simplifying our proofs a great deal, and so we shall employ it. However, this matter will raise its head again when we come to deal with a definition for *wp* (see § 5.4.1), and we shall confront the issue a little more directly at that point.

We can now express the meaning of *{P} S {Q}* in the schema calculus as follows:

$$\forall\, x \cdot P(x) \;\Rightarrow\; (\forall\, x'\;\; S(x,x') \Rightarrow Q(x'))$$

This simply states that if a state *x* satisfies *P*, and is mapped to some state *x´* by *S*, then

x' will satisfy Q. We note that this formula is logically equivalent to:

$$\forall\ x, x' \cdot (P(x) \land\ S(x,x')) \Rightarrow Q(x')$$

5.3.2 The Assignment Axiom: *{P[t/x]} x:= t {P}*

We will consider the single assignment to one variable; the same argument can easily be extended to the general case involving a number of variables. Let us suppose that the variables brought into scope by *Sch* correspond to some state x, containing the variables $x_1 \ldots x_n$. The assignment statement will then involve one of these variables, and will look something like: $x_i := t$, where t is some expression of the appropriate type.

We can thus expand P(x) to $P(x_1,..,x_i,..,x_n)$, and $P(x)[t/x_i]$ becomes $P(x_1,..,t,..,x_n)$. What we want to show is that:

$$\forall\ x, x' \cdot (P(x)[t/x_i] \land\ S(x,x')) \Rightarrow\ P(x')$$

But $S(x,x')$ is the schema representing assignment, so that we can expand the left-hand-side of the implication to:

$$\forall\ \Delta\ Sch \mid x_1' = x_1 \land \ldots \land x_i' = t \land \ldots \land x_n' = x_n$$

It is thus obvious that direct substitution will allow us to assert that

$$\forall \Delta Sch \mid$$

$$(P(x_1,..,t,..,x_n) \wedge x_1' = x_1 \wedge ... \wedge x_i' = t \wedge ... \wedge x_n' = x_n) \Rightarrow P(x_1,..,x_i,..,x_n)$$

which is what we needed to show.

5.3.3 The Composition Rule: *{P} s_1 {R}, {R} s_2 {Q}* $\vdash$ *{P} s_1;s_2 {R}*

If we let S_1 and S_2 be the schemas representing the statements s_1 and s_2 respectively, then the first two triples of the composition rule are:

$$\forall x, x' \cdot P(x) \wedge S_1(x,x') \Rightarrow R(x') \qquad (A)$$

$$\forall y, y' \cdot R(y) \wedge S_2(y,y') \Rightarrow Q(y') \qquad (B)$$

We can replace x' in (A) and y in (B) with some state z without changing their meaning and we can then rewrite (B) using implications only, so that by the transitivity of $\Rightarrow$ we can combine (A) and (B) to get:

$$\forall x, z, y' \cdot (P(x) \wedge S_1(x,z) \wedge S_2(z,y')) \Rightarrow Q(y')$$

Since z does not occur free in $Q(y')$, this can be rewritten as

$$\forall x, y' \cdot (P(x) \wedge (\exists z \cdot S_1(x,z) \wedge S_2(z,y'))) \Rightarrow Q(y')$$

But we know that the definition of $S_1;S_2(x,y')$ is $\forall\, x\, \exists\, z \cdot S_1(x,z) \land S_2(z,y')$, so that our resulting statement is:

$$\forall\, x, y' \cdot P(x) \land S_1;S_2(x,y') \;\Rightarrow\; Q(y')$$

which, since chapter 4 defines statement composition in terms of schema composition, is equivalent to saying that $\{P\}\ s_1;s_2\ \{Q\}$, as required.

5.3.4 The *if-then-else* Rule:

$$\{P \land E\}\ s_1\ \{Q\},\ \{P \land \neg E\}\ s_2\ \{Q\} \;\vdash\; \{P\}\ \textbf{\textit{if}}\ e\ \textbf{\textit{then}}\ s_1\ \textbf{\textit{else}}\ s_2\ \textbf{\textit{fi}}\ \{R\}$$

In our statement of the above rule we assume that the schema *E* represents the Boolean expression *e*, and we now assume that S_1 and S_2 are the schemas representing s_1 and s_2 respectively. Noting that $(P\land E)(x)$ can be written as $P(x) \land E(x)$ (and similarly for $(P\land\neg E)(y)$) we can expand the first two triples to give us

$$\forall\, x, x' \cdot P(x) \land E(x) \land S_1(x,x') \;\Rightarrow\; Q(x')$$

$$\forall\, y, y' \cdot P(y) \land \neg E(y) \land S_2(y,y') \;\Rightarrow\; Q(y')$$

Renaming *y* and *y´* to *x* and *x´* in the second of these, and then combining gives us:

$$\forall\, x, x' \cdot (P(x) \land E(x) \land S_1(x,x')) \lor (P(x) \land \neg E(x) \land S_2(x,x')) \;\Rightarrow\; Q(x')$$

Since *P(x)* is common to both terms in the disjunction, this can be written as:

$$\forall\ x, x' \cdot P(x) \wedge ((E(x) \wedge S_1(x,x')) \vee (\neg E(x) \wedge S_2(x,x'))) \Rightarrow Q(x')$$

The term $(E(x) \wedge S_1(x,x')) \vee (\neg E(x) \wedge S_2(x,x'))$ obviously corresponds with our definition of the semantics for ***if e then*** s_1 ***else*** s_2 ***fi***, and is thus what we were required to prove.

5.3.5 The *while* Rule: *{P ∧ E} s {P}* ⊢ *{P} while e do s od {P ∧ ¬E}*

Assuming that *E* and *S* represent *e* and *s* respectively, we are given:

$$\forall\ x, x' \cdot P(x) \wedge E(x) \wedge S(x,x') \Rightarrow P(x') \qquad (G)$$

and we must show:

$$\forall\ y, y' \cdot P(y) \wedge (E \wedge S)^*(y,y') \wedge \neg E(y') \Rightarrow P(y') \wedge \neg E(y')$$

Given that $(E \wedge S)^0(y,y')$ is simply $y = y'$, we are in effect being asked to prove

$$\forall\ n : N, \forall\ y, y' \cdot P(y) \wedge (E \wedge S)^n(y,y') \wedge \neg E(y') \Rightarrow P(y') \wedge \neg E(y')$$

We can prove this by induction over *n*.

For the base case $n = 0$, we are required to show:

$$\forall\, y, y' \cdot P(y) \wedge (y = y') \wedge \neg E(y') \;\Rightarrow\; P(y') \wedge \neg E(y')$$

which is obviously true by simple substitution.

For the inductive step we assume:

$$\forall\, y, y' \cdot P(y) \wedge (E \wedge S)^n(y,y') \wedge \neg E(y') \;\Rightarrow\; P(y') \wedge \neg E(y')$$

which we will write as:

$$\forall\, y, y' \cdot P(y) \Rightarrow (E \wedge S)^n(y,y') \wedge \neg E(y') \;\Rightarrow\; P(y') \wedge \neg E(y') \qquad (H)$$

and we must examine the proposition that:

$$\forall\, y, y' \cdot P(y) \wedge (E \wedge S)^{n+1}(y,y') \wedge \neg E(y') \;\Rightarrow\; P(y') \wedge \neg E(y')$$

Expanding the schema composition by one term, this can be written as

$$\forall\, y, y' \cdot P(y) \wedge (E \wedge S);(E \wedge S)^n(y,y') \wedge \neg E(y') \;\Rightarrow\; P(y') \wedge \neg E(y')$$

or, by using the definition of ;, as

$$\forall\, y, y' \cdot P(y) \wedge (\exists\, z \cdot E(y) \wedge S(y,z) \wedge (E \wedge S)^n(z,y')) \wedge \neg E(y') \;\Rightarrow\; P(y') \wedge \neg E(y')$$

which is equivalent to:

$$\forall\ y, y', z \cdot P(y) \wedge E(y) \wedge S(y,z) \;\Rightarrow\; ((E \wedge S)^n(z,y') \wedge \neg E(y') \;\Rightarrow\; (P(y') \wedge \neg E(y')))$$

and this can be deduced by combining *(G)* and *(H)*, using the transitivity of ⇒ and a suitable renaming.

5.4 Schema before- and after-states

We will take a brief interlude here to note two properties of the schemas which define groups of statements.

First of all, we are moving forward through the program, so we are effectively starting from a pre-condition of *true*. At the end of the program - or of the schema which corresponds to the program - we should have a predicate which defines the relationship between all the variables which are in scope at that point. Thus in analyzing a program in this way we are attempting to characterise its strongest post-state, as opposed to picking some post-state and identifying the corresponding pre-state (this would, of course, be possible to do once we have constructed our strongest post-state.) An implication of this approach is that any schema which defines a statement effectively has a pre-condition which is true. More precisely, its pre-condition is exactly that of the state invariant. Since this state-invariant is *Sch*, which has no predicate part, the only pre-condition to a statement being executed from a particular state is that the variables in that state have the right names and types.

Secondly, we had pointed out that one of the goals of any refinement strategy is the introduction of determinism, since this will make the specification more "implementable". When dealing with programs, the implementability factor has obviously reached a maximum and so has the corresponding degree of determinism. Our language contains no non-deterministic constructs; that is, for any before-state and any statement, there can exist only one corresponding after-state. We could thus describe the schema as being "functional" over its inputs (as opposed to relational), and we can assert that for a state x and a schema S representing a group of statements, we have $\forall x \cdot \exists_1 y' \cdot S(x,y')$

5.5 Incorporating the *wp*-calculus

5.5.1 A one-state version

Once we have defined Hoare triples, we are not far from a definition of the *wp* operator. We recall that the statement $P = wp(S,Q)$ meant that the execution of S from a before-state satisfying P would terminate with an after-state satisfying Q. Based on the "one-state" approach of § 5.3.1 above, we could regard $wp(S,Q)$ as defining a one-state schema thus:

$$\forall x \cdot wp(S,Q)(x) \quad \Leftrightarrow \quad pre\text{-}S(x) \wedge (\forall x' \cdot S(x,x') \Rightarrow Q(x'))$$

The main difference between this and the definition of Hoare triples would be the insistence that *pre-S(x)* holds; ie. that S must terminate. $wp(S,Q)$ is thus a restriction of the before-states of S to those that will produce Q as an after state; in other words it consists of the pre-condition of *S* which has been strengthened in some way (or in terms of the calculus, $\forall\ x \cdot wp(S,Q)(x) \Rightarrow pre\text{-}S(x)$). The weaker *Q* is, the closer we come to specifying the full pre-condition of *S*; the weakest case is *wp(S,true)(x)* which is exactly *pre-S(x)*.

5.5.2 *wp* as a two-state schema

However, let us consider now the case where *Q* is a two-state schema, as would be normal in the specification of an after-state in Z. Since we will want to be able to compose the *wp* of two statements (in the manner of $wp(S_2,\ wp(S_1,Q))$) we will also want *wp(S,Q)* to be a two-state schema - but what exactly does it represent? In our one-state version of *wp*, we would have expected that in the situation where $P(x) = wp(S,Q)(x)$, then the establishment of *P(x)* followed by the execution of $S(x,x')$ would have led to the establishment of $Q(x')$. The two-state situation is much the same, except that we have schema composition in place of "followed by", and we will want $P;S(x,x')$ to be a sub-specification of $Q(x,x')$. This is equivalent to what [HoHe87] call the *weakest prespecification* of program *S* and specification *Q* (which they write as *S\Q*).

From now on when we speak of *wp* we will mean the two-state version, which we will define as:

$$\forall\ x,\ y' \cdot wp(S,Q)(x,y') \Leftrightarrow pre\text{-}S(x) \wedge \forall\ z' \cdot S(y',z') \Rightarrow Q(x,z')$$

The $S(y',z')$ looks a little unconventional, but it is logically correct, and is equivalent to (and simpler than) introducing something like $\forall\ u \cdot (y' = u) \wedge S(u,z')$).

5.5.3 Characteristic properties of *wp*

In order to further justify our definition, let us look again at the properties of *wp* which we introduced in § 1.4.2, and show that they hold for the *wp* defined above. First of all we will need two schemas to represent the predicates *TRUE* and *FALSE*; we define these as:

$$\forall\ x,x' \cdot \mathrm{TRUE}(x,x')$$

$$\forall\ x,x' \cdot \neg\mathrm{FALSE}(x,x')$$

In other words, any pair of states will satisfy *TRUE(x,x′)*, and no pair of states will ever satisfy *FALSE(x,x′)*.

We can now deal with each of the properties in turn:

5.5.3.1 Law of the Excluded Miracle

By our definition of *wp*, we can expand *wp(S,FALSE)(x,y′)* to:

$$\forall\ x,\ y' \cdot \text{pre-S}(x)\ \wedge\ (\forall\ z' \cdot S(y',z') \Rightarrow \mathrm{FALSE}(x,z'))$$

We have asserted that $\nexists\, x,z' \cdot FALSE(x,z')$ and so this conjunction is obviously false, as required.

5.5.3.2 Monotonicity of *wp* with respect to implication

Suppose that we have two schemas Q and R such that $\forall\ x,x' \cdot Q(x,x') \Rightarrow R(x,x')$; to prove this property, we are required to show that:

$$\forall\ y, y' \cdot wp(S,Q)(y,y') \Rightarrow wp(S,R)(y,y')$$

However, the term *pre-S(y)* will be common to both these expansions, and so we will just have to show that

$$\forall\ y, y' \cdot (\forall\ u' \cdot S(y',u') \Rightarrow Q(y',u')) \ \Rightarrow\ (\forall\ v' \cdot S(y',v') \Rightarrow R(y',v'))$$

which is a straightforward consequence of our initial assumption and the transitivity of implication.

5.5.3.3 Distributivity of conjunction and disjunction

First of all let us note two rules of logical equivalence[1] that we will use; for arbitrary predicates A and B, and some variable(s) t of type T,

$$(\forall\ t : T \cdot A) \wedge (\forall\ t : T \cdot B) \ \dashv\vdash\ (\forall\ t: T \cdot A \wedge B) \qquad (R1)$$

$$(\forall\ t : T \cdot A) \vee (\forall\ t : T \cdot B) \ \vdash\ (\forall\ t: T \cdot A \vee B) \qquad (R2)$$

Note that *(R2)* only allows deduction from left to right, while *(R1)* asserts full equivalence between the two sequents.

[1]A list of such rules is given in [Dill90], § 20.3

To show that conjunction is distributive, we must prove that:

$$\forall\ x, x' \cdot wp(S,Q)(x,x') \wedge wp(S,R)(x,x') \Leftrightarrow wp(S,Q{\wedge}R)(x,x')$$

Since x and x' are quantified over the whole expression we will not keep mentioning them, and we can expand the left-hand-side of the implication to:

$$pre\text{-}S(x) \wedge (\forall\ z' \cdot S(x',z') \Rightarrow Q(x',z')) \wedge (\forall\ y' \cdot S(x',y') \Rightarrow R(x',y'))$$

We can rename y' to z', and use *(R1)* to express this as:

$$pre\text{-}S(x) \wedge (\forall\ z' \cdot (S(x',z') \Rightarrow Q(x',z')) \wedge (S(x',z') \Rightarrow R(x',z')))$$

which (since $S(x',z')$ is common) is equivalent to:

$$pre\text{-}S(x) \wedge (\forall\ z' \cdot S(x',z') \Rightarrow (Q(x',z') \wedge R(x',z')))$$

and this is simply $wp(S,Q{\wedge}R)(x,x')$, as required.

The proof for the distributivity of disjunction is the same, except that now we will be using *(R2)* which, since it only allows deduction in one direction, only permits us to establish implication, but not full equivalence.

5.5.3.4 Distributing the disjunction when the schema is deterministic

We have established the distributive properties in the situation where $S(x,x')$ is any schema, and thus may be non-deterministic. Let us now consider the definition of *wp(S,Q∧R)(x,x′)* in the situation where *S(x,x′)* is definitely deterministic. As usual we can expand the definition of the *wp* and make the assumption that we know:

$$\text{pre-}S(x) \wedge (\forall z' \cdot S(x',z') \Rightarrow (Q(x',z') \wedge R(x',z')))$$

But assuming that *pre-S(x)* holds is equivalent to assuming that $\exists\, y' \cdot S(x,y')$ holds, but since *S(x,x′)* is deterministic this *y′* is unique, and we could write $\exists_1\, y' \cdot S(x,y')$, and for this unique *y′* we have $Q(x',y') \wedge R(x',y')$. Thus we can write:

$$\exists_1\, y' \cdot S(x,y') \Rightarrow (Q(x',y') \wedge R(x',y'))$$

Which we can immediately rephrase as:

$$\exists_1\, y' \cdot (S(x,y') \Rightarrow (Q(x',y')) \wedge (S(x,y') \Rightarrow R(x',y'))$$

But we are now working under an existential quantifier, and can use the rule:

$$(\exists\, t : T \cdot A) \vee (\exists\, t : T \cdot B) \quad \dashv\vdash \quad (\exists\, t : T \cdot A \vee B) \qquad \textit{(R3)}$$

and thus deduce that:

$$(\exists_1\, y' \cdot S(x,y') \Rightarrow (Q(x',y')) \wedge (\exists_1\, z' \cdot S(x,z') \wedge R(x',z'))$$

And by the definition of unique existence, we can convert this back to

$$\exists_1\, u \cdot S(x,u) \Rightarrow (\forall\, y' \cdot S(x,y') \Rightarrow (Q(x',y')) \wedge (\forall\, z' \cdot S(x,z') \wedge R(x',z'))$$

from which we can subsequently assert $wp(S,Q)(x,x') \wedge wp(S,R)(x,x')$, and this establishes equivalence, as required.

5.5.4 Other features of *wp*

This definition of *wp* means that if $P(x,y') = wp(S,Q)(x,y')$ then every valid before-state of *P* is a valid before state of *Q*, and is mapped by *P* to a state which is mapped by *S* to the corresponding after-state of *Q*. We could thus write:

$$\forall\ x,x' \cdot wp(S,Q);S(x,x') \Rightarrow Q(x,x')$$

We can use the weakest prespecification in situations where we wish to split up a specification into the composition of two simpler specifications.

The less "specific" *S* is, the closer we are to describing the specification *Q*, and, in the situation where *S* is just the identity over *x*, we have

$$\forall\ x,x' \cdot wp(Id,Q)(x,x') \Rightarrow Q(x,x')$$

where *Id* is such that: $\forall\ x, x' \cdot Id(x,x') \Leftrightarrow x = x'$.

Another trivial case is where we have $\forall\ x, x' \cdot wp(S,Q)(x,x') = TRUE(x,x')$ - this corresponds to the situation where *S* is a total operation, and for every before-state we can expect *S* to produce one of the corresponding after-states that *Q* would have produced.

Since *wp(S,Q)* is the weakest prespecification, we have that for any other specification *T* which is stronger,

$$\forall\ x, x' \cdot (T(x,x') \Rightarrow wp(S,Q)(x,x')) \quad \Leftrightarrow \quad (T;S(x,x') \Rightarrow Q(x,x'))$$

In other words, if *T* implies the weakest prespecification of *S* and *Q*, then it is strong enough to be a prespecification of *S* and *Q*, and so can be composed with *S* to give a sub-specification of *Q*.

5.5.5 *wp* and Operation Refinement

Recall that in § 3.4.4 we defined the proof obligation for operational refinement as:

$$\forall\ s,s' \cdot \text{pre-AbsOp}(s) \ \Rightarrow\ (\text{pre-ConOp}(s) \wedge (\text{ConOp}(s,s') \Rightarrow \text{AbsOp}(s,s')))$$

We can see now that the right-hand side of this definition has the same form as the definition of *wp*, and we could thus replace it to get the condition:

$$\forall\ s \cdot \text{pre-AbsOp}(s) \ \Rightarrow\ \forall\ s' \cdot \text{wp(ConOp,AbsOp)}(s,s')$$

From our discussion above, this assertion states that over every valid before-state of *AbsOp* we can expect *ConOp* to behave as *AbsOp* might. *ConOp* is more deterministic than *AbsOp* since it is defined over a larger set of before-states (ie. *ConOp* will terminate from a greater number of states).

5.5.6 *wp* and Data Refinement

We can combine the correctness and applicability conditions for data refinement to give the proof obligation:

$$\forall\, a, c, c' \cdot (\text{pre-AbsOp}(a) \wedge \text{Rel}(a,c)) \Rightarrow$$
$$\text{pre-ConOp}(c) \wedge (\text{ConOp}(c,c') \Rightarrow \exists\, a' \cdot \text{Rel}(a',c') \wedge \text{AbsOp}(a,a'))$$

Finding an application for *wp* in this formula presents some problems with making sure that we are dealing with schemas over the same state-space. Based on the structure of the above formula, it seems that we would like to apply the *wp* to the terms *ConOp(c,c′)* and $\exists$ *a′ ·Rel(a′,c′)* $\wedge$ *AbsOp(a,a′)*, except that we can't do so directly, since the latter is not defined solely over the states *c* and *c′*.

First of all we note that $\exists$ *a′ ·Rel(a′,c′)* $\wedge$ *AbsOp(a,a′)* is similar to the form of a definition for the composition of two schemas, and thus we can write it as *Rel;AbsOp(a,c′)*. Now since this term is actually the right-hand-side of an implication, we are only interested in its value under certain conditions, namely those conditions specified by the left-hand-side of the implication. Accordingly, in our case we are given that *Rel;AbsOp(a,c′)* must hold under the supposition that *pre-AbsOp(a)* $\wedge$ *Rel(a,c)* holds. Immediately we can assert that if we have *Rel(c,a)* $\wedge$ *Rel;AbsOp(a,c′)* then we can deduce *Rel;AbsOp;Rel(c,c′)*, which gives us a term in *c* and *c′*, as required.

We can now rewrite the proof obligation as:

$$\forall\, a, c, c' \cdot (\text{pre-AbsOp}(a) \wedge \text{Rel}(a,c)) \Rightarrow$$
$$\text{pre-ConOp}(c) \wedge (\text{ConOp}(c,c') \Rightarrow \text{Rel;AbsOp;Rel}(c,c'))$$

This is much more amenable to representation using *wp*, and we can write:

$$\forall\ a, c \cdot (\text{pre-AbsOp}(a) \wedge \text{Rel}(a,c)) \Rightarrow \forall\ c' \cdot \text{wp}(\text{ConOp}, \text{Rel;AbsOp;Rel})(c,c')$$

The proof obligation is thus to show that *ConOp* and *Rel;AbsOp;Rel* will produce the same after-state in situations where the before-state can be mapped by *Rel* to a valid before-state of *AbsOp*.

5.6 Annotations and Guarded Commands

When attempting to understand - or describe - the action of a program, it is quite common to write down a predicate (as a comment) at a particular point, with the implication being that whenever control reaches that point, then the particular predicate should hold for the current value of the variables. In certain situations during the development of a program we may include a predicate to indicate that our deduction is based on the understanding that this predicate will hold true at the point indicated, even if this is not necessarily implied by the preceding program statements. In the terminology of [MoVi90] we will describe both these predicates as *annotations*, and call the former *assertions* and the latter *coercions*.

5.6.1 Assertions

Assertions do not add anything to our program; they merely serve to highlight some particular point that is of interest. In [Ders83] these assertions are used to document a program, and rules are given which allow more assertions to be deduced from an existing program and set of assertions. Some of these assertions may be written at a particular point in the program based solely on the statement before or after it. Others may be moved "backwards" and "forwards" through the program text, with their final form depending on the intervening statements.

These type of assertions are implicit in our semantics; for example, the guards in the alternative and iterative commands are automatically introduced as pre-conditions to the ensuing block of statements. We shall not examine this equivalence in any depth, since we would just be reiterating much of the proof of the equivalence of the Hoare triples to our semantics. It will suffice to note that if we wish to introduce an assertion *A* between two blocks of statements *S* and *T*, (which might be written as *S; {A} T*) then we would have to show that:

$$\forall\, x \cdot \text{post-S}(x) \wedge \text{pre-T}(x) \Rightarrow A(x)$$

5.6.2 Coercions

This type of annotation is basically a specification which has yet to be expressed in programming terms. We may use it because we wish to reserve the realisation of some part of the specification until later, or because we do not wish to be concerned with the actual implementation details. For instance, after a statement like *read(v)* we may introduce a coercion specified by the predicate *P(v)* which would indicate the range of acceptable values for the variable; the method used to confine *v* to these values may not interest us at this stage. (We would write such a coercion as: *read(v) [P(v)]*).

All of the commands that we have described in the previous chapter are total, in that whenever we have introduced pre-conditions to a number of statements, we have made sure that the conjunction of all these pre-conditions is *true*. However, if we use coercions with the langauge then we will be insisting that some condition holds at a particular point, and not considering at all the case where it does not hold. What this means is that the rest of the statements in the block are basically the body of a guarded command, to which the coercion is the guard.

When we have a coercion before a sequence of statements, such as $[P(x,x')]\ S(x,x')$ we are effectively specifying the operation $(P \wedge S)(x,x')$. Thus a coercion is just a method of constraining the valid before-states of an operation. The alternative statement *if B then S_1 else S_2 fi* could just as well be regarded as the disjunction of two coerced statements: $([B(x)]\ \ S_1(x,x')) \ \vee\ ([\neg B(x)]\ \ S_2(x,x'))$. The iterative statement *while B do S od* would then be the reflexive-transitive closure of $([B(x)]\ \ S(x,x'))$ composed with $([\neg B(x)]\ Id(x,x'))$.

A program which consists simply of a coercion $[Q(x,x')]$ and nothing else is what we would normally describe as a specification. During the process of refinement we would hope to introduce statements in place of this coercion. This will be a step-by-step process, so each type we would want to replace $[Q(x,x')]$ with $[P(x,x')]\ S$, where the statement $S(x,x')$ will be guaranteed to establish $Q(x,x')$ when started in a before-state satisfying $P(x,x')$. This should look rather familiar, since the specification of $P(x,x')$ would be $wp(S,Q)(x,x')$.

Consider again the maximally weak specification $TRUE(x,x')$ which is basically an indication that we do not constrain the before- or after-states in any way; *TRUE* maps anything to anything. This will allow us to write $[P(x,x')]\ \ S(x,x')$ as the total command; namely *if P then S else TRUE fi.*

5.7 Meaning Functions

We met the idea of a meaning function when we were discussing denotational semantics in chapter 1. The basic idea was to concentrate on some particular aspect of the semantics as being enough to describe the operation of the program.

Our semantics essentially regard a program as specifying an operation over those sets of variables which are currently in scope. A fully-expanded semantics for some program, where each statement was represented by its corresponding schema, would amount to a detailed description of the program as a sequence of states; this corresponds to the operational view, or the meaning function M_{ALL}. If we were to bring our knowledge of mathematical logic, and of the rules governing members of the sets $\mathbb{N}$ and $\mathbb{Z}$ (and whatever other data sets we were using), to bear on this, we could hope to simplify the whole program down into just one schema which expressed the relationship between the start and finish values of the variables directly, without giving any details of their intermediate values. This would correspond to the more abstract view of M_{END}.

We could also regard the program purely in terms of a mapping from its input to output. This would correspond to the meaning function M_{IO} and could be achieved by eliminating the schema *IO* and changing the definition of ϕ to:

ϕ : program $\rightarrow$ SPEC

$\forall$ P : ident, blks : block*, dls : decl*, sts : stat* $\cdot$

ϕ [[Program P dls blks sts]] =

[*GIVEN*]

Boolean == {*TRUE*, *FALSE*}

[outseq : seq $\mathbb{Z}$ | outseq = < >]

[P] $\hat{=}$ λ inseq : seq $\mathbb{Z}$ $\cdot$

σ([sts], β([blks], δ([dls], GlobEnv))).outseq´

CHAPTER 6 - SAMPLE PROGRAMS

Up to now we have been concerned with bringing together disparate elements involving the link between programs and specifications, using the proof calculus of chapter 2 and the semantic framework of chapter 4. In this chapter we relax the emphasis on formality a little, and begin to deal with programs on a first-name basis. Later on we will be taking some sample programs and attempting to fit them into our model, but first of all let us consider the nature of the problem that we will be confronting.

We will not be attempting to provide some general set of rules and heuristics which might serve as some sort of "handbook" of the annotation and analysis process. Much work has already been done on deriving programs from specifications; many texts give examples and useful rules which can help with this task (see [Drom89] in particular for an extensive selection). In the next section we argue that these rules and guidelines will also be useful also for what we are trying to do. In previous chapters we have cast programs and specifications into the same mould, regarding them as being different in emphasis rather then in fundamental substance. We develop this theme to claim that the link between programs and specifications should not be considered purely in directional terms (*from* specifications *to* programs), but as a property of the structures that encompass both of them.

6.1 Reversing Algorithm Development?

Leaving aside the business of formalising representations for abstract objects, and introducing refinements etc. for the moment, let us concentrate on the task of finding an *algorithm* which will implement the specification.

Assume for the moment that we have been able to map each specification state into some "equivalent" program state; we'll refer to these as "abstract" and "concrete" states respectively. Then for each operation we would hope to be able to present a state to the program as input, and if its equivalent abstract state satisfies the pre-condition of the operation, then we would expect our program to produce a state whose equivalent abstract version satisfies the post-condition. Assuming that it is decidable whether or not some state satisfies the pre- or post-condition, a rather naive program might simply opt to search the set of valid states until it has found a suitable one. Efficiency considerations seldom alow us to tolerate such innocence. Developing a realistic algorithm to solve a problem involves exploiting properties of the state-set, and of the pre- and post-conditions, in order to narrow the range of our search.

Much of this "narrowing" is second nature to a programmer. For example, suppose we are asked to sort an array of integers; our answer will lie somewhere in the set of arrays of integers. It is trivial to eliminate infinitely many arrays from our search-space by noting that the result must be a permutation of the initial array, and that it must be sorted. Few programmers would feel a great deal of pride in asserting that their program also made use of the transitivity of the $\leq$ operator over integers. However, increasing degrees of cunning (or "intuition") are involved as we progress towards more and more efficient searching strategies: we would agree that the first person to discover the Quicksort had something worth taking about. Were we concerned here with automating the programming process, we might be discussing the role of experience, skill, inspiration, etc... As we are not, let us be content just to note their influence.

The above discussion is hardly profound, but it is important to us as we consider the relative difficulty of re-constructing a specification from a program. If writing a program involves "adding in" details to the specification (the task of deciding which details being presumably what programmers are paid for), can we expect our task to be simply "removing" these details until we have attained an acceptable degree of abstractness? Deciding what to remove certainly sounds easier than thinking up things to add in. After all, the fruits of our experience/skill/intuition have been written into the program, and are thus available to us.

Things are not quite that easy. Let us consider an example: suppose we are asked to write a program to find the greatest common divisor of two natural numbers. Our specification might look something like this:

$$\begin{array}{|l}
\text{divisors} : \mathbb{N}_1 \rightarrow \mathbb{P}_1\, \mathbb{N}_1 \\
\hline
\text{divisors} = \lambda\, n : \mathbb{N}_1 \cdot \{d : \mathbb{N}_1 \mid \exists\, c : \mathbb{N}_1 \cdot d * c = n\}
\end{array}$$

$$\begin{array}{|l}
\text{gcd} : \mathbb{N}_1 \times \mathbb{N}_1 \rightarrow \mathbb{N}_1 \\
\hline
\text{gcd} = \lambda\, x,y : \mathbb{N}_1 \cdot \max(\text{divisors}(x) \cap \text{divisors}(y))
\end{array}$$

We could fairly routinely deduce that the answer lies somewhere between 0 and the smaller of the two numbers; we may begin working on the properties of divisibility to see if this can help us. But suppose a mathematician passed by at this point and, when hearing of our problem, pointed out that the g.c.d. of two numbers had a interesting property: that it was equal to the g.c.d. of the smaller number, and the larger less the smaller. We check that this is so, and could rewrite our specification accordingly:

$$\begin{array}{|l}
gcd : \mathbb{N}_1 \times \mathbb{N}_1 \rightarrow \mathbb{N}_1 \\
\hline
\forall x,y : \mathbb{N}_1 \cdot \\
\quad (x > y \Rightarrow gcd(x,y) = gcd(x-y,y)) \land \\
\quad (x < y \Rightarrow gcd(x,y) = gcd(x,y-x)) \land \\
\quad (x = y \Rightarrow gcd(x,y) = x)
\end{array}$$

We can then proceed fairly routinely to an implementation looking something like:

```
[Program GCD
x : Nat
y : Nat
read (x)
read (y)
while (x ≠ y) do
  if (x > y) then x := x - y else y := y - x  fi
od
write (x)]
```

Obviously the connection between the program and the second specification above is relatively transparent; we would expect to be able to extract this type of recursive equation from any loop body. However if we had hoped to be able to get back to something like the original specification, then we would effectively be searching for something which has the *property* given above. Yet there is nothing "inferior" about the second specification; indeed it would probably form the basis for an algebraic (or property-oriented) specification of the program. After all, if it did not fully specify the g.c.d. function, we could not have used it so conveniently.

The nub of our argument is this: both of the above specifications detail properties of the g.c.d. which are sufficient to describe the function over its input domain. We might regard the first as being more useful to us, but a completely objective view would not differentiate between them. In other words, starting with either specification, we could expect to come across the other one in our search for an implementation, depending on our search method. Thus there exists a *symmetry* here; we have moved to a new expression of our problem (which is hopefully better oriented towards our target domain), but moving back again will not necessarily be any easier.

Let us imagine a state-space consisting of "properties", and sets of properties, and so on, where any set of properties will specify some kind of a relation (not always being consistent/sufficient/sensible). We can regard a particular function as imposing an equivalence relation on this state-space, consisting of all those sets of properties which describe it fully. Our search for an algorithm involves moving around in one of these equivalence classes; some moves may be easier than others, but we cannot guarantee a strict hierarchy.

The bottom line here is that we do not intend to consider deeply the problem of finding out what algorithms do; the heuristics involved in this process are likely to be as widely varied as those used for developing programs. However, the introduction of formal methods into program development was not designed just to provide inspiration to people to create new algorithms; its purpose it to provide a framework within which this development can take place. In this thesis we have been concerned with providing a rigorous basis for moving between programs and specifications; to illustrate this basis, and to provide a *sample* of the techniques we can expect to employ, we will use the rest of this chapter to analyze a small selection of sample programs.

6.2 Some notational conveniences

Before we deal with the specific examples, let us relax our program notation a little bit. In order to eliminate unnecessary details, we will make some alterations to the way in which we can write programs - these should be considered as notational abbreviations rather that extensions to the language. We could have introduced some of them into our language in chapter 4, but they do not add to the power of the language in any way, and thus would have complicated our semantic definition unnecessarily.

1. We may write declarations as $[\![i_1, \ldots i_n : t]\!]$, which is shorthand for $[\![i_1 : t \ldots i_n : t]\!]$

2. Rather that use specific integer constants for certain values (such as array bounds) we will allow identifiers (such as N) to represent the required values. Again, this should not be seen as an extension to the language, but more as a first step towards the generalisation of the program. We will introduce any such constants by declaring them in a coercion before the start of the program.

3. We will write $[\![\textit{if } B \textit{ then } S \textit{ fi}]\!]$ to mean $[\![\textit{if } B \textit{ then } S \textit{ else skip fi}]\!]$.

4. As a further simplification, we will abbreviate $[\![\textit{if } B_1 \textit{ then } S_1 \textit{ else if } B_2 \textit{ then } S_2 \textit{ fi fi}]\!]$ to the more common $[\![\textit{if } B_1 \textit{ then } S_1 \textit{ elseif } B_2 \textit{ then } S_2 \textit{ fi}]\!]$. When we are translating this to a schema we get something of the form $[B_1 \wedge S_1] \vee [B_2 \wedge S_2] \vee [\neg B_1 \wedge \neg B_2 \wedge Id]$,

where *Id* is the identity over the current environment.

5. We will ignore the function *AT* in example not involving pointers.

6.3 The Dutch National Flag [Dijk76]

We will use this example to illustrate the process of giving a program its appropriate semantics in terms of chapter 4; we will not go into as much detail in later examples.

```
[RED, WHITE, BLUE : ℤ]
[N : ℕ | N > 1]

[Program DNF
r, w, b : Int
A : array 1 to N+1 of Int

[Proc READARR (var A : array 1 to N+1 of Int)
i : Nat
x : Int
i := 1
while i ≤ N do
  read(x)
  A[i],i := x,i+1
od]
call READARR (A)
r,w,b := 0,0,N+1
while (w ≠ b-1) do
  if A[w+1] = WHITE then
    w := w + 1
  elseif A[w+1] = BLUE then
    A[w+1],A{b-1] := A[b-1],A[w+1]
    b := b-1
  elseif A[w+1] = RED
    A[w+1],A[r+1] := A[r+1],A[w+1]
    w,r := w+1,r+1
  fi
od]
```

Applying ϕ to the above program means that we must first apply δ to the program declarations, which produces the main program environment:

$$
\begin{array}{|l}
\text{MEnv} : \text{ENV} \\
\hline
\text{MEnv} == \{[\![\text{in}]\!] \mapsto \text{inseq} : \text{seq}\ \mathbb{Z},\ [\![\text{out}]\!] \mapsto \text{outseq} : \text{seq}\ \mathbb{Z}, \\
\qquad [\![\text{r}]\!] \mapsto \text{r} : \mathbb{Z}^{\perp},\ [\![\text{w}]\!] \mapsto \text{w} : \mathbb{Z}^{\perp},\ [\![\text{b}]\!] \mapsto \text{b} : \mathbb{Z}^{\perp},\ [\![\text{A}]\!] \mapsto \ \text{AA} : \{1 \mathinner{..} N\} \rightarrow \mathbb{Z}^{\perp}\}
\end{array}
$$

6.3.1 The procedure READARR

We will take the opportunity to go through *READARR* in some detail at this point. We apply β to *READARR*, using the rule:

$$
\begin{array}{|l}
\beta\ [\![\text{Proc N (pms) dls blks}_{\text{N}}\ \text{sts}]\!]\ \text{E} \ = \ \text{E}\ \oplus\ \{[\![\text{N}]\!] \mapsto \text{PE}\} \\
\text{where}\ \ \text{PE} : \text{ProcEnv}\ \mid \\
\qquad\qquad \text{PE.formals} = \pi\ [\![\text{pms}]\!]\ \text{E}_2 \\
\qquad\qquad \text{PE.body} = (\Omega\ [\![\text{dls}]\!]\ \text{E}_2)\ \wedge\ (\sigma\ [\![\text{sts}]\!]\ (\beta\ [\![\text{blks}]\!]\ \text{E}_2)) \\
\text{and}\ \ \text{E}_2 : \text{ENV}\ \mid \text{E}_2 = \delta\ [\![\text{dls}]\!]\ (\delta\ [\![\text{pms}]\!]\ \text{E})
\end{array}
$$

Working out E_2 will involve adding the parameter and the two variable declarations to the environment, to get:

$$
\begin{array}{|l}
\text{SEnv} : \text{ENV} \\
\hline
\text{SEnv} = \text{MEnv} \oplus \{[\![\text{A}]\!] \mapsto \text{BA} : \{1 \mathinner{..} N\} \rightarrow \mathbb{Z}^{\perp},\ [\![\text{i}]\!] \mapsto \text{i} : \mathbb{N}^{\perp},\ [\![\text{x}]\!] \mapsto \text{x} : \mathbb{Z}^{\perp}\}
\end{array}
$$

Note that there is a new mapping for the program identifier *[A]* which will overwrite the previous mapping.

In the function σ we will be using *MakeSch(SEnv* $\triangleright$ *DECL)* to bring the relevant Z identifiers into scope; we will define the result of this application as the schema:

```
┌── SDec ──────────────────────
│ inseq, outseq : seq ℤ⊥
│ r, w, b : ℤ⊥
│ i : ℕ⊥
│ x : ℤ⊥
│ BA : {1 .. N} → ℤ⊥
└──────────────────────────────
```

Since *READARR* has no sub-blocks, we can get right down to applying the mapping for statements. The first statement is just a simple assignment, which leaves us with:

```
┌── PStat1 ────────────────
│ Δ SDec
├──────────────
│  (BA = { }) ∧ (x = ⊥) ∧ (i = ⊥)
│ (inseq′ = inseq) ∧ (outseq′ = outseq) ∧ (r′ = r) ∧ (w′ = w) ∧ (b′ = b)
│ (BA′ = BA) ∧ (x′ = x) ∧ (i′ = 1)
└──────────────────────────
```

The loop body can be straightforwardly interpreted as:

PLBody

Δ SDec

$inseq' = tail(inseq)$

$(outseq' = outseq) \wedge (r' = r) \wedge (w' = w) \wedge (b' = b)$

$x' = head(inseq)$

$(i \leq N) \wedge (i' = i+1)$

$(BA' = BA \otimes \{i \mapsto x'\})$

Accordingly, the schema which represents the procedure will consist of

$$PStat1 \ ; \ PLBody^{*} \wedge [SDec' \mid i' > N]$$

The loop body is fairly straightforward, so that we may formulate a loop invariant based on *i*. If we let *Ins* be the initial value of *inseq* before the loop, and note that since the index set of the function *BA* is *{1 .. N}* we may treat it as though it were a sequence, so that we get the invariant:

```
┌── PInv ─────────────────────
│ SDec
│ Ins : seq ℤ
├──────────────
│ 1 ≤ i
│ x = Ins(i-1)
│ (BA for (i-1)) = (Ins for (i-1))
│ inseq = (Ins after (i-1))
└─────────────────────────────
```

In order to verify the invariant, we must show that it is established by the initialisation:

$$[\text{SDec};\ \text{Ins} : \text{seq}\ \mathbb{Z} \mid \text{inseq} = \text{Ins}] \wedge \text{PStat1} \;\Rightarrow\; \text{PInv}'$$

which is trivially true (allowing for the fact that *x* has no value). We must then show that each iteration of the loop preserves the invariant, or:

$$(\text{PInv} \wedge \text{PLBody} \wedge (\text{Ins}' = \text{Ins})) \;\Rightarrow\; \text{PInv}'$$

which, since *inseq = Ins after (i-1)*, we can verify by noting that:

$$\text{inseq}' = \text{tail(inseq)} = \text{tail(Ins after (i-1))} = \text{Ins after i}$$

$$\text{x}' = \text{head(inseq)} = \text{head(Ins after (i-1))} = \text{Ins(i)}$$

and

$$\text{BA}' = \text{BA} \oplus \{\text{i} \mapsto \text{x}'\} = (\text{BA for (i-1)}) \oplus \{\text{i} \mapsto \text{x}'\} \oplus (\text{BA after i})$$

$$= (\text{Ins for (i-1)}) \oplus \{\text{i} \mapsto \text{Ins(i)}\} \oplus (\text{BA after i})$$

$$= (\text{Ins for i}) \oplus (\text{BA after i})$$

Upon termination of the loop we have $i = N+1$, and combining this with *PInv* allows us to assert that the overall effect of the procedure is:

```
┌── ReadArr ──────────────────
│ Δ SDec
├─────────────────
│ (outseq′ = outseq) ∧ (r′ = r) ∧ (w′ = w) ∧ (b′ = b)
│ i′ = N+1
│ x′ = inseq(N)
│ BA′ = inseq for N
│ inseq′ = inseq after N
└─────────────────────────────
```

The semantic function π when applied to the parameters of *READARR* will have produced a mapping consisting of $\{VAR \mapsto (1 \mapsto BA)\}$, and so the function β will have added $[\![READARR]\!] \mapsto RPE$ to *MEnv*, where we have:

$$\text{RPE.formals} = \{\text{VAR} \mapsto (1 \mapsto \text{BA})\}$$

$$\text{RPE.body} = \text{ReadArr}$$

Thus $\text{MEnv}([\![\text{READARR}]\!]).\text{body} = \text{ReadArr}$

6.3.2 Back in the main program...

We will need a corresponding version of *SDec*, called *MDec* which will be used to bring the Z variables From the main part of the program into scope - this will be the result of *MakeSch(MEnv* $\rhd$ *DECL)*, which is:

$$\begin{array}{|l}
\multicolumn{1}{l}{\text{MDec}} \\
\mathit{inseq}, \mathit{outseq} : \mathrm{seq}\ \mathbb{Z} \\
r, w, b : \mathbb{Z}^{\perp} \\
AA : \{1 \mathrel{..} N\} \rightarrow \mathbb{Z}^{\perp} \\
\hline
\end{array}$$

Note that the purpose of range-restricting *MEnv* to the set *DECL* is to exclude the mapping for the procedure ⟦*READARR*⟧ (which is of type *ProcEnv*), since this has no place among our declaration of Z variables.

So, to give a meaning for the statement in the main program ⟦*call READARR (A)*⟧, we must take the schema *ReadArr*, replace all occurrences of the formal parameter *BA* with the corresponding argument *AA*, and then use schema projection to hide all the "local" variables of the procedure; this is expressed as *ReadArr [AA/BA]* $\upharpoonright$ Δ *MDec*, which, when we ignore the hidden variables, gives us:

$$\begin{array}{|l}
\multicolumn{1}{l}{\text{MStat1}} \\
\Delta\ \mathit{MDec} \\
\hline
(AA = \{\}) \wedge (r = \perp) \wedge (w = \perp) \wedge (b = \perp) \\
(\mathit{outseq}' = \mathit{outseq}) \wedge (r' = r) \wedge (w' = w) \wedge (b' = b) \\
AA' = \mathit{inseq} \text{ for } N \\
\mathit{inseq}' = \mathit{inseq} \text{ after } N \\
\hline
\end{array}$$

So far we have just given a semantics for the first command in the program! However, we dealt with this command in such detail in order to illustrate the mechanism for handling procedure calls: we will not be dealing with the rest of the commands at such length.

Before we work out a definition for the main program loop, let us illustrate the assignment command, by looking at the assignment which occurs in the last branch of the conditional.

6.3.3 The concurrent assignment

The assignment we wish to specify can be written as:

$$A[w+1],A[r+1],w,r := A[r+1],A[w+1],w+1,r+1$$

(We used two assignments in the program in order to make this look at little nicer; we use the above version just for the sake of the example.)

We recall the rule for the assignment statement:

$$\begin{array}{|l} \sigma\ [\![v_1,..,v_n := e_1,..,e_n]\!],\ \mathrm{E}) \ = \\ \quad \Xi\ Sch \setminus ((\xi\ [\![v_1]\!]\ \mathrm{E})',\ \ldots\ (\xi\ [\![v_n]\!]\mathrm{E})') \ \wedge\ [\Delta\ Sch \mid\ \hat{}/\ \gamma{<}\alpha(v_1,e_1)\mathrm{E},\ldots,\alpha(v_n,e_n)\mathrm{E}{>}] \end{array}$$

Working out the sequence that results from applying α to each individual assignment is fairly straightforward, since it only involves variable and array references. Since the array is one-dimensional, we can use $\oplus$ instead of $\otimes$, and we get:

$$< AA' = AA \oplus \{w+1 \mapsto AA(r+1)\},\ \ AA' = AA \oplus \{r+1 \mapsto AA(w+1)\},$$
$$w' = w+1,\ \ r' = r+1 >$$

Next we have to apply the grouping function γ to this sequence. According to the definition of this function we can see that the condition:

$$\exists\, i : \mathbb{N} \cdot (i \neq 1) \wedge (S(i) = (l' = l \otimes \{t_2\}))$$

holds, with $i = 2$, $l = AA$ and $t_2 = r+1 \mapsto AA(w+1)$. We thus apply the consequence:

$$\gamma(S) = \gamma(\text{tail } (S \oplus \{i \mapsto (l' = l \otimes \{t_1,t_2\})\}))$$

to get the new sequence (with three elements) to which we must apply γ:

$$< AA' = AA \oplus \{w+1 \mapsto AA(r+1),\ r+1 \mapsto AA(w+1)\},\ \ w' = w+1,\ \ r' = r+1 >$$

Further application of γ leaves the sequence unchanged, and so our last step is just to distribute the conjunction through this to get:

$$AA' = AA \oplus \{w+1 \mapsto AA(r+1),\ r+1 \mapsto AA(w+1)\}\ \wedge\ w' = w+1\ \wedge\ r' = r+1$$

This then gives us the schema for this statement:

MAss3

Δ MDec

$(inseq' = inseq) \wedge (outseq' = outseq)$

$(b' = b)$

$(AA' = AA \oplus \{w+1 \mapsto AA(r+1), r+1 \mapsto AA(w+1)\}) \wedge (w' = w+1) \wedge (r' = r+1)$

6.3.4 The main loop body

In a similar manner we can characterise the other assignments and, if we add in the preceding guard from the conditional (as we would do when writing the definition for that assignment) we get the four schemas:

MBr1

Δ MDec

$AA(w+1) = \mathit{WHITE}$

$(inseq' = inseq) \wedge (outseq' = outseq)$

$(b' = b) \wedge (AA' = AA) \wedge (r' = r)$

$(w' = w+1)$

MBr2

Δ MDec

(AA(w+1) $\neq$ *WHITE*) $\wedge$ (AA(w+1) = *BLUE*)

(inseq′ = inseq) $\wedge$ (outseq′ = outseq)

(r′ = r) $\wedge$ (w′ = w)

(AA′ = AA $\oplus$ {w+1 $\mapsto$ AA(b-1), b-1 $\mapsto$ AA(w+1)}) $\wedge$ (b′ = b-1)

MBr3

Δ MDec

(AA(w+1) $\neq$ *WHITE*) $\wedge$ (AA(w+1) $\neq$ *BLUE*) $\wedge$ (AA(w+1) = *RED*)

(inseq′ = inseq) $\wedge$ (outseq′ = outseq)

(b′ = b)

(AA′ = AA $\oplus$ {w+1 $\mapsto$ AA(r+1), r+1 $\mapsto$ AA(w+1)}) $\wedge$ (w′ = w+1) $\wedge$ (r′ = r+1)

MBr4

Ξ MDec

(AA(w+1) $\neq$ *WHITE*) $\wedge$ (AA(w+1) $\neq$ *BLUE*) $\wedge$ (AA(w+1) $\neq$ *RED*)

The loop body is thus defined by:

$$MLBody \triangleq [MDec \mid w \neq b-1] \land (MBr1 \lor MBr2 \lor MBr3 \lor MBr4)$$

The statement executed just before this loop is $[\![r,w,b := 0,0,N+1]\!]$, which gives us the loop initialisation schema:

```
┌── MLInit ──────────────────
│ Δ MDec
│
├───────────
│ (outseq′ = outseq) ∧ (inseq′ = inseq)
│ AA′ = AA
│ (r′ = 0) ∧ (w′ = 0) ∧ (b′ = N+1)
│
└──────────────────────────
```

We can now attempt to formulate the loop invariant:

```
┌── MLInv ───────────────────
│ MDec
│
├───────────
│ ∀ j : ℕ | 1 ≤ j ≤ N •
│   j ∈ {1 .. r}  ⇒  AA(j) = RED
│   j ∈ {r+1 .. w}  ⇒  AA(j) = WHITE
│   j ∈ {b+1 .. N}  ⇒  AA(j) = BLUE
│
└──────────────────────────
```

To prove that it is an invariant, we will need to show that:

$$(\mathrm{MStat1;MLInit} \Rightarrow \mathrm{MLInv}) \ \wedge \ ((\mathrm{MLInv} \wedge \mathrm{MLBody}) \Rightarrow \mathrm{MLInv}')$$

We will not go through the details of the proof here; this is dealt with in any of the texts which discuss this problem. On termination of the loop we have $w = b\text{-}1$, and so we can fill the other variables into what is essentially $MLInv' \wedge (w' = b'\text{-}1)$ to get the schema:

```
┌── MLoop ──────────────
│ $\Delta$ MDec
├──────────
│ $(\mathrm{outseq}' = \mathrm{outseq}) \ \wedge \ (\mathrm{inseq}' = \mathrm{inseq})$
│ $\mathrm{w}' = \mathrm{b}'\text{-}1$
│ $\forall\, \mathrm{j} : \mathbb{N} \mid 1 \le \mathrm{j} \le N \cdot$
│   $\mathrm{j} \in \{1 \,..\, \mathrm{r}'\} \ \Rightarrow \ \mathrm{AA}'(\mathrm{j}) = \mathit{RED}$
│   $\mathrm{j} \in \{\mathrm{r}'+1 \,..\, \mathrm{w}'\} \ \Rightarrow \ \mathrm{AA}'(\mathrm{j}) = \mathit{WHITE}$
│   $\mathrm{j} \in \{\mathrm{b}'+1 \,..\, N\} \ \Rightarrow \ \mathrm{AA}'(\mathrm{j}) = \mathit{BLUE}$
└────────────────────────
```

We can accordingly view our program as being represented by *MStat1 ; MLoop.*

We did not include a *write* statement at the end of the program, so there is no indication as to which variables we are interested in; presumably the partitioned array AA' is of interest, possibly we might want the indices r' and w'.

6.4 Integer Square Root

We'll take a small example next which we hope will reinforce some of the points made in § 6.1. The purpose of this program is to calculate the square root of a natural number, rounded down to the nearest whole number.

```
[Program ISR
x, y, z : Nat
m : Nat
read (m)
x,y,z := 0,1,1
while (y ≤ m) do
  x,y,z := x+1,z+2,y+z
od
write (x)
]
```

The declarations can be translated routinely to the schema:

$$\text{Dec} \mathrel{\hat{=}} [\text{inseq},\text{outseq} : \text{seq}\ \mathbb{Z};\ x,y,z,m : \mathbb{N}^{\perp}]$$

Let us split the rest of the program into three parts - the first two (initialisation) statements, the loop body, and the *write* statement; we get three corresponding schemas:

Init

Δ Dec

$(m = \perp) \wedge (x = \perp) \wedge (y = \perp) \wedge (z = \perp)$

$(\text{outseq}' = \text{outseq}) \;\wedge\; (\text{inseq}' = \text{tail}(\text{inseq}))$

$m' = \text{head}(\text{inseq})$

$(x' = 0) \wedge (y' = 1) \wedge (z' = 1)$

```
┌── LBody ──────────────────
│ Δ Dec
├──────────
│ (outseq′ = outseq)  ∧  (inseq′ = inseq)
│ y ≤ m
│ (x′ = x+1) ∧ (y′ = y+2) ∧ (z′ = z+y) ∧ (m′ = m)
└─────────────────────────
```

```
┌── Outp ──────────────────
│ Δ Dec
├──────────
│ (outseq′ = <x>)  ∧  (inseq′ = inseq)
│ (x′ = x) ∧ (y′ = y) ∧ (z′ = z) ∧ (m′ = m)
└─────────────────────────
```

The whole program is thus defined as:

$$\text{Init} \; ; (\text{LBody}^{*} \wedge [\text{Dec}' \mid y' > m']) \; ; \text{Outp}$$

To formulate the rather simple invariant we can attempt to relate the values of the variables to the number of iterations of the loop (and thus transitively to each other). If we wish to refer to the value of the variable x after i iterations of the loop, assuming that it iterates that many times, then we can write this in terms of i compositions of schema for the loop body: it is just $LBody^{i}.x'$.

[*Aside* - this definition is based on the assumption that the loop has not terminated before i iterations; if we did not make this assumption, things would be a little bit more complicated. We would first need to define something like $S^{*<k}$ for any schema S and

$k : \mathbb{N}$, to mean $S^0 \vee S^1 \vee ... \vee S^{k-1}$ (a restriction of the reflexive-transitive closure to at most $k-1$ applications). Then we can write

$$((\mathrm{LBody}^{*<i} \wedge [\mathrm{Dec}' \mid y' > m']) \vee \mathrm{LBody}^i).x'$$

to denote the value of x after i "iterations", even if the loop has actually terminated in fewer iterations (we could regard the rest of these "iterations" as just being successive Ξ *Dec* operations). Note that the uniqueness property that we proved in § 5.2.1 is central to our definition. *End of aside*]

The repeated application of addition is obviously central to our program. A consequence of this is that we will need a notation to express summation of all members of a set of integers; we define[1] one as:

$$[\Sigma : \mathbf{F}\, \mathbb{Z} \rightarrow \mathbb{Z} \mid (\Sigma\{\} = 0) \wedge (\forall\ S : \mathbf{F}_1\, \mathbb{Z}, i : \mathbb{Z} \mid i \in S \cdot \Sigma S = i + \Sigma(S/\{i\}))]$$

We would thus write $\Sigma\{i : 1\ ..\ n \cdot P(i)\}$ instead of the more standard $\sum_{i=1}^{n} P(i)$.
(Our notation has the advantage of being more consistent with the rest of our presentation, and of being much simpler to write!).

From the schema *LBody* we can postulate that for some $k : \mathbb{N}$ such that $pre\text{-}LBody^{k+1}$ (ie. the loop has not terminated), we will have: $LBody^k.x' = k$, $LBody^k.z' = 1 + 2k$ and $LBody^k.y' = 1 + \Sigma\{i : 1\ ..\ k \cdot LBody^i.z'\}$. The last of these can of course be expressed as the summation $1 + \Sigma\{i : 1\ ..\ k \cdot (1 + 2i)\}$, or $\Sigma\{i : 0\ ..\ k \cdot (1 + 2i)\}$.

The loop will terminate after k iterations if the loop guard is false; we can thus express the function of our program as finding the smallest such k where the guard is true after $k-1$ iterations but false after one more iteration, and then taking the value of x as this

[1]This is similar to the notation defined in [Back86] § 2.4

point. We can express this as:

$$\min \{k : \mathbb{N} \mid (LBody^{k-1}.y' \leq m) \wedge (m < LBody^{k}.y') \cdot LBody^{k}.x'\}$$

$$= \min \{k : \mathbb{N} \mid \Sigma\{i : 0 .. k \cdot (1 + 2i)\} \leq m < \Sigma\{i : 0 .. k \cdot (1 + 2i)\}\}$$

We could feel justified in leaving things at this point - however, there is another piece of information that we could use (if we happened to be familiar with it), namely that $\Sigma\{i : 0 .. k \cdot (1 + 2i)\} = (k+1)^2$, ie. the sum of the first *k* odd numbers is actually the square of *k*. We can thus rewrite the purpose of our program as finding some *k* such that we have:

$$\min \{k : \mathbb{N} \mid k^2 \leq m < (k+1)^2\}$$

which is the square root of *m* rounded down to the nearest whole number (and is thus in accordance with the name given to the program!)

An important point of the above discussion is that had we not known that last property of the summation of odd numbers, then we would have ended up with a different (but equivalent) result. If we were attempting to write a program to find the integer square root of a number, and we did not know this property of the squares of numbers, then we would have ended up with a different (but equivalent) program. Thus we see that this property will have played an equally important part whether we were going from program to specification or vice-versa, which reinforces the argument we made in § 6.1.

This is not true for just the last equivalence above; during our reasoning we also made use of the fact that $\Sigma\{i : 1..k \cdot 1\} = k$ and that $\Sigma\{i : 1..k \cdot 2\} = 2k$; neither of these are very difficult to work out, but we must acknowledge their role in connecting the program and specification. Accordingly, we observe that the process in which we are interested is founded on a wealth of such equivalences (ie. mathematical properties) which allow us to move from one expression of a problem to another; the properties which have been used in deriving programs from specifications will thus be symmetric to those needed for extracting specifications from programs.

6.5 McCarthy's 91 Function

We now look at an example of the involvement of a schema-based definition of a program being used in "common sense" type reasoning about that program. Our specimen this time is the rather simple looking program:

```
[Program Mc91
x, y : Nat
read (x)
y := 1
while (y ≠ 0) do
  if x ≤ 100 then
    x,y := x+11,y+1
  else
    x,y := x-10,y-1
  fi
od
write (x)
]
```

We will skip the definition for the first two (initialisation) statements, and proceed directly to examine the schemas which represent the two branches of the loop. If we let $Dec \mathrel{\hat{=}} [inseq,outseq : seq\ \mathbb{Z};\ x,y : \mathbb{N}^{\perp}]$, and since the loop body does not involve any IO, we let $IDec \mathrel{\hat{=}} [\Delta\ Dec \mid (inseq' = inseq) \wedge (outseq' = outseq)]$, then we get:

Br1

IDec

$(y > 0) \wedge (x \leq 100)$

$(y' = y+1) \wedge (x' = x+11)$

and

Br2

IDec

$(y > 0) \wedge (x > 100)$

$(y' = y-1) \wedge (x' = x-10)$

Rather than attempt to formulate an invariant at this point, it is useful to take a slightly "operational" view of things. First of all, we note that if the loop terminates after some iteration then, since $y > 0$ in the loop body, we must have executed the second branch of the conditional: this corresponds to the schema $Br2\&Ex \mathrel{\hat{=}} Br2 \wedge [Dec' \mid y' = 0]$. We note the special situation which occurs when the value of x that is input is greater than 100 and the program terminates after just one iteration; this situation can be characterised by the schema $1Br2\&E \mathrel{\hat{=}} [Dec \mid x > 100 \wedge y = 1] \wedge Br2\&Ex$. The expansion of both these schemas is:

Br2&Ex

IDec

$(y > 0) \wedge (x > 100)$

$(y' = 0) \wedge (x' = x-10)$

and

1Br2&E

IDec

$(y = 1) \wedge (x > 100)$

$(y' = 0) \wedge (x' = x-10)$

If $x \leq 100$ initially, then we are guaranteed at least two iterations, either *Br1;Br1* or *Br1;Br2*, which we can simplify to:

Br1Br1

IDec

$(y \geq 1) \wedge (x \leq 89)$

$(y' = y+2) \wedge (x' = x+22)$

and

Br1Br2

IDec

$(y \geq 1) \wedge (90 \leq x \leq 100)$

$(y' = y) \wedge (x' = x+1)$

Similarly, the next iteration can be either of these, or either *Br2;Br2*, or *Br2;Br1*, which can be simplified to:

Br2Br2

IDec

$(y \geq 2) \wedge (x \geq 111)$

$(y' = y-2) \wedge (x' = x-20)$

and

Br2Br1

IDec

$(y \geq 2) \wedge (101 \leq x \leq 110)$

$(y' = y) \wedge (x' = x+1)$

Leaving aside the case where $x > 100$ on input and *1Br2&E* holds, we can characterise the loop body by the disjunction of the other five schemas above. If we analyze these schemas in terms of the possible ranges of value for *x, x´, y* and *y´* we get:

<u>Br1Br1</u>	<u>Br1Br2</u>	<u>Br2Br2</u>	<u>Br2Br1</u>	<u>Br2&Ex</u>
$x \leq 89$	$90 \leq x \leq 100$	$x \geq 111$	$101 \leq x \leq 110$	$x \geq 101$
$y \geq 1$	$y \geq 1$	$y \geq 2$	$y \geq 2$	$y = 1$
$x' \leq 111$	$91 \leq x' \leq 101$	$x' \geq 91$	$102 \leq x' \leq 111$	$x' \geq 91$
$y' \geq 3$	$y' \geq 1$	$y' \geq 0$	$y' \geq 2$	$y' = 0$

Let us allow ourselves some terminological leeway to talk about the "execution" of a schema when we really mean the execution of the corresponding statements. We can assert that the last three of the above schemas are special in that they will only be executed following one of the first two (since only *Br1Br1* or *Br1Br2* can correspond to the first two iterations of the loop). Based on the ranges for the after-state variables, we note that *Br2Br2* can only be executed after either *Br1Br1* or *Br2Br1*, and that *Br2&Ex* can only happen after *Br1Br2*. We can strengthen the pre-conditions of these two schemas to reflect this (and consequently narrow the range of possible values for x' and y') to get:

Br1Br1	Br1Br2	Br2Br2	Br2Br1	Br2&Ex
$x \leq 89$	$90 \leq x \leq 100$	$x = 111$	$101 \leq x \leq 110$	$x = 100$
$y \geq 1$	$y \geq 1$	$y \geq 2$	$y \geq 2$	$y = 1$
$x' \leq 111$	$91 \leq x' \leq 101$	$x' = 91$	$102 \leq x' \leq 111$	$x' = 91$
$y' \geq 3$	$y' \geq 1$	$y' \geq 0$	$y' \geq 2$	$y' = 0$

We will not go into the details, but it is plain from this table that the program must terminate (based on reasoning such as: "once the pre-conditions of *Br1Br1* have become false, they can never become true again" and so on). There are only two schemas here which can possibly correspond to termination (ie $y' = 0$): these are *Br2Br2* and *Br2&Ex*, both of which establish $x' = 91$.

Bringing back the situation where *1Br2&E* holds, we can now formulate a schema to represent our program. If we assume that the initial value of *outseq* was < >, then we can represent the program in terms of an operation on just the input and output sequences:

```
┌── Mc91 ──────────────────
│ inseq, outseq : seq ℤ
├──────────────
│ inseq′ = tail(inseq)
│ (head(inseq) > 100) ⇒ (outseq = <head(inseq)-10>)
│ (head(inseq) ≤ 100) ⇒ (outseq = <91>)
└──────────────────────────
```

This example illustrates the usefulness of our schema definitions even in what seems to be an "informal" type of reasoning about programs. We mentioned earlier that we would not approach the problem by formulating a loop invariant directly; however, we have implicitly formulated such an invariant, albeit not in the usual form. If we let *X* denote the initial value of the variable *x*, then directly from the table we can formulate the schema which describes the values of the variables where the input is less than or equal to 100:

```
┌── LE100 ─────────────────
│ Dec; X : ℕ
├──────────────
│ X ≤ 100
│ ((x ≤ 89 ∧ y ≥ 1) ∨ (x ≤ 111 ∧ y ≥ 3) ∨
│ (90 ≤ x ≤ 100 ∧ y ≥ 1) ∨ (91 ≤ x ≤ 101 ∧ y ≥ 1) ∨
│ (x = 111 ∧ y ≥ 2) ∨ (x = 91 ∧ y ≥ 0) ∨
│ (101 ≤ x ≤ 110 ∧ y ≥ 2) ∨ (102 ≤ x ≤ 111 ∧ y ≥ 2) ∨
│ (x = 100 ∧ y = 1) ∨ (x = 91 ∧ y = 0))
└──────────────────────────
```

We also have the situation corresponding to an input greater than 100:

$$GT100 \triangleq [Dec;\ X : \mathbb{N} \mid (X > 100) \wedge (x = X\text{-}10) \wedge (y = 0)]$$

and the initial states just before loop execution:

$$LInit \triangleq [Dec;\ X : \mathbb{N} \mid (X = x) \wedge (y = 1)]$$

The loop invariant is thus $LInit \vee LE100 \vee GT100$. If we conjoin this with the negation of the loop guard, $y = 0$, we get the required result.

6.6 A Singly-Linked List

We will now look at an example which makes use of a singly-linked list. We will not examine a complete program this time - just some declarations and three procedures. The program represents a simplified view of a directory as consisting of an array, each entry of which corresponds to one file. A file is deemed to consist of a label (ie. a name) and a linked-list of blocks, which hold the contents of the file.

We are assuming that the identifers *NAME* and *BLOCK* have already been appropriately declared, and will thus assume that equivalent declarations exist in our specification. We will also assume some integer value *S* which represents the total number of blocks in the system at any time, and two integers *EMPTY* and *FULL* which will be used to represent error conditions.

The global declarations for our program are therefore:

```
[EMPTY, FULL : Z]
[S : ℕ]

system : array 1 to S of BLOCK

bnode : record
  contents : BLOCK
  next : ↑ bnode
endrec

freelist : ↑ bnode

direntry : record
  label : NAME
  blist : ↑ bnode
endrec

dir : array 1 to 33 of direntry
```

If we let the Z identifiers *BNODE* and *DIRENTRY* correspond to *bnode* and *direntry* respectively, then these record declarations will produce the corresponding Z declarations:

$$\text{BNODE} \;:\; \{\text{"contents"}\} \rightarrow \text{BLOCK}^{\perp} \;\cup\; \{\text{"next"}\} \rightarrow \text{MLoc}^{\perp}$$

$$\text{DIRENTRY} \;:\; \{\text{"label"}\} \rightarrow \text{NAME}^{\perp} \;\cup\; \{\text{"blist"}\} \rightarrow \text{MLoc}^{\perp}$$

The array *dir* is simply mapped to $DIR : \{1 .. 32\} \rightarrow DIRENTRY^{\perp}$.

Thus we regard a directory as consisting of 32 entries, each of which consists of a name and a list of blocks. The pointer *freelist* holds the address of the first block in a linked list of blocks which have not been assigned to any file in the directory.

The purpose of this example is to demonstrate our treatment of data structures involving pointers; thus we will not be examining the actual mapping from the program to Z in detail - we will just present the relevant schemas. We will consider only three procedures. Procedure *initialise* simply assigns all the system blocks to the free list and initialises the directory to "empty". We can add a new file to the directory using *create*, which takes the filename and the number of blocks to be allocated as parameters; note that this will automatically overwrite any previous entry. We delete a file using the procedure *erase*, which simply adds the relevant blocks to the free block list, and marks the directory entry as "empty".

```
[Proc initialise ( )
i : Nat
tmp : ↑ bnode
i := 1
while i ≤ 32 do
  dir[i].label,dir[i].blist,i := NULL,NULL,i+1
od
new (freelist  bnode)
↑(freelist).contents,i,tmp := SYSTEM[0],1,freelist
while i < S do
  new (↑(tmp).next  bnode)
  tmp := (↑tmp).next
  ↑(tmp).contents,↑(tmp).next  :=  SYSTEM[i],NULL
od]
```

```
[Proc erase (val n : NAME)
i : Nat
tmp,lend : ↑ bnode
i := 1
while (i < 32) and (dir[i].label ≠ NULL) and (dir[i].label ≠ n) do
  i:= i+1
od
if (dir[i].label = n) then
  tmp,lend := dir[i].blist,NULL
  while (tmp ≠ NULL) do
    tmp,lend := ↑(tmp).next,tmp
  od
  if (lend ≠ NULL) then
    dir[i].label := NULL
    dir[i].blist,freelist,↑(lend).next  :=  NULL,dir[i].blist,freelist
  fi
fi]
```

```
[Proc create (val n : NAME  val nb : Nat)
i,j : Nat
tmp : ↑ bnode

call erase (n)
i := 1
while (i < 32) and (dir[i].label ≠ NULL) do
  i := i+1
od
if (dir[i].label = NULL) then
  tmp,j := freelist,1
  while (tmp ≠ NULL) and (j < nb) do
    tmp,j := ↑(tmp).next,j+1
  od
  if (tmp ≠ NULL) then
    dir[i].label := n
    dir[i].blist,freelist := freelist,↑(tmp).next
    ↑(tmp).next := NULL
  else
    write (EMPTY)
  fi
else
  write (FULL)
fi]
```

There are only two error conditions, both of which apply to the creation of a file; an error occurs if there are not enough blocks left on the free list, or if there are no slots left in the array *dir*. The "message" corresponding to these situations is represented by the integer constants *EMPTY* and *FULL* respectively.

The constant *S : N* specifies the total number of blocks in the system at any one time; during the procedure *initialise* we use the *new* statement exactly *S* times to create the correct number of *BNODE*s to hold these blocks. The blocks are taken from the array *system* - we will not consider the initialisation for this array, or include it in our specification.

6.6.1 Defining schemas for the procedures

The program is fairly straightforward in operation. A file is created with *nb* blocks by simply taking the first *nb* blocks off the free list and linking them with the appropriate position in the array *DIR*. In order to delete a file we need to get to the last block in the list for that file (this is represented by the variable *lend*), and then link this to the first block in the free list. Before we deal with the procedures, it will be useful to define a function *linked*, which maps a node of a list into the set consisting of all those nodes which are "linked" after it.

$$\begin{array}{|l}
\text{linked} : \text{BNODE} \rightarrow \mathbf{P}\ \text{BNODE} \\
\hline
\forall\ \text{bn} : \text{BNODE} \cdot \\
\quad \text{bn} \in \text{linked(bn)} \\
\quad \forall\ \text{ln} : \text{BNODE} \mid (\text{ln} \in \text{linked(bn)}) \wedge (\text{ln("next")} \neq \bot) \cdot \\
\qquad \text{AT(ln("next"))} \in \text{linked(bn)}
\end{array}$$

We will note (without offering a proof) that two consequences of this definition are:

$$\forall\ \text{bn} : \text{BNODE} \cdot \exists_1\ \text{ln} : \text{BNODE} \cdot \text{ln} \in \text{linked(bn)} \wedge \text{ln("next")} = \bot$$

and

$$\forall\ \text{bn,ln,sn} : \text{BNODE} \cdot (\text{sn} \in \text{linked(ln)} \wedge \text{ln} \in \text{linked(bn)}) \Rightarrow \text{sn} \in \text{linked(bn)}$$

Using the schema $PSystem \triangleq BNODE;FREELIST;DIRENTRY;DIR;\ AT : MLoc \rightarrow GIVEN$ to represent the declarations of the system, we can now formulate a schema for the result of the procedure *initialise*:

```
── PInit ──────────────────
PSystem′
──────────────
```

$\forall\ i : 1 \mathinner{..} 32 \cdot DIR'(i) = \{\text{"label"} \mapsto \perp, \text{"blist"} \mapsto \perp\}$

$\exists\ P_1,\ldots P_S : MLoc,\ B_1,\ldots B_S : BLOCK \cdot$

$\quad AT'(P_1)(\text{"next"}) = P_2 \wedge \ldots \wedge AT'(P_{S-1})(\text{"next"}) = P_S \wedge AT'(P_S)(\text{"next"}) = \perp$

$\quad AT'(P_1)(\text{"contents"}) = B_1 \wedge \ldots \wedge AT'(P_S)(\text{"contents"}) = B_S$

$FREELIST' = P_1$

We ignore the predicate *(BNODE′ = BNODE)* ∧ *(DIRENTRY′ = DIRENTRY)* since it will appear in all the relevant schemas.

We will be dealing with the two procedures *create* and *erase* in their own right without examining them in an actual "calling" situation, and so we will take the liberty of adding the parameter declarations into the predicate part of each.

We can characterise procedure *erase* by the schema: $PErase \mathrel{\hat{=}} PDelete \vee PNotThere$, where the schema *PDelete* represents the situation where the file name was found in the directory, and its blocks are then returned to the free block list; *PNotThere* represents the situation where the file was not in the directory - we do not regard this as an error condition, we just do nothing.

If we assume that the schema *PIO* holds the declaration of *inseq* and *outseq*, then we can straightforwardly extract the following definitions from the program:

PDelete

Δ PSystem; Ξ PIO

n : NAME

$\exists\, i : \{1 \,..32\} \mid DIR(i)(\text{"label"}) = n \cdot$

$DIR' = DIR \otimes \{(i,\text{"label"}) \mapsto \bot, (i,\text{"blist"}) \mapsto \bot\}$

$FREELIST' = DIR(i)(\text{"blist"})$

$\exists\, lend : MLoc \mid (AT(lend) \in linked(AT(DIR(i)(\text{"blist"}))) \wedge AT(lend)(\text{"next"}) = \bot$

$AT' = AT \oplus \{(lend,\text{"next"}) \mapsto FREELIST\}$

PNotThere

Ξ PSystem; Ξ PIO

n : NAME

$\nexists\, i : \{1 \,..32\} \cdot DIR(i)(\text{"label"}) = n$

The schema which will represent *create* will contain *PErase*, since the corresponding procedure *erase* is called at the start. We define three schemas corresponding to the three possible scenarios: *PMake* represents the successful creation of the file (with the allocation of the appropriate number of blocks), *PEmpty* occurs when there are not enough blocks in the free block list, and *PFull* corresponds to the situation where there are no free positions in the array representing the directory.

Thus we can represent the creation of a new directory entry by the schema:

$$PCreate \mathrel{\hat{=}} PErase \; ; [PMake \vee PEmpty \vee PFull]$$

The schema *PMake* is defined as:

PMake

$\Delta\ \mathrm{PSystem};\ \ \Xi\ \mathrm{PIO}$

$n : \mathrm{NAME};\ nb : \mathbb{N}$

$\exists\, i : \{1 \,..32\} \mid i = \min\{j : \{1 \,..32\} \mid \mathrm{DIR}(j)(\text{"label"}) = \perp\}$

$\exists\, tmp : \mathrm{MLoc} \cdot \mathrm{AT}(tmp) \in \mathrm{linked}(\mathrm{AT}(\mathrm{FREELIST})) \wedge$

$\quad \#(\mathrm{linked}(\mathrm{AT}(tmp)) \ = \ \#\mathrm{linked}(\mathrm{AT}(\mathrm{FREELIST})) - (nb - 1)$

$\mathrm{DIR}' = \mathrm{DIR} \otimes \{(i,\text{"label"}) \mapsto n,\ (i,\text{"blist"}) \mapsto \mathrm{FREELIST}\}$

$\mathrm{FREELIST}' = \mathrm{AT}(tmp)(\text{"next"})$

$\mathrm{AT}' = \mathrm{AT} \otimes \{(tmp,\text{"next"}) \mapsto \perp\}$

The schemas *PEmpty* and *PFull* cater for the "error" situations where the first or second conditions of *PMake* do not hold:

PEmpty

$\Xi\ \mathrm{PSystem};\ \ \Delta\ \mathrm{PIO}$

$n : \mathrm{NAME};\ nb : \mathbb{N}$

$\exists\, i : \{1 \,..32\} \mid i = \min\{j : \{1 \,..32\} \mid \mathrm{DIR}(j)(\text{"label"}) = \perp\}$

$\nexists\, tmp : \mathrm{MLoc} \cdot \mathrm{AT}(tmp) \in \mathrm{linked}(\mathrm{AT}(\mathrm{FREELIST})) \wedge$

$\quad \#(\mathrm{linked}(\mathrm{AT}(tmp)) \ = \ \#\mathrm{linked}(\mathrm{AT}(\mathrm{FREELIST})) - (nb - 1)$

$(inseq' = inseq) \wedge (outseq' = outseq \frown \mathit{EMPTY})$

```
── PFull ──────────────
Ξ PSystem;  Δ PIO
n : NAME;  nb : ℕ
──────────
∄ i : {1 ..32} | i = min{j : {1 ..32} | DIR(j)("label") = ⊥}
inseq′ = inseq
outseq′ = outseq ⁀ FULL
──────────────────────
```

6.6.2 Linked lists as sequences

If we were to attempt to abstract the schemas *PInit*, *PErase* and *PCreate* to a higher level, we might begin by trying to characterise *DIR* and *FREELIST* in some form other than arrays and linked lists: we will choose sequences.

Assuming that the types *NAME* and *BLOCK* are available to us, we can define:

$$\text{FREESEQ} \;==\; \text{seq BLOCK} \mid \#\,\text{FREESEQ} \leq S$$
$$\text{DIRSEQ} \;==\; \text{seq}\,(\text{NAME}^{\perp} \times \text{seq BLOCK}) \mid \#\,\text{DIRSEQ} = 32$$

and then let $ASystem \mathrel{\hat{=}} FREESEQ;DIRSEQ$.

We will next need to describe the relationship between these "abstract" definitions with the "concrete" definitions *FREELIST* and *DIR*. In the manner discussed in § 3.2, we can approach this by first of all relating linked lists and sequences; we do this using the following schema:

$$
\begin{array}{|l}
\text{MakeSeq} : \text{MLoc} \rightarrow (\text{MLoc} \rightarrow \text{GIVEN}) \rightarrow \text{seq BLOCK} \\
\hline
\text{MakeDeq} = \lambda\ \text{mloc} : \text{MLoc},\ \text{AT} : \text{MLoc} \rightarrow \text{GIVEN} \cdot \\
\qquad\qquad \mu\ \text{bseq} : \text{seq BLOCK} \mid \\
(\text{mloc} = \bot) \Leftrightarrow (\text{bseq} = \langle\ \rangle) \\
\exists\ \text{bn} : \text{BNODE} \cdot \text{AT}(\text{mloc}) = \text{bn} \Rightarrow \\
\ (\text{bn}(\text{"contents"}) = \text{head}(\text{bseq}) \\
\quad \forall\ i : \{1 \ldots \#\text{bseq}-1\},\ \text{ln} : \text{BNODE} \mid \text{ln} \in \text{linked}(\text{bn}) \cdot \\
\qquad (\text{ln}(\text{"contents"}) = \text{bseq}(i) \wedge (\text{ln}(\text{"next"}) \neq \bot) \\
\qquad \Rightarrow \text{AT}(\text{ln}(\text{"next"}))(\text{"contents"}) = \text{bseq}(i+1)) \\
\qquad (\text{ln}(\text{"next"}) = \bot \Rightarrow \text{ln}(\text{"contents"}) = \text{last}(\text{bseq})))
\end{array}
$$

Thus the function *MakeSeq* will take what is basically a "pointer" to the first node in a linked list of blocks, and return a sequence which corresponds to it in the obvious way. We note that for the node *bn* and sequence *bseq* as defined in the above function, we have *#bseq = #linked(bn)*.

We can now use this function in the schema *Rel* which describes the relationship between the abstract and concrete declarations. The "list" *FREELIST* and sequence *FREESEQ* will correspond exactly (using *MakeSeq*, of course), while for each entry in *DIR*, we will expect its counterpart in the sequence *DIRSEQ* to have the same label, and to contain a corresponding (again via *MakeSeq*) list of blocks.

Rel

PSystem; ASystem

MakeSeq(FREELIST AT) = FREESEQ

$\exists$ i : {1 .. 32} ·

$\exists$ nm : NAME, bs : seq BLOCK · DIRSEQ(i) = (nm,bs) $\Rightarrow$

DIR(i)("label") = nm

MakeSeq(DIR(i)("blist") AT) = bs

We can assert that a suitable initialisation for the abstract model is the schema

AInit

ASystem′

$\forall$ i : 1 .. 32 · DIRSEQ′(i) = ($\perp$, < >)

$\exists$ $B_1,\ldots B_S$: BLOCK · $\{B_1,\ldots B_S\}$ = ran(FREESEQ′)

We can then write the proof obligation for the initialisation as requiring us to show that: *PInit* $\Rightarrow$ *(AInit* $\wedge$ *Rel)*, which is trivially true.

In a similar manner we can formulate the abstract equivalent of file erasure as being represented by the schema *AErase* $\hat{=}$ *ADelete* $\vee$ *ANotThere*, where we have:

ADelete

Δ ASystem; Ξ PIO

n : NAME

$\exists$ i : {1 ..32}, bs seq BLOCK $\cdot$ DIRSEQ(i) = (n,bs)

DIRSEQ′ = DIRSEQ $\otimes$ {i $\mapsto$ ($\perp$, < >)}

FREESEQ′ = bs $\frown$ FREESEQ

ANotThere

Ξ ASystem; Ξ PIO

n : NAME

$\nexists$ i : {1 ..32}, bs : seq BLOCK $\cdot$ DIRSEQ(i) = (n,bs)

Since we are given that the lists starting at addresses *FREELIST* and *DIR(i)("blist")* correspond to the sequences *FREESEQ* and the second element of the tuple *DIRSEQ(i)* respectively, discharging the proof obligation will hinge on showing that:

$$\text{FREESEQ}' = \text{bs} \frown \text{FREESEQ}$$

or, in terms of the elements of *PDelete*,

$$\text{MakeSeq(FREELIST}'\ \text{AT}') =$$
$$\text{MakeSeq(DIR(i)("blist") AT)} \frown \text{MakeSeq(FREELIST AT)}$$

which is the same as showing that:

$$\forall\ i : \{1 .. \#bs\},\ j : \{1 .. \#FREESEQ\} \cdot$$
$$\quad \mathrm{MakeSeq}(FREELIST'\ AT')(i) = \mathrm{MakeSeq}(DIR(i)(\text{"blist"})\ AT)(i)$$
$$\wedge\ \mathrm{MakeSeq}(FREELIST'\ AT')(\#bs+i) = \mathrm{MakeSeq}(FREELIST\ AT)(i)$$

We will not detail the proof of this here as it is fairly routine, since we know that no node changes its *"contents"* field, and only the last node in *linked(AT(DIR(i)("blist"))* changes its *"next"* field.

We could continue in this manner to provide a representation for *PCreate*, the main part of which will involve adding the sequence of blocks *FREESEQ for nb* to the appropriate tuple in *DIRSEQ*, and asserting that *FREESEQ´ = FREESEQ after nb*. The proof obligation here will centre around finding a representation for the sequence operations *for* and *after* in terms of lists; this is similar to the proof obligation above which involved sequence concatenation.

CONCLUSION

The central aim of this paper has been to explore the program-specification link using the formal specification language Z as a unifying notation. The three most important aspects of this were:

1. The provision of a predicate-based schema calculus which we believe can facilitate reasoning about general instances of schemas. We presented a basis for this calculus in the ordinary propositional-like Z calculus, and reasserted the standard definitions such as schema pre-conditions and composition. The proof obligations for specification realisation were explored in detail using this calculus.

2. We specified a collection of semantic functions which can be used to provide a formal semantics for a simple programming language in Z. Program variables were mapped into variables in a Z schema, and statements were interpreted as operations over these variables. The description of these functions is close to the denotational style, but the resulting schema which is constructed is based on the standard axiomatic approach. Features such as call-by-name and call-by-value parameter passing, and variables representing pointers have been incorporated into the model.

3. The purpose of giving a semantics in Z for the simple language was to provide a basis for the study of programs written in that language. We explored the implications of our definition and used it, along with the predicate-based schema calculus introduced earlier, to develop concepts such as coercions and the weakest pre-specification which can be used in manipulating specifications.

The semantics given in chapter 4 form the core of the thesis, in that they act as a foundation (and thus a justification) for many of the explorations in the rest of the work. Fundamental to this was the belief that the application of our semantics to a program should be relatively mechanical, should cover a range of features which was wide enough to be generally useful, and should result in the construction of a schema which could be intuitively identified with the initial program. We have endeavoured to spell out the semantic functions in as much detail as possible in order to fully explain and justify our mappings; perhaps some of them may be better understood in the light of the examples given in chapter 6.

While the language that we considered is relatively simple, it nevertheless contains many of the important features of a "usable" programming language. We have not provided functions (ie. procedures which "return" a value) because they do not added to the power of the language in any way. There is no significant barrier to the introduction and definition of functions: for our purposes this would have added extra complexity to the definition with little material gain.

Similarly, we have not allowed a "full scale" memory model where we could access the address of arbitrary variables. Again we do not see any great difficulty in providing such a definition - it would simply involve an extension of the function *AT* which we defined, so that all the variables currently in scope were in its range. One feature of our semantics is that we have mapped program identifiers to Z identifers, and associated a memory location with these, rather than mapping the program identifier to a memory location: this allows us to ignore the issue of memory allocation in pieces of code that do not make use of it.

One notable omission from our language is recursion (either direct or indirect). An essential part of any further work on these semantics would be the provision of some facility which captured the meaning of a recursive call: we cannot, as yet, point to an approach which would allow for the incorporation of recursion. However, our experiences in formulating the semantics given here would indicate that it should be possible to extend our framework to include most additional features, since we have not deviated in essence from the main semantic techniques.

As can be judged from the examples in chapter 6, we do not consider it essential that every program be dealt with in minute formal detail; however, any rigorous treatment will need to have the assurance that such a formal framework exists in order to be used with confidence. This underlines the importance of the intuitive association between a program and its corresponding semantic specification - with experience, we can construct such a specification without going through the detail of all the functions given in chapter 4, but we will know that our work has a formal basis, and that this basis can act as a final arbitrator should our results be challenged.

REFERENCES

[Apt81]	K.R. Apt. Ten Years of Hoare's Logic - A Survey. *ACM Trans on Prog Languages and Systems*, Vol 3 No 4, (Oct. 1981), 431-483
[ArMa86]	M.A. Arbib & E.G. Manes. *Algebraic Approaches to Program Semantics*. Springer-Verlag, 1986
[Babe87]	Robert L. Baber. *The Spine of Software*. John Wiley & Sons, 1987
[Back86]	Roland C. Backhouse. *Program Construction and Verification*. Prentice-Hall, 1986
[BeBi80]	Pierre Berlioux & Philippe Bizard. *Algorithms - The construction, proof and analysis of programs*. Prentice-Hall, 1980
[BjJo82]	D. Bjørner & C.B. Jones. *Formal Specification and Software Development*. Prentice-Hall, 1982
[BrSc82]	M. Broy & G. Schmidt (eds.). *Theoretical Foundations of Programming Methodology*. D. Reidel Publishing Company, 1982.
[BuGo77]	R.M. Burstall & J.A. Goguen. Putting Theories together to make Specifications. *Proc 5th International Joint Conference on AI*, 1045-1058
[Ders83]	Nachum Dershowitz. *The Evolution of Programs*. Birkhauser, 1983
[Dijk75]	Edsger W. Dijkstra. Guarded Commands, Nondeterminicity, and Formal Derivation of Programs. *Communications of the ACM*, Vol 18 No. 8 (August 1975), 453-457 (Reprinted in [Grie78])
[Dijk76]	Edsger W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1986.
[Dijk90]	Edsger W. Dijkstra. *Formal Development of Programs and Proofs*. Addison-Wesley, 1990
[Dill90]	Antoni Diller. *Z - An Introduction to Formal Methods*. John Wiley & Sons, 1990
[Drom89]	R.G. Dromey. *Program Derivation - The development of programs from specifications*. Addison-Wesley, 1989
[EhMa85]	H. Ehrig & B. Mahr. *Fundamentals of Algebraic Specification 1*. Springer-Verlag, 1985

[GHW85] J.V. Guttag, J.J. Horning & J.M. Wing. The Larch Family of Specification Languages. *IEEE Software*, Vol 2 No 5, (Sept 1985), 24-36

[Gord79] Michael Gordon. *The Denotational Description of Programming Languages*. Springer-Verlag, 1979.

[Grie78] David Gries (ed.) *Programming Methodology - A collection of Articles by Members of IFIP WG 2.3*. Springer-Verlag, 1978.

[Grie81] David Gries. *The Science of Programming*. Springer-Verlag, 1981

[GrPr85] D. Gries & J. Prins. A New Notion of Encapsulation. *Proceedings of the Symposium on Language Issues in Programming Environments.* SIGPLAN, 1985

[GuHo86a] J.V. Guttag & J.J Horning. Report on the Larch Shared Language. *Science of Computer Programming*, Vol 6, (1986), 103-134

[GuHo86b] J.V. Guttag & J.J Horning. A Larch Shared Language Handbook. *Science of Computer Programming*, Vol 6, (1986), 135-157

[HaJo89] I.J. Hayes & C.B. Jones. Specifications are not (necessarily) executable. *Software Engineering Journal.* Vol 4, No 6 (November 1989), 330-338

[Haye87] I. Hayes (ed.). *Specification Case Studies*. Prentice-Hall, 1987

[Hehn84] E.C.R. Hehner. Predicative Programming Part 1. *Communications of the ACM*, Vol 27 No. 2 (February 1984), 134-143

[Hehn90] E.C.R. Hehner. A Practical Theory of Programming. *Science of Computer Programming*, Vol 14 (1990), 133-158

[Herm69] H. Hermes. *Enumerability, Decidability, Computability*. Springer-Verlag, 1969

[HHS87] C.A.R. Hoare, He, Jifeng & J.W. Sanders. PreSpecification in Data Refinement. *Information Processing Letters*, Vol 25 (May 1987), 71-76

[Hoar69] C.A.R. Hoare. An Axiomatic Basis for Computer Programming. *Communications of the ACM*, Vol 12, No. 10 (October 1969), 576-581 (Reprinted in [Grie78])

[Hoar87] C.A.R. Hoare et al. Laws of Programming. *Communications of the ACM*, Vol 30, No. 8 (August 1987), 672-686

[HoHe87] C.A.R. Hoare & He, Jifeng. The Weakest PreSpecification. *Information Processing Letters*, Vol 24 (1987), 127-132

[HoSh85] C.A.R. Hoare & J.C. Shepherdson (eds.). *Mathematical Logic and Programming Languages.* Prentice-Hall, 1985

[HoWi73] C.A.R. Hoare & N. Wirth. An Axiomatic Definition of the Programming Language PASCAL. *Acta Informatica*, Vol 2 (1973), 335-355

[Jone80] C.B. Jones. *Software Development - A Rigorous Approach.* Prentice-Hall, 1980

[Jone90] C.B. Jones. *Systematic Software Development Using VDM.* Prentice-Hall, 1990

[Jose88] M.B. Josephs. The Data Refinement Calculator for Z Specifications. *Information Processing Letters*, Vol 27 (1988), 29-33

[JoSh90] C.B. Jones & R.C. Shaw. *Case Studies in Systematic Software Development.* Prentice-Hall, 1990

[KiSo89] S. King & I.H. Sørensen. From Specification, through Design, to Code: A Case Study in Refinement. In *[McDe89]*

[Lips81] John D. Lipson. *Elements of Algebra and Algebraic Computing.* Benjamin/Cummings, 1981

[Lutz90] Earlin Lutz. Some Proofs of Data Refinement. *Information Processing Letters*, Vol 34 (1990), 179-185

[Mann74] Zohar Manna. *Mathematical Theory of Computation.* McGraw-Hill, 1974

[Mann80] Zohar Manna. *Lectures on the Logic of Computer Programming.* Society for Industrial and Applied Mathematics, 1980

[McDe89] J.A. McDermid (ed). *The Theory and Practice of Refinement.* Butterworths, 1989

[MDR86] A. Mili, J. Desharnais & J.R. Gagné. Formal Models of Stepwise Refinement of Programs. *ACM Computing Surveys*, Vol 18, No. 3 (September 1986), 231-276

[Morg88] C. Morgan. The Specification Statement. *ACM Trans on Prog Languages and Systems*, Vol 10, No. 3 (July 1988) 403-419

[Morr90a] J.M. Morris. Programs from Specifications. In *[Dijk90]*

[Morr90b] J.M. Morris. Piecewise Data Refinement. In *[Dijk90]*

[MoVi90] C. Morgan & T. Vickers. Types and Invariants in the Refinement Calculus. *Science of Computer Programming*, Vol 14 (1990), 281-304

[Nels89] G. Nelson. A Generalization of Dijkstra's Calculus. *ACM Trans on Prog Languages and Systems*, Vol 11 No. 4 (October 1989) 517-561

[NHN80] R. Nakajima, M. Honda & H. Nakahara. Hierarchical Program Specification and Verification. *Acta Informatica*, 14, (1980), 135-155

[Niel89] D. Nielson. Hierarchical Refinement of a Z Specification. In [McDe89]

[Paga81] Frank G. Pagan. *Formal Specification of Programming Languages.* Prentice-Hall, 1981.

[Schm86] David A. Schmidt. *Denotational Semantics - A Methodology for Language Development.* W.C. Brown Publishing Company, 1986

[Spiv87] J.M. Spivey. *The Z Notation - A Reference Manual.* Prentice-Hall, 1987

[Spiv88] J.M. Spivey. *Understanding Z - A Specification Language and its Formal Semantics.* Cambridge University Press, 1988

[Spiv89] J.M. Spivey. An Introduction to Z and Formal Specifications. *Software Engineering Journal*, Vol 4 No 1, (Jan. 1989), 40-50

[Stoy77] Joseph E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory.* MIT Press, 1977.

[Tenn81] R.D. Tennent. *Principles of Programming Languages.* Prentice-Hall, 1981.

[TuMa87] W.M. Turski & T.S.E. Maibaum . *The Specification of Computer Programs.* Addison-Wesley, 1987

[Wing87] J.M. Wing. Writing Larch Interface Language Specifications. *ACM Trans on Prog Languages and Systems*, Vol 9 No 1, (Jan. 1987), 1-24

[Wood89] J.C.P. Woodcock. Structuring Specifications in Z. *Software Engineering Journal*, Vol 4 No 1, (Jan. 1989), 51-66

APPENDIX A - Summary of Z Notation

In what follows, *S* and *T* are sets, *P* and *Q* are predicates, *R* is a relation and *F* and *G* are functions.

A.1 Numbers

$\mathbb{N}$	Natural Numbers
$\mathbb{N}_1$	Natural numbers excluding 0
$\mathbb{Z}$	Integers

A.2 Declarations and Sets

$x : T$	*x* is of type *T* (x is a member of the set *T*)
$\hat{=}$	is defined as
$(S \times T)$	Cartesian product of sets *S* and *T*
$x : \mathbb{P}\ S$	*x* is a subset of *S* (a member of the power set of *S*)
$\{D \mid P \cdot x\}$	The set of all *t* for which *P* holds, given the declarations *D*

A.3 Logic Symbols

$P \wedge Q$	*P* and *Q* (conjunction)
$P \vee Q$	*P* or *Q* (disjunction)
$\neg P$	not *P* (negation)
$P \Rightarrow Q$	if *P* then *Q* (implication)
$P \Leftrightarrow Q$	*P* if and only if *Q* (bi-implication)
$\forall x : T \cdot P$	All *x* of type *T* satisfy *P* (universal quantification)
$\exists x : T \cdot P$	There exists some *x* of type *T* satisfying *P* (existential quantification)
$\exists_1 x : T \cdot P$	There exists a unique *x* of type *T* satisfying *P*

$$\forall x : T \mid P \cdot Q \ \hat{=}\ \forall x : T \cdot P \Rightarrow Q$$
$$\exists x : T \mid P \cdot Q \ \hat{=}\ \exists x : T \cdot P \wedge Q$$

A.4 Sequences

x : seq T	*x* is a sequence of elements of type *T*
$<x_1, \ldots x_n>$	The sequence consisting of $x_1,..x_n$
< >	The empty sequence

head $<x_1,x_2,\ldots,x_n>$	=	x_1
tail $<x_1,x_2,\ldots,x_n>$	=	$<x_2,\ldots x_n>$
front $<x_1,x_2,\ldots,x_n>$	=	$<x_1,\ldots x_{n-1}>$
last $<x_1,x_2,\ldots,x_n>$	=	x_n
$<x_1,\ldots x_n> \frown <x_m,\ldots x_k>$	=	$<x_1,\ldots x_n,x_m,\ldots x_k>$
$<x_1,\ldots x_i,x_{i+1},\ldots x_n>$ for i	=	$<x_1,\ldots x_i>$
$<x_1,\ldots x_i,x_{i+1},\ldots x_n>$ after i	=	$<x_{i+1},\ldots x_n>$

A.5 Relations and Functions

$F : S \leftrightarrow T$	*F* is a relation from *S* to *T*
$F : S \rightarrow T$	*F* is a total function from *S* to *T*
$F : S \nrightarrow T$	*F* is a partial function from *S* to *T*
$F : S \twoheadrightarrow\!\!\!\!\!\!+\; T$	*F* is a finite partial function from *S* to *T*
$F : S \rightarrowtail T$	*F* is an injective (one-one) function from *S* to *T*
$F : S \twoheadrightarrow T$	*F* is a surjective (onto) function from *S* to *T*
$F : S \rightarrowtail\!\!\!\!\twoheadrightarrow T$	*F* is a bijective function from *S* to *T*
$\{x \mapsto y\}$	The function consisting of the mapping from *x* to *y*
F^{-1}	The inverse of *F*
F ; G	*F* followed by *G* (relational composition)
R^k	*R* composed with itself *k* times (where *k : N*)
dom F	The domain of *F*
ran F	The range of *F*
$R(\!|x|\!)$	*R* applied to *x* (giving a subset of *ran R*)
F(x)	*F* applied to *x* (giving an element of *ran F*)
$F \oplus G$	A function which agrees with *G* for values in the domain of *G*, and agrees with *F* for all other values
$S \lhd G$	A function which is the same as *G* except that its domain is exactly *S*
$F \rhd T$	A function which is the same as *G* except that its range is exactly *T*