Modelling the Algebra of Weakest Preconditions

Ingrid Moira Rewitzky

September 1991

A thesis submitted in fulfilment of the requirements for the degree of

Master of Science

Supervisor: Prof Chris Brink

Department of Mathematics, University of Cape Town.

Acknowledgements

I express sincere thanks to my supervisor, Prof Chris Brink for his invaluable assistance and encouragement throughout the preparation of this thesis. His dedication, insight and enthusiasm, was and will remain a source of inspiration to me.

Furthermore I am extremely grateful to the Department of Mathematics for financial assistance and for all the opportunities offered to me since I joined in 1990. Collective thanks go to all members of staff, especially Lyn Waldron for her guidance with word processing and printing.

Thanks are also due to the Foundation of Research Development for financial assistance during my postgraduate studies.

Special thanks to my parents and sister for their unfailing support and encouragement. As a token of my appreciation I dedicate this thesis to them.

Contents

List of Tables

List of Figures

Preface

'The algebra of weakest preconditions' of the title refers to Edsger Dijkstra's ([1975], [1976]) *calculus of weakest preconditions*, viewed in an algebraic context.

In his classic monograph [1976], Dijkstra defined the notion of a *weakest precondition* to describe completely the meaning (or semantics) of a program. For any program α and any given postcondition Q which is desired to be true upon termination, the *weakest precondition* of α with respect to Q, written '$wp(\alpha, Q)$', is defined to be the set of all those states s such that execution of α from s will result in α terminating in a state where Q is true. Thus the program α induces a mapping $wp(\alpha, -)$ which maps any predicate Q onto another predicate $wp(\alpha, Q)$. Such a mapping is called a *predicate transformer* for α. Dijkstra also introduced a nondeterministic programming language called the *guarded command language.* To capture the intuitive meanings of these programs Dijkstra axiomatised some conditions on the weakest precondition predicate transformer. These conditions, often called *healthiness conditions*, constitute Dijkstra's *calculus of weakest preconditions.* Since his axiomatisation is equational it is appropriate to think of it as an *algebra.* In **Chapter 2** I give an exposition of the calculus and the algebra of weakest preconditions.

As algebras go one would expect the algebra of weakest preconditions to have some standard (preferably set-theoretic) model — as, for example, a standard model for Boolean algebras is the calculus of sets. 'Modelling' the algebra of weakest preconditions refers to the problem of finding such a reasonable model for the algebra of weakest preconditions. In this thesis I examine in detail two set-theoretic models of programs: the *relational model* in **Chapter 4** and the *flowsets model* in **Chapter 6**.

Dijkstra's weakest precondition semantics is well-suited to showing the correctness of programs. To specify the weakest precondition of a program α with respect to a given postcondition Q is one way of trying to *specify* what the program must do. The issue of *correctness* is that of a program satisfying or not satisfying its specification. I explain this in **Chapter 3**.

The essence of Dijkstra's approach to program semantics is to capture the meaning of programs not by considering the programs themselves, but by axiomatising the predicate transformers which they induce. In **Chapter 5** I consider the notion of a predicate transformer in a general algebraic context. I use the notion of the *power operation* of a given relation, as first introduced by Jónsson and Tarski ([1951], [1952]) in a seminal paper on Boolean algebras with operators.

In expounding the notions of pre- and postconditions, of termination and nontermination, of correctness and of predicate transformers I found that the same trivalent distinction played a major role in all contexts. Namely:

Initialisation properties:
An execution of a program always, sometimes or never starts from an initial state.
Termination/nontermination properties:
If it starts, the execution always, sometimes or never terminates.
Clean-/messy termination properties:
A terminating execution always, sometimes or never terminates *cleanly.*
Final state properties:
All, some or no final states of α from s have a given property.

I considered it worthwhile to attempt a thorough analysis and classification of the way in which these possibilities interact and determine what we mean by *correctness*, by *representing* programs and by *executing* programs. This presentation of core intuitions is done in **Chapter 1**.

In so far as this thesis makes any original contributions I take these to be:

(1) The case analysis of execution properties, representation methods, execution methods and notions of correctness in **Chapters 1** and **3**.

(2) Utilising the notion of *power algebra* for the study of predicate transformers.

(3) The flowsets model. (This is joint work with my supervisor; to appear in Brink and

Rewitzky [19?].)

The *Harvard* system is used in this thesis for literature referencing. In each chapter, definitions, lemmas, theorems, corollaries and examples are numbered consecutively in order of appearance. Within a chapter, say Chapter 2, '(3)' would for example refer to the third entity in that chapter, whether this be a definition, a lemma, a theorem, a corollary or an example. Outside Chapter 2, '(2.3)' would refer to this same entity. Figures and tables are numbered separately.

All mathematical symbols used are listed in the **Index of Notation**, with a page reference to their definition or first use . There is also a **List of Tables** and **List of Figures**.

Chapter 1

Introduction

In this chapter I introduce what I take to be the core intuitions behind any model of programs. The first part of this chapter focusses on different kinds of possible initialisation and execution properties of programs. I distinguish three initialisation properties and fifteen execution properties. In the second part I introduce various choices that must be made in modelling programs. These choices lead to twenty-four representation methods. I conclude the chapter with a mathematical characterisation of four different execution methods for programs. This chapter is for motivation only and my later exposition refers back where necessary to the distinctions and choices outlined here.

Any *execution* of a program starts in a certain *state*, and may or may not *terminate*. If it terminates it may do so *cleanly*, which means that it may do so in some state, or *messily*, which means that upon termination it is not in one of the states making up the *state space*. A set of states is called a *predicate*, and a program may be viewed as the set of all its possible executions. A program is called *nondeterministic* if from a given initial state different final states are possible.

Any model of programs should attempt to capture these intuitions. Thus the following distinctions will appear a number of times in this thesis.

(1) For any program α, and any state s, is s a possible initial state of α or not? [Intuitively: *From s, would* α *start, or not?*]

(2) For any execution of a program α, does it terminate or not? If it does, does it terminate cleanly or not? [Intuitively: *Once started, would α stop? If so, how?*]

(3) Does a given property hold *for all* or only *for some* elements of a set? [Intuitively, this occurs in a variety of contexts. For example, *Do all executions terminate, or only some? Do all final states have a property Q, or only some?*]

In the latter case, note that by negation two further cases arise. For example,

- Negating 'α always terminates' yields 'α sometimes does not terminate'
- Negating 'α sometimes terminates' yields 'α always does not terminate' (or 'α never terminates').
- Negating 'Every final state has property Q' yields 'Some final state does not have property Q'.
- Negating 'Some final state has property Q' yields 'Every final state does not have property Q' (or 'No final state has property Q').

Note also that 'sometimes' can be interpreted in two ways. First 'sometimes' can be taken in the sense of including the possibility of 'always'. That is, 'P sometimes holds' means 'there is at least one instance of P holding'. Alternatively, 'sometimes' can be taken in the sense of excluding the possibility of 'always'. That is, 'P sometimes holds' means 'there is at least one instance of P holding and there is at least one instance of P not holding'. In other words 'P sometimes holds' can be taken to mean either one of

- 'at least once P holds', or
- 'at least once P holds and at least once P does not hold'.

In this thesis I choose as default option the latter. The former will be explicitly indicated, where necessary, by writing something like 'P sometimes or always holds'. Analogously, 'some' can be taken to include or exclude the possibility of 'every', and unless otherwise indicated I will always use 'some' with the latter meaning.

These distinctions allow us to say the following of a program α, a predicate Q and an initial state s.

A **Initialisation properties** (from any initial state)

(1) α always starts from s, or

(2) α sometimes starts from s and

α sometimes does not start from s, or

(3) α never starts from s.

B **Termination / nontermination properties** (from any initial state)

(1) α always terminates from s, or

(2) α sometimes terminates from s and

α sometimes does not terminate from s, or

(3) α never terminates from s.

The terminating executions of a program α from an initial state s include cleanly terminating executions and messily terminating executions. So we get:

C **Clean- / messy termination properties** (from any initial state)

Under the assumption that α terminates, we have the three possibilities:

(1) α always terminates cleanly from s, or

(2) α sometimes terminates cleanly from s and

α sometimes terminates messily from s, or

(3) α always terminates messily from s.

Properties of the final states of programs (as will be seen in Chapter 3) are important when determining whether a program behaves as it was intended.

D **Final state properties** (from any initial state)

(1) Every final state of α from s has property Q, or

(2) some final state of α from s has property Q and

some final state of α from s does not have property Q, or

(3) No final state of α from s has property Q.

Next, embedding C(1), C(2) and C(3) in B we can say the following of a program α started in a state s.

E Clean-, Messy- and Nontermination Properties

Under the assumption that α starts from initial state s, we have the seven possibilites:

Either B(1)	α always terminates from s, which means:
	either C(1) α always terminates cleanly from s, or C(2) α sometimes terminates cleanly from s and α sometimes terminates messily from s, or C(3) α always terminates messily from s.
or B(2)	α sometimes terminates from s and α sometimes does not terminate from s, which means:
	α sometimes terminates from s and α sometimes does not terminate from s, and of the terminating executions of α from s, either C(1) α always terminates cleanly from s, or C(2) α sometimes terminates cleanly from s and α sometimes terminates messily from s, or C(3) α always terminates messily from s.
or B(3)	α never terminates from s.

So we get seven properties for a program α started in an initial state s, namely

B(1) C(1), B(1) C(2), B(1) C(3), B(2) C(1), B(2) C(2), B(2) C(3), B(3).

Now embedding these seven execution properties in A, we can say the following of a program α and an intial state s:

F **Execution properties**

A(1)	α always starts from s, which means:
either B(1)	α always terminates from s, that is:
	either C(1) α always terminates cleanly from s, or C(2) α sometimes terminates cleanly from s and α sometimes terminates messily from s, or C(3) α always terminates messily from s.
or B(2)	α sometimes terminates from s and α sometimes does not terminate from s, that is:
	α sometimes terminates from s and α sometimes does not terminate from s, and of the terminating executions of α from s, either C(1) α always terminates cleanly from s, or C(2) α sometimes terminates cleanly from s and α sometimes terminates messily from s, or C(3) α always terminates messily from s.
or B(3)	α never terminates from s.

A(2)	α sometimes starts from s and α sometimes does not start from s, which means:
	α sometimes starts from s and α sometimes does not start from s and once started from s
either B(1)	α always terminates from s, that is:
	either C(1) α always terminates cleanly from s, or C(2) α sometimes terminates cleanly from s and α sometimes terminates messily from s, or C(3) α always terminates messily from s.

or B(2)	α sometimes terminates from s and α sometimes does not terminate, from s, which means:
	α sometimes terminates from s and α sometimes does not terminate from s, and of the terminating executions of α from s, either C(1) α always terminates cleanly from s, or C(2) α sometimes terminates cleanly from s and α sometimes terminates messily from s, or C(3) α always terminates messily from s.
or B(3)	α never terminates from s.
A(3)	α never starts from s.

So we get fifteen execution properties for a program α and an initial state s. As a reference these are listed in **Table 1.**

Execution Property (i):	A(1) B(1) C(1)
Execution Property (ii):	A(1) B(1) C(2)
Execution Property (iii):	A(1) B(1) C(3)
Execution Property (iv):	A(1) B(2) C(1)
Execution Property (v):	A(1) B(2) C(2)
Execution Property (vi):	A(1) B(2) C(3)
Execution Property (vii):	A(1) B(3)
Execution Property (viii):	A(2) B(1) C(1)
Execution Property (ix):	A(2) B(1) C(2)
Execution Property (x):	A(2) B(1) C(3)
Execution Property (xi):	A(2) B(2) C(1)
Execution Property (xii):	A(2) B(2) C(2)
Execution Property (xiii):	A(2) B(2) C(3)
Execution Property (xiv):	A(2) B(3)
Execution Property (xv):	A(3)

Table 1: Execution Properties

Are these the only possible properties? To check this I use some elementary combinatorics. First consider a program α for which there is *at most one* outcome from each initial state. The logical possibilities of executing α from a state s are:

either (a) α does not start from s,
or (b) α starts and terminates cleanly from s,
or (c) α starts and terminates messily from s,
or (d) α starts and does not terminate from s.

Now consider a program α which may have *several* possible outputs for some of its inputs. The logical possibilities of executing α from a state s are obtained by taking all possible combinations of (a) - (d). So the number of logical possibilities is:

$$\binom{4}{0} + \binom{4}{1} + \binom{4}{2} + \binom{4}{3} + \binom{4}{4} = 1 + 4 + 6 + 4 + 1 = 16$$

The first possibility arises from not taking any of the four properties, which corresponds to not *activating* α in state s at all. In my enumeration of all the execution properties of a program α and a state s I assume α is activated in the state s. Accordingly, exactly fifteen possibilities remain, as indicated above.

I now come to the second part of this chapter. Here I consider the choices that must be made in modelling programs. First it must be decided which initialisation properties (A(1), A(2), A(3)) of a program α from a state s we wish to represent in a model. Using elementary combinatorics we see there are:

$$\binom{3}{0} + \binom{3}{1} + \binom{3}{2} + \binom{3}{3} = 1 + 3 + 3 + 1 = 8$$

possibilities. Namely,

either		None are represented.
or	A(1)	Only executions which always start are represented.
or	A(2)	Only executions which sometimes start are represented.
or	A(3)	Only executions which never start are represented.
or	[A(1) and A(2)]	All executions which sometimes or always starts are represented.
or	[A(1) and A(3)]	All executions which always start or which never start are represented.
or	[A(2) and A(3)]	All executions which sometimes or never start are represented.
or	[A(1), A(2) and A(3)]	All executions are represented.

Of these possibilities only three are interesting: A(1) , [A(1) and A(2)], and [A(1), A(2) and A(3)]. So we could choose:

R1 Initialisation Property Representations

either (1) Only states from which α *always* starts are included as initial states. (That is, a state from which α sometimes or always does not start is disregarded as an initial state.)

or (2) All states from which α sometimes or always starts are included as initial states. (That is, only states from which α *never* starts are disregarded as initial states.)

or (3) All states in which α is activated are included as initial states. (That is, no states are disregarded as initial states.)

I will now consider the relationship(s) (if any) between these three choices. My analysis is based on the following criterion:

> Option (n) in **R1** is *subsumed* under another option (m) in **R1** if the set of initialisation properties that can be represented when option (n) is chosen is contained in the set of initialisation properties that can be represented when option (m) is chosen.

The subsumption relationship between the above three choices is given in **Figure 1**,

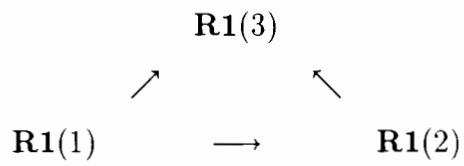


Figure 1

where an arrow means 'is subsumed by'. For example, an arrow from **R1**(2) to **R1**(3) means 'option **R1**(2) is subsumed by option **R1**(3)' interpreted as 'the set of initialisation properties that can be represented when option **R1**(2) is chosen is contained in the set of initialisation properties that can be represented when option **R1**(3)' is chosen.

Second, it must be decided which kind(s) of execution properties for a program started from a state are to be represented directly in a model. Namely, we could choose from the list (E) of clean-, messy- and nontermination properties (on p 4):

either	B(1) C(1)	Only cleanly terminating executions are represented.
or	B(1) C(2)	Only terminating executions are represented, but these may include both cleanly- and messily terminating executions.
or	B(1) C(3)	Only messily terminating executions are repesented.
or	B(2) C(1)	Only cleanly- and nonterminating executions are represented.
or	B(2) C(2)	All kinds of executions are represented.
or	B(2) C(3)	Only messily- and nonterminating executions are represented.
or	B(3)	Only nonterminating executions are represented.

The choices B(1) C(3), B(2) C(3) and B(3) are not interesting. So the number of choices reduces to four. Namely:

R2 Execution Property Representations

either (1) Only cleanly terminating executions are represented.

or (2) Only terminating executions are represented, but these may include both clean- and messy termination.

or (3) Only cleanly terminating and nonterminating executions are represented.

or (4) All kinds of executions are represented.

I will now discuss ways of including/excluding states as initial states when respectively option **R2**(1), **R2**(2), **R2**(3) or **R2**(4) is chosen.

- For option **R2**(1) there is no direct representation for messily terminating and for nonterminating executions. To represent terminating executions (where 'terminating' now means 'cleanly terminating') we could choose:

either (a) Only states from which a program *always* terminates are included as initial states. (That is, a state from which a program sometimes or always does not terminate is disregarded as an initial state.)

or (b) All states from which a program sometimes or always terminates are included as initial states. (That is, only states from which a program *never* terminates are disregarded as initial states.)

- For option **R2**(2) there is no direct representation for nonterminating executions. There are two possibilities for representing terminating executions. They are the same as above, except that 'terminating' now means 'either cleanly or messily terminating.'

- For option **R2**(3) there is no direct representation for messily terminating executions. There are two possibilities for representing cleanly terminating and nonterminating executions. Namely, we could choose:

either (a) Only states from which a program *always* terminates cleanly and/or does not terminate are included as initial states. (That is, a state from which a program sometimes or always terminates messily is disregarded as an initial state.)

or (b) All states from which a program sometimes or always terminates cleanly and/or does not terminate are included as initial states. (That is, only states from which a program *always* terminates messily are disregarded as initial states.)

- For option **R2**(4) all the execution properties for a program which starts from a state can be represented. For this there are two approaches. First, nontermination could be equated with messy termination (as in for example, Dijkstra ([1975], [1976]), Gries [1981], Dijkstra and Scholten [1990]). For this approach, the convention is to introduce a special symbol '$\perp$' (called *bottom*) to denote the final state of such executions. This is not always a good idea because nontermination may be the normal behaviour pattern of a program. Also there may be other ways in which a program can go wrong, for example, 'overflow', 'underflow' (due to a value being out of range), 'break' (due to a deliberate break in program execution), 'undefinedness' (due to say division by zero), etc. If these possibilities are taken into account equating nontermination with messy termination is not adequate — a direct representation is required respectively for messily terminating executions and nonterminating executions. The second approach, therefore, is to distinguish explicitly between nontermination and messy termination. So the two approaches are that:

either (a) one special representation simultaneously for messily terminating executions and nonterminating executions could be introduced,

or (b) two special representations respectively for messily terminating executions and nonterminating executions could be introduced.

So there are eight choices for the kind(s) of executions to be represented in a model of programs, namely

R2(1)(a), **R2**(1)(b), **R2**(2)(a), **R2**(2)(b), **R2**(3)(a), **R2**(3)(b), **R2**(4)(a), **R2**(4)(b).

I will now consider the relationship(s) (if any) between these eight choices. My analysis is based on the following criterion:

> Option (n) in **R2** is *subsumed* under another option (m) in **R2** if the set of execution properties that can be represented when option (n) is chosen is contained in the set of execution properties that can be represented when option (m) is chosen.

Table 2 indicates whether or not the various execution properties in **Table 1** can be represented when the options in **R2** are chosen. The rows refer to the execution properties p (where p = (i) - (xv)); the columns refer to the eight options **R2**$(i)(x)$ (where $i = 1, \ldots, 4$; x = a, b) for executions. I use the following notation:

> '$\surd$' in row 'p' under column '**R2**$(i)(x)$' means execution property p can be represented when option **R2**$(i)(x)$ is chosen.
> '$\times$' in row 'p' under column '**R2**$(i)(x)$' means execution property p cannot be represented when option **R2**$(i)(x)$ is chosen.

For example, execution property (ii) can be explicitly represented when option **R2**(1)(b) is chosen; but not when option **R2**(1)(a) is chosen.

	R2(1)		**R2**(2)		**R2**(3)		**R2**(4)	
	(a)	(b)	(a)	(b)	(a)	(b)	(a)	(b)
(i)	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$
(ii)	$\times$	$\surd$	$\surd$	$\surd$	$\times$	$\surd$	$\surd$	$\surd$
(iii)	$\times$	$\times$	$\surd$	$\surd$	$\times$	$\times$	$\surd$	$\surd$
(iv)	$\times$	$\surd$	$\times$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$
(v)	$\times$	$\surd$	$\times$	$\surd$	$\times$	$\surd$	$\surd$	$\surd$
(vi)	$\times$	$\times$	$\times$	$\surd$	$\times$	$\times$	$\surd$	$\surd$
(vii)	$\times$	$\times$	$\times$	$\times$	$\surd$	$\surd$	$\surd$	$\surd$
(viii)	$\times$	$\surd$	$\times$	$\surd$	$\times$	$\surd$	$\surd$	$\surd$
(ix)	$\times$	$\surd$	$\times$	$\surd$	$\times$	$\surd$	$\surd$	$\surd$
(x)	$\times$	$\times$	$\times$	$\surd$	$\times$	$\times$	$\surd$	$\surd$
(xi)	$\times$	$\surd$	$\times$	$\surd$	$\times$	$\surd$	$\surd$	$\surd$
(xii)	$\times$	$\surd$	$\times$	$\surd$	$\times$	$\surd$	$\surd$	$\surd$
(xiii)	$\times$	$\times$	$\times$	$\surd$	$\times$	$\times$	$\surd$	$\surd$
(xiv)	$\times$	$\times$	$\times$	$\times$	$\times$	$\surd$	$\surd$	$\surd$
(xv)	$\times$	$\times$	$\times$	$\times$	$\times$	$\times$	$\surd$	$\surd$

Table 2: Execution Properties and Execution Property Representations

The subsumption relationship between the above eight choices is given in **Figure 2**,

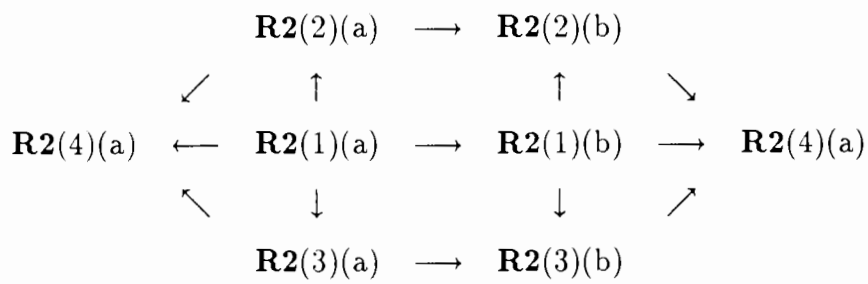


Figure 2

where an arrow means 'is subsumed under'. For example, an arrow from option **R2**(3)(a) to **R2**(3)(b) means 'option **R2**(3)(a) is subsumed under option **R2**(3)(b)' interpreted as 'the set of execution properties that can be represented when option **R2**(3)(a) is chosen is contained in the set of execution properties that can be represented when option **R2**(3)(b) is chosen'. (Note that I have only used **R2**(4)(a) in **Figure 2** because **R2**(4)(a) subsumes **R2**(4)(b) and conversely .)

The three choices (**R1**(1), **R1**(2), **R1**(3) (on p 8)) for the set of initial states and the eight choices (**R2**(i)(x) (where i = 1, 2, 3, 4 and x = a, b) (on p 11)) for executions lead to twenty-four different representation methods for programs. As a reference these are listed in **Table 3**.

Representation Method $\mathbf{R_j(1)}$:	**R1**(j) and **R2**(1)(a)
Representation Method $\mathbf{R_j(2)}$:	**R1**(j) and **R2**(1)(b)
Representation Method $\mathbf{R_j(3)}$:	**R1**(j) and **R2**(2)(a)
Representation Method $\mathbf{R_j(4)}$:	**R1**(j) and **R2**(2)(b)
Representation Method $\mathbf{R_j(5)}$:	**R1**(j) and **R2**(3)(a)
Representation Method $\mathbf{R_j(6)}$:	**R1**(j) and **R2**(3)(b)
Representation Method $\mathbf{R_j(7)}$:	**R1**(j) and **R2**(4)(a)
Representation Method $\mathbf{R_j(8)}$:	**R1**(j) and **R2**(4)(b)

Table 3: Representation Methods

where $\mathbf{j} = 1, 2, 3$.

Note that in $\mathbf{R_j(n)}$, that is, $\mathbf{R1}(\mathrm{j})$ and $\mathbf{R2}(i)(x)$ (where $\mathbf{j} = 1, 2, 3$; $\mathbf{n} = 1, 2, \ldots, 8$; $i = 1, 2, 3, 4$; $x = \mathrm{a}, \mathrm{b}$), the conjunct '$\mathbf{R1}(\mathrm{j})$ and $\mathbf{R2}(i)(x)$' means the set of states included as initial states when representation method $\mathbf{R_j(n)}$ is chosen is simply the intersection of the set of states included as initial states when $\mathbf{R1}(\mathrm{j})$ is chosen and the set of states included as initial states when $\mathbf{R2}(i)(x)$ is chosen. (That is, the set of states disregarded as initial states when representation method $\mathbf{R_j(n)}$ is chosen is simply the union of the set of states disregarded as initial states when $\mathbf{R1}(\mathrm{j})$ is chosen and the set of states disregarded as initial states when $\mathbf{R2}(i)(x)$ is chosen.) For example, in $\mathbf{R_1(1)}$ the conjunct '$\mathbf{R1}(1)$ and $\mathbf{R2}(1)(\mathrm{a})$' means 'Only states from which a program *always* starts and *always* terminates cleanly are included as initial states'.

I now investigate the connection(s) (if any) between the various representation methods. I need a notation for the representation with respect to a representation method of an execution property of a program α from a state s.

> For any representation method $\mathbf{R_j(n)}$ (where $\mathbf{j} = 1, 2, 3$; $\mathbf{n} = 1, 2, \ldots, 8$) and any execution property p (where $p =$ (i) - (xv)) of a program α from any state s, $exrep_{\mathbf{R_j(n)}}(\alpha, s)$ denotes the representation with respect to $\mathbf{R_j(n)}$ of the execution property p of α from s. If there is no direct representation with respect to $\mathbf{R_j(n)}$ of the execution property p then $exrep_{\mathbf{R_j(n)}}(\alpha, s) = \emptyset$.

For example, if (as will be done in Chapter 4) input-output pairs of states are used to represent executions of programs then $exrep_{\mathbf{R_j(n)}}(\alpha, s)$ could denote a set of pair(s) of states with first component s. If (as will be done in Chapter 6) execution sequences are used to represent executions of programs then $exrep_{\mathbf{R_j(n)}}(\alpha, s)$ could denote a set of execution sequences of states with first component s. In this chapter I use a pictorial representation for executions of programs. As a reference, the symbols used in this chapter are collected together in **Table 4**.

> An execution property p (where $p =$ (i) - (xv)) of a program α from a state s will

be called *representable* with respect to a representation method $\mathbf{R_j(n)}$ (where $\mathbf{j}$ = 1, 2, 3; $\mathbf{n}$ = 1, 2, ..., 8) iff $exrep_{\mathbf{R_j(n)}}(\alpha, s) \neq \emptyset$

'• →'	means that	'α always terminates cleanly from state s'
'• → ...'	means that	'α never terminates from state s'
'• ↯'	means that	'α always terminates messily from state s'
'• ↗↯'	means that	'α sometimes terminates cleanly and sometimes terminates messily from state s'
'• ↗ ...'	means that	'α sometimes terminates cleanly and sometimes does not terminate from state s'
'• ↘...↯'	means that	'α sometimes terminates messily and sometimes does not terminate from state s'
'• ↗↘...↯'	means that	'α sometimes terminates cleanly, sometimes terminates messily and sometimes does not terminate from state s'
'•'	means that	'α never starts from state s'
a space	means that	'there is no direct representation for the execution of α from s'

Table 4: Table of Pictorial Representations

Table 5 indicates whether or not an execution property p (where p = (i) - (xv) in **Table 1**) is representable with respect to a representation method $\mathbf{R_j(n)}$ (where $\mathbf{j}$ = 1, 2, 3; $\mathbf{n}$ = 1, 2, ..., 8 in **Table 3**). If p is representable it also illustrates the pictorial representation of p with respect to the representation method $\mathbf{R_j(n)}$. In **Table 5** the verticals refer to the twenty-four representation methods ($\mathbf{R_j(n)}$ (where $\mathbf{j}$ = 1, 2, 3; $\mathbf{n}$ = 1, 2, ..., 8) in **Table 3**); while the horizontals refer to the fifteen execution properties ((i) - (xv) in **Table 1**).

A blank entry in a row labelled p (for p = (i) - (xv)) under a column labelled $\mathbf{R_j(n)}$ (for $\mathbf{n}$ = 1, 2, ..., 8) indicates that execution property p is not representable with respect to representation method $\mathbf{R_j(n)}$.

A nonblank entry in a row labelled p (for p = (i) - (xv)) under a column labelled $\mathbf{R_j(n)}$ (for $\mathbf{n} = 1, 2, \ldots, 8$) indicates that p is representable with respect to $\mathbf{R_j(n)}$ and illustrates the pictorial representation of p with respect to $\mathbf{R_j(n)}$.

I now explain what each possible entry means. Consider a program α with execution property p from some state s. I briefly explain what an entry in row p under column $\mathbf{R_j(n)}$ (where $\mathbf{j} = 1, 2, 3$; $\mathbf{n} = 1, 2, \ldots, 8$) means. An entry of the form:

'• →'	means that	using $\mathbf{R_j(n)}$ p is represented as an execution which always terminates cleanly from s
'• → ...'	means that	using $\mathbf{R_j(n)}$ p is represented as an execution which never terminates from s
'• ↯'	means that	using $\mathbf{R_j(n)}$ p is represented as an execution which always terminates messily from s
'•↗↯'	means that	using $\mathbf{R_j(n)}$ p is represented as an execution which sometimes terminates cleanly and sometimes terminates messily from state s
'•↗→ ...'	means that	using $\mathbf{R_j(n)}$ p is represented as an execution which sometimes terminates cleanly and sometimes does not terminate from state s
'•↖...↯'	means that	using $\mathbf{R_j(n)}$ p is represented as an execution which sometimes terminates messily and sometimes does not terminate from state s
'•↗↖...↯'	means that	using $\mathbf{R_j(n)}$ p is represented as an execution which sometimes terminates cleanly, sometimes terminates messily and sometimes does not terminate from state s
'•'	means that	using $\mathbf{R_j(n)}$ p is represented as an execution which never starts from state s
a space	means that	using $\mathbf{R_j(n)}$ there is no direct representation for p

An item in parentheses refers to selected values of $\mathbf{j}$. The representations in parentheses for execution properties (viii) - (xiv) mean that for a program

α with any such execution property from a state s $exrep_{\mathbf{R_1(n)}}(\alpha, s) = \emptyset$ but $exrep_{\mathbf{R_j(n)}}(\alpha, s)$ (for $\mathbf{j} = 2, 3$) is given by the corresponding entry in **Table 5**. The representation in parentheses for execution property (xv) means $exrep_{\mathbf{R_3(n)}}(\alpha, s) is \bullet$ but $exrep_{\mathbf{R_j(n)}}(\alpha, s) = \emptyset$ (for $\mathbf{j} = 1, 2$).

So

an execution property p of α from s is representable with respect to a representation method $\mathbf{R_j(n)}$ iff there is a nonblank entry in row p under column $\mathbf{R_j(n)}$, iff $exrep_{\mathbf{R_j(n)}}(\alpha, s) \neq \emptyset$.

	$\mathbf{R_j(1)}$	$\mathbf{R_j(2)}$	$\mathbf{R_j(3)}$	$\mathbf{R_j(4)}$	$\mathbf{R_j(5)}$	$\mathbf{R_j(6)}$	$\mathbf{R_j(7)}$	$\mathbf{R_j(8)}$
(i)	$\bullet \rightarrow$	$\bullet \rightarrow$	$\bullet \rightarrow$	$\bullet \rightarrow$	$\bullet \rightarrow$	$\bullet \rightarrow$	$\bullet \rightarrow$	$\bullet \rightarrow$
(ii)		$\bullet \rightarrow$	$\bullet$	$\bullet$		$\bullet \rightarrow$	$\bullet$	$\bullet$
(iii)			$\bullet$	$\bullet$			$\bullet$	$\bullet$
(iv)		$\bullet \rightarrow$		$\bullet \rightarrow$	$\bullet \rightarrow \ldots$	$\bullet \rightarrow \ldots$	$\bullet$	$\bullet \rightarrow \ldots$
(v)		$\bullet \rightarrow$		$\bullet$		$\bullet \rightarrow \ldots$	$\bullet$	$\bullet$
(vi)				$\bullet$		$\bullet \rightarrow \ldots$	$\bullet$	$\bullet$
(vii)					$\bullet \rightarrow \ldots$	$\bullet \rightarrow \ldots$	$\bullet$	$\bullet \rightarrow \ldots$
(viii)		$(\bullet \rightarrow)$		$(\bullet \rightarrow)$		$(\bullet \rightarrow)$	$(\bullet \rightarrow)$	$(\bullet \rightarrow)$
(ix)		$(\bullet \rightarrow)$		$(\bullet$		$(\bullet \rightarrow)$	$(\bullet$	$(\bullet$
(x)				$(\bullet$			$(\bullet$	$(\bullet$
(xi)		$(\bullet \rightarrow)$		$(\bullet \rightarrow)$		$(\bullet \rightarrow \ldots)$	$(\bullet$	$(\bullet \rightarrow \ldots)$
(xii)		$(\bullet \rightarrow)$		$(\bullet$		$(\bullet \rightarrow \ldots)$	$(\bullet$	$(\bullet$
(xiii)				$(\bullet$		$(\bullet \rightarrow \ldots)$	$(\bullet$	$(\bullet$
(xiv)						$(\bullet \rightarrow \ldots)$	$(\bullet$	$(\bullet \rightarrow \ldots)$
(xv)							$(\bullet$	$(\bullet)$

Table 5: Execution Properties and Representation Methods

The information in **Table 5** elaborates on the information in **Table 2** as follows. Consider any representation method $\mathbf{R_j(n)}$: **R1**(j) and **R2**$(i)(x)$ (where $\mathbf{j} = 1, 2, 3$; $\mathbf{n} = 1, 2, \ldots, 8$;

i = 1, 2, 3, 4; x = a, b) and an execution property p (where p = (i) - (xv)) of a program α from a state s. Then

> p is representable with respect to $\mathbf{R_j(n)}$ iff s is included as an initial state when option **R1**(j) is chosen and p can be represented when option $\mathbf{R2}(i)(x)$ is chosen.

That is, $exrep_{\mathbf{R_j(n)}}(\alpha, s) \neq \emptyset$ iff s is included as an initial state when option **R1**(j) is chosen and there is a '$\surd$' in row 'p' under column $\mathbf{R2}(i)(x)$ in **Table 2**. Therefore $exrep_{\mathbf{R_j(n)}}(\alpha, s) = \emptyset$ iff s is not included as an initial state when option **R1**(j) is chosen or there is a '$\times$' in row 'p' under column $\mathbf{R2}(i)(x)$ in **Table 2**.

For example, execution property (ii) of a program α from a state s is representable with respect to $\mathbf{R_1(2)}$ (because α always starts from s and there is a '$\surd$' in row (ii) under column **R2(1)(b)** in **Table 2**) but not with respect to $\mathbf{R_1(1)}$ (because there is a '$\times$' in row (ii) under column **R2**(1)(a) in **Table 2**).

Also execution property (ix) of a program α from a state s is representable with respect to $\mathbf{R_j(2)}$ (where **j** = 2, 3) (because s is included as an initial state when **R1**(2) is chosen and there is a '$\surd$' in row (ix) under column **R2**(1)(b) in **Table 1**. This property is not representable with respect to $\mathbf{R_1(2)}$ (because only states from which programs always start are included as initial state when option **R1**(1) is chosen).

Table 5 provides the following information about execution properties and representation methods.

- The representation methods with respect to which a particular execution property p (where p = (i) - (xv)) is representable can be found by examining the nonblank entries in the row labelled p. For example, execution property (ii) is representable with respect to $\mathbf{R_j(n)}$ (for **j** = 1, 2, 3; **n** = 2, 3, 4, 6, 7, 8) but not with respect to $\mathbf{R_j(n)}$ (for **j** = 1, 2, 3; **n** = 1, 5)

- The execution properties representable with respect to a particular representation method $\mathbf{R_j(n)}$ (where **j** = 1, 2, 3; **n** = 1, 2, ... 8) can be found by examining the nonblank entries in the column labelled $\mathbf{R_j(n)}$. For example, when $\mathbf{R_1(2)}$ is chosen

execution properties (i), (ii), (iv), (v) can be represented; when $\mathbf{R_j(2)}$ (for $\mathbf{j} = 2, 3$) is chosen execution properties (i), (ii), (iv), (v) as well as (viii), (ix), (xi), (xii) can be represented.

- The execution properties distinguishable with respect to a particular representation method $\mathbf{R_j(n)}$ (where $\mathbf{j} = 1, 2, 3$; $\mathbf{n} = 1, 2, \ldots, 8$) can be found by comparing the entries in the column labelled $\mathbf{R_j(n)}$ and grouping together the execution properties with the same pictorial representation into equivalence classes. Recall $exrep_{\mathbf{R_j(n)}}(\alpha, s)$ denotes an entry in **Table 5**. This means we can examine the equivalence classes generated by $exrep_{\mathbf{R_j(n)}}$. For a representation method $\mathbf{R_j(n)}$, let $[exrep_{\mathbf{R_j(n)}}]$ denote the set of equivalence classes generated by $exrep_{\mathbf{R_j(n)}}$. For example, $[exrep_{\mathbf{R_3(n)}}]$ (for $\mathbf{n} = 1, 2, \ldots, 8$) is obtained as follows:

  In column $\mathbf{R_3(1)}$ there are two distinct entries; hence two equivalence classes. So

  $[exrep_{\mathbf{R_3(1)}}]$

  = {[(i)], [(ii),(iii),(iv),(v),(vi),(vii),(viii),(ix),(x),(xi),(xii),(xiii),(xiv),(xv)]}

  Likewise

  $[exrep_{\mathbf{R_3(2)}}]$

  = {[(i),(ii),(iv),(v),(viii),(ix),(xi),(xii)], [(iii),(vi),(vii),(x),(xiii),(xiv),(xv)]}

  $[exrep_{\mathbf{R_3(3)}}]$

  = {[(i)], [(ii)], [(iii)], [(iv),(v),(vi),(vii),(viii),(ix),(x),(xi),(xii),(xiii),(xiv),(xv)]}

  $[exrep_{\mathbf{R_3(4)}}]$

  = {[(i),(iv),(viii),(xi)], [(ii),(v),(ix),(xii)], [(iii),(vi),(x),(xiii)], [(vii),(xiv),(xv)]}

  $[exrep_{\mathbf{R_3(5)}}]$

  = {[(i)], [(iv)], [(vii)], [(ii),(iii),(v),(vi),(viii),(ix),(x),(xi),(xii),(xiii),(xiv),(xv)]}

  $[exrep_{\mathbf{R_3(6)}}]$

  = {[(i),(ii),(viii),(ix)], [(iv),(v),(xi),(xii)], [(vi),(vii),(xiii),(xiv)], [(iii),(x),(xv)]}

$[exrep_{\mathbf{R_3(7)}}]$
= {[(i),(viii)], [(ii),(iv),(v),(ix),(xi),(xii)], [(iii), (vi),(vii),(x),(xiii),(xiv),(xv)]}

$[exrep_{\mathbf{R_3(8)}}]$
= {[(i),(viii)],[(ii),(ix)],[(iii),(x)], [(iv),(xi)],[(v),(xii)],[(vi),(xiii)],[(vii),(xiv)],[(xv)]}

As a reference in Theorem 1 the relationships between these eight sets $[exrep_{\mathbf{R_3(n)}}]$ (for $\mathbf{n} = 1, 2, \ldots, 8$) of equivalence classes are given in **Figure 3**,

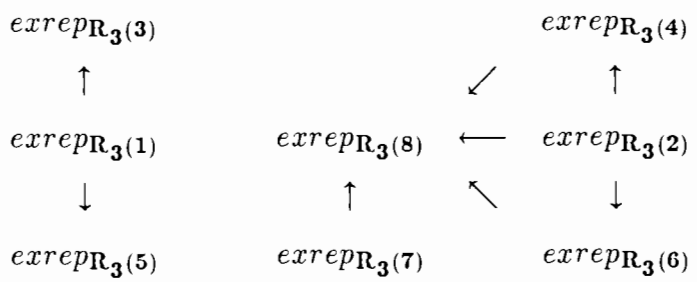


Figure 3

where an arrow means 'is coarser than'. For example, an arrow from $exrep_{\mathbf{R_3(4)}}$ to $exrep_{\mathbf{R_3(8)}}$ means '$exrep_{\mathbf{R_3(4)}}$ is coarser than $exrep_{\mathbf{R_3(8)}}$ interpreted as 'the equivalence relation induced by $exrep_{\mathbf{R_3(8)}}$ is contained in the equivalence relation induced by $exrep_{\mathbf{R_3(4)}}$'.

- The representations distinguishable with respect to a particular execution property p (where p = (i) - (xv)) can be found by comparing the entries in the row labelled p and grouping the representation methods with the same pictorial representations of p into equivalence classes. For example, for execution property (ii) there are three different pictorial representations in row (ii) and hence three equivalence classes. Namely:

$$[\mathbf{R_j(1)}, \mathbf{R_j(5)}]; \quad [\mathbf{R_j(2)}, \mathbf{R_j(6)}]; \quad [\mathbf{R_j(3)}, \mathbf{R_j(4)}, \mathbf{R_j(7)}, \mathbf{R_j(8)}]$$

My investigation of the connections between the representation methods is based on the following criterion.

A representation method (n) *is subsumed under* a representation method (m) if

(i) the set of executions representable with respect to (n) is contained in the set of executions representable with respect to (m),

and (ii) the representable executions distinguishable with respect to (n) are distinguishable with respect to (m) (or equivalently, the representable executions indistinguishable with respect to (m) are indistinguishable with respect to (n)).

Therefore a representation method (n) *is subsumed under* a representation method (m) if the equivalence relation induced by $exrep_m$ is contained in the equivalence relation induced by $exrep_n$ (that is, if $exrep_m$ is *finer* than $exrep_n$ (or if $exrep_n$ is *coarser* than $exrep_m$)).

A representation method (n) is not subsumed under a representation method (m) iff

either (i) not every program representable with respect to (n) is representable with respect to (m),

or (ii) $exrep_n$ is not *coarser* than $exrep_m$

I will call two representation methods *unrelated* if neither is subsumed under the other.

(1) **Theorem**

(a) $\mathbf{R_j(1)}$ *is subsumed under* $\mathbf{R_j(3)}$ (*for* $\mathbf{j} = 1, 2, 3$).

(b) $\mathbf{R_j(2)}$ *is subsumed under* $\mathbf{R_j(4)}$ (*for* $\mathbf{j} = 1, 2, 3$).

(c) $\mathbf{R_j(1)}$ *is subsumed under* $\mathbf{R_j(5)}$ (*for* $\mathbf{j} = 1, 2, 3$).

(d) $\mathbf{R_j(2)}$ *is subsumed under* $\mathbf{R_j(6)}$ (*for* $\mathbf{j} = 1, 2, 3$).

(e) $\mathbf{R_j(n)}$ *is subsumed under* $\mathbf{R_j(8)}$ (*for* $\mathbf{j} = 1, 2, 3$; *for* $\mathbf{n} = 2, 4, 6, 7$)

(f) $\mathbf{R_1(n)}$ *is subsumed under* $\mathbf{R_2(n)}$ (*for* $\mathbf{n} = 1, 2, \ldots, 8$).

(g) $\mathbf{R_2(n)}$ *is subsumed under* $\mathbf{R_3(n)}$ (*for* $\mathbf{n} = 1, 2, \ldots, 8$).

(h) $\mathbf{R_1(n)}$ *is subsumed under* $\mathbf{R_3(n)}$ (*for* $\mathbf{n} = 1, 2, \ldots, 8$).

But the converses of (a) - (h) do not hold.

Proof I prove (a) - (e) for $\mathbf{j} = 3$; the other cases are similar. Recall (from page 14) that the set of execution properties representable with respect to $\mathbf{R_3(n)}$ (for $\mathbf{n} = 1, 2, \ldots, 8$) is the same as those that can be represented when the option in **R2** is chosen. So the subsumption relationships (in **Figure 2**) between the options in **R2** can be used to determine whether condition (i) for subsumption of representation methods holds or not. Recall also from **Figure 3** the sets $[exrep_{\mathbf{R_3(n)}}]$ (for $\mathbf{n} = 1, 2, \ldots, 8$) of equivalences classes and the relationships between them.

(a) Since **R2**(1)(a) is subsumed under **R2**(2)(a) and $exrep_{\mathbf{R_3(1)}}$ is coarser than $exrep_{\mathbf{R_3(3)}}$, $\mathbf{R_1(1)}$ is subsumed under $\mathbf{R_3(3)}$. But $exrep_{\mathbf{R_3(3)}}$ is not coarser than $exrep_{\mathbf{R_3(1)}}$, so $\mathbf{R_3(3)}$ is not subsumed under $\mathbf{R_3(1)}$.

(b) Since **R2**(1)(b) is subsumed under **R2**(2)(b) and $exrep_{\mathbf{R_3(2)}}$ is coarser than $exrep_{\mathbf{R_3(4)}}$, $\mathbf{R_3(2)}$ is subsumed under $\mathbf{R_3(4)}$. But $exrep_{\mathbf{R_3(4)}}$ is not coarser than $exrep_{\mathbf{R_3(2)}}$, so $\mathbf{R_3(4)}$ is not subsumed under $\mathbf{R_3(2)}$.

(c) Since **R2**(1)(a) is subsumed under **R2**(3)(a) and $exrep_{\mathbf{R_3(1)}}$ is coarser than $exrep_{\mathbf{R_3(5)}}$, $\mathbf{R_3(1)}$ is subsumed under $\mathbf{R_3(5)}$. But $exrep_{\mathbf{R_3(5)}}$ is not coarser than $exrep_{\mathbf{R_3(1)}}$, so $\mathbf{R_3(5)}$ is not subsumed under $\mathbf{R_3(1)}$.

(d) Since **R2**(1)(b) is subsumed under **R2**(2)(b) and $exrep_{\mathbf{R_3(2)}}$ is coarser than $exrep_{\mathbf{R_3(6)}}$, $\mathbf{R_3(2)}$ is subsumed under $\mathbf{R_3(6)}$. But $exrep_{\mathbf{R_3(6)}}$ is not coarser than $exrep_{\mathbf{R_3(2)}}$, so $\mathbf{R_3(6)}$ is not subsumed under $\mathbf{R_3(2)}$.

(e) Since $\mathbf{R2}(i)$(a) (and $\mathbf{R2}(i)$(b)) (for $i = 1, 2, 3, 4$) is subsumed under **R2**(4)(a) and $exrep_{\mathbf{R_3(n)}}$ (for $\mathbf{n} = 2, 4, 6, 7$) is coarser than $exrep_{\mathbf{R_3(8)}}$, $\mathbf{R_3(n)}$ is subsumed under $\mathbf{R_3(8)}$ (for $\mathbf{n} = 2, 4, 6, 7$). But $exrep_{\mathbf{R_3(8)}}$ is not coarser than $exrep_{\mathbf{R_3(n)}}$ (for $\mathbf{n} = 1, 2, \ldots, 7$), so $\mathbf{R_3(8)}$ is not subsumed under $\mathbf{R_3(n)}$ (for $\mathbf{n} = 1, 2, \ldots, 7$).

(f) Since option **R1**(1) is subsumed under option **R1**(3) (**Figure 1**), $\mathbf{R_1(n)}$ is subsumed under $\mathbf{R_2(n)}$ (for $\mathbf{n} = 1, 2, \ldots, 8$). For the converse, consider a program

α which sometimes starts and sometimes terminates cleanly from some state s. Then for $\mathbf{n} = 2, 4, 6, 7$ or 8, $exrep_{\mathbf{R_1(n)}}(\alpha, s) = \emptyset$ but $exrep_{\mathbf{R_2(n)}}(\alpha, s) \neq \emptyset$ (since there are nonblank entries in the row labeled (viii) under the columns $\mathbf{R_2(2)}$, $\mathbf{R_2(4)}$, $\mathbf{R_2(6)}$, and $\mathbf{R_2(8)}$); hence not every execution property of a program representable using $\mathbf{R_2(n)}$ is representable using $\mathbf{R_1(n)}$ (for $\mathbf{n} = 2, 4, 6, 7$ or 8). $\mathbf{R_2(n)}$ is subsumed under $\mathbf{R_1(n)}$ (for $\mathbf{n} = 1, 3, 5$) (because the set of executions representable and distinguishable with respect to $\mathbf{R_2(n)}$ (for $\mathbf{n} = 1, 3, 5$) is the same as that for $\mathbf{R_1(n)}$).

(g) Since option **R1**(2) is subsumed under option **R1**(3) (**Figure 1**), $\mathbf{R_2(n)}$ is subsumed under $\mathbf{R_3(n)}$. For the converse, consider a program α which never starts from some state s. Then $exrep_{\mathbf{R_2(8)}}(\alpha, s) = \emptyset$ but $exrep_{\mathbf{R_3(8)}}(\alpha, s) \neq \emptyset$; hence not every program representable using $\mathbf{R_3(8)}$ is representable using $\mathbf{R_2(8)}$.

(h) Since $\mathbf{R_1(n)}$ is subsumed under $\mathbf{R_2(n)}$ (by (f)) and $\mathbf{R_2(n)}$ is subsumed under $\mathbf{R_3(n)}$ (by (g)) it follows that $\mathbf{R_1(n)}$ is subsumed under $\mathbf{R_3(n)}$. □

There are fourteen subsumption relationships in **Figure 2** between the options in **R2**. Here there are only eight. I now show that I have indeed covered all the cases.

- $\mathbf{R_3(1)}$ is not subsumed under $\mathbf{R_3(2)}$ because **R2**(1)(a) is subsumed under **R2**(1)(b) but $exrep_{\mathbf{R_3(1)}}$ is not coarser than $exrep_{\mathbf{R_3(2)}}$. (For example, from **Table 5** $\mathbf{R_3(1)}$ distinguishes between execution properties (i) and (ii) but $\mathbf{R_3(2)}$ does not.)

- $\mathbf{R_3(3)}$ is not subsumed under $\mathbf{R_3(4)}$ because **R2**(2)(a) is subsumed under **R2**(2)(b) but $exrep_{\mathbf{R_3(3)}}$ is not coarser than $exrep_{\mathbf{R_3(4)}}$. (For example, from **Table 5** $\mathbf{R_3(3)}$ distinguishes between execution properties (i) and (iv) but $\mathbf{R_3(4)}$ does not.)

- $\mathbf{R_3(5)}$ is not subsumed under $\mathbf{R_3(6)}$ because **R2**(3)(a) is subsumed under **R2**(3)(b) but $exrep_{\mathbf{R_3(5)}}$ is not coarser than $exrep_{\mathbf{R_3(6)}}$. (For example, from **Table 5** $\mathbf{R_3(5)}$ distinguishes between execution properties (i) and (ii) but $\mathbf{R_3(6)}$ does not.)

- $\mathbf{R_3(7)}$ is unrelated to $\mathbf{R_3(n)}$. First $\mathbf{R_3(n)}$ is not subsumed under $\mathbf{R_3(7)}$ because **R2**(i)(a) (for $i = 1, 2, 3$) is subsumed under **R2**(4)(a) but $exrep_{\mathbf{R_3(n)}}$ is not coarser

than $exrep_{\mathbf{R_3(7)}}$. (For example, from **Table 5** $\mathbf{R_3(5)}$ distinguishes between execution properties (iv) and (v) but $\mathbf{R_3(7)}$ does not.) Second $\mathbf{R_3(7)}$ is not subsumed under $\mathbf{R_3(n)}$ because $\mathbf{R2}(4)$(a) is not subsumed under $\mathbf{R2}(i)$(a) (for $i = 1, 2, 3$)

- $\mathbf{R_3(n)}$ (for $\mathbf{n} = 1, 3, 5$) is not subsumed under $\mathbf{R_3(8)}$ because $\mathbf{R2}(i)$(a) (for $i = 1, 2, 3$) is subsumed under $\mathbf{R2}(4)$(a) but $exrep_{\mathbf{R_3(n)}}$ (for $\mathbf{n} = 1, 3, 5$) is not coarser than $exrep_{\mathbf{R_3(8)}}$. (For example, from **Table 5** $\mathbf{R_3(1)}$ distinguishes between execution properties (i) and (viii) but $\mathbf{R_3(8)}$ does not.)

The subsumption relationship between the eight representation method $\mathbf{R_3(n)}$ (for $\mathbf{n} = 1, 2, \ldots, 8$) is given in **Figure 4**.

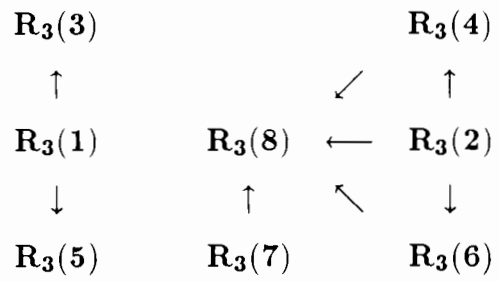


Figure 4

where an arrow means 'is subsumed under'. For example, an arrow from $\mathbf{R_3(4)}$ to $\mathbf{R_3(8)}$ means '$\mathbf{R_3(4)}$ is subsumed under $\mathbf{R_3(8)}$ interpreted as 'the set of execution properties representable and distinguishable with respect to $\mathbf{R_3(4)}$ is contained in the set of execution properties representable and distinguishable with respect to $\mathbf{R_3(8)}$'.

In conclusion I introduce four methods for executing nondeterministic programs. An execution of a nondeterministic program can be viewed operationally as a sequence of states, starting with an initial state and terminating (if at all) in a final state. I will call any such sequence an *execution sequence* (or *exseq* for short). Intuitively, we may think of an *execution tree* (or *extree* for short) of states developing from any initial state. Then any path from the root of such a tree represents a legal execution. In particular, nonterminating and messily terminating executions are captured naturally by the notion of an execution tree: an infinite

path represents the former; while a finite path with '$\perp$' as its leaf represents the latter. So if $\mathcal{S}$ is the state space, then cleanly terminating executions yield finite exseqs ending in some state $t \in \mathcal{S}$; messily terminating executions yield finite exseqs ending in a special symbol '$\perp$' ($\perp \notin \mathcal{S}$) and nonterminating executions yield infinite exseqs. An execution method for a nondeterministic program then corresponds to traversing the execution tree of the program in search of a final state (that is, a leaf of the tree). My objective here is to clarify and to define precisely four methods for executing nondeterministic programs by describing four algorithms for traversing the extree of a program. For this I need some set-theoretic notation. This is as follows:

First, I denote exseqs by $\mathbf{x}$, $\mathbf{y}$, $\mathbf{z}$, ..., etc. Second, I need a notation for the set of all exseqs of a program from an initial state. For any program α and any initial state s I use:

$$\mathbf{extree}(\alpha, s) = \{\mathbf{x} \mid \mathbf{x} \text{ is an exseq of } \alpha \text{ from } s\}$$

There are, in general, many tree traversal algorithms (for example, depth search, breadth search, left-first search, etc) and hence many execution methods for programs. Instead of attempting to present a complete list of such methods, I present in **Table 6** the four methods described in Harel ([1979a] p 68, 69) which as he points out 'represent fair methods in which no specific group of leaves is drastically favoured over others'. (Harel [1979b] presented mathematical characterisations of execution methods in terms of trees of programs of states.) Consider $\mathbf{extree}(\alpha, s)$ for some program α and some state s.

(a)	**Depth-first execution method** (1) Choose an arbitrary enumeration $\{\mathbf{x}_n\}$ of **extree**(α, s). (2) Execute some $\mathbf{x}_n$.
(b)	**Depth-first execution method with backtracking** (1) Choose an arbitrary enumeration $\{\mathbf{x}_n\}$ of **extree**(α, s). (2) Execute $\mathbf{x}_n$ (if none, execution terminates but α produces no result) until: either (i) $\perp$ is encountered in which case increment n by one, restore the original machine state s and repeat step (2), or (ii) execution of $\mathbf{x}_n$ terminates cleanly in some state t in which case α produces t as its result.
(c)	**Breadth-first execution method** (1) Choose an arbitrary enumeration $\{\mathbf{x}_n\}$ of **extree**(α, s). (2) Execute (concurrently) $\mathbf{x}_n$ until: either (i) $\perp$ is encountered for some n in which case execution of all the $\mathbf{x}_n$'s terminates but α produces no result, or (ii) some $\mathbf{x}_n$ terminates cleanly in some state t in which case execution of all the $\mathbf{x}_n$'s terminates and t is produced as result.
(d)	**Breadth-first execution method with backtracking** (1) Choose an arbitrary enumeration $\{\mathbf{x}_n\}$ of **extree**(α, s). (2) Execute (concurrently) $\mathbf{x}_n$ until: either (i) $\perp$ is encountered for some n in which case repeat step (2) but ignore $\mathbf{x}_n$, or (ii) some $\mathbf{x}_n$ terminates cleanly in some state t in which case execution of all the $\mathbf{x}_n$'s terminates and t is produced as result.

Table 6: Execution Methods

(Note that algorithms **(a)** and **(b)** are sequential methods because the exseqs in **extree**(α, s) are executed one at a time. The algorithms **(c)** and **(d)** are parallel (or concurrent) methods because the exseqs in **extree**(α, s) are executed simultaneously — the time taken to execute

α from s is no more than the time taken to execute the longest exseq in **extree**(α, s).)

To compare these four methods, consider the execution properties ((i) - (xv)) in **Table 1** of a program α from an initial state s. **Table 7** indicates the result (if any) produced by a program α activated in a state s using the various execution methods ((**a**) - (**d**)). A nonblank entry in the table indicates that α does something; a blank entry in the table indicates that α does nothing. The horizontals refer to the execution properties (i) - (xv) of a program α from a state s and the verticals refer to the execution methods (**a**) - (**d**). I use the pictorial representations described in **Table 4** — here 'terminates cleanly' is replaced by 'produces a state $t \in \mathcal{S}$ as result'.

Consider a progam α with an execution property p from some state s. An entry in **Table 7** in row p under column **x** (where **x** = (**a**), (**b**), (**c**), (**d**)) means the same as explained on p 16, only reading 'execution method' for 'representation method' and 'executed' for 'represented'.

For the present purposes it seems sufficient to voice my opinion that execution method (**b**) is to some extent unrealistic. In fact, using this method, if a program α, reaches an abort state '$\perp$', α will backtrack to the most recent nondeterministic choice statement and execute another alternative. If there are no more alternatives then α backtracks to the next recent one and so on. If all the alternatives are exhausted in this way the execution terminates but no result is produced. It is unclear to me how an implementation of a language can adopt such a technique.

Execution methods (**b**) and (**d**) are discussed in Hoare [1978]. In fact, Floyd [1967b] originally suggested execution method (**d**).

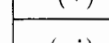
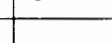
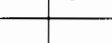
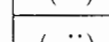
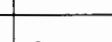
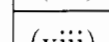
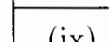
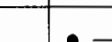
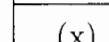
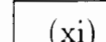
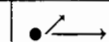
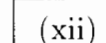
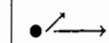
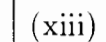
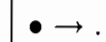
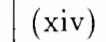
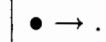
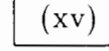
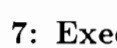
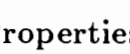
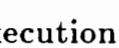
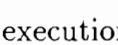
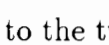
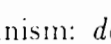
	(a)	(b)	(c)	(d)
(i)				
(ii)				
(iii)				
(iv)				
(v)				
(vi)				
(vii)				
(viii)				
(ix)				
(x)				
(xi)				
(xii)				
(xiii)				
(xiv)				
(xv)				

Table 7: Execution Properties and Execution Methods I

Finally I relate the execution methods to the two dual notions of nondeterminism: *demonic* and *angelic*.

A *demonic* notion of nondeterminism assumes there is some demon controlling the nondeterminism in the sense that if there is at least one nondeterministic choice that will terminate messily then this choice will be chosen and the program will not terminate cleanly. In other words if anything can go *wrong* it will. Any statement about a program which must be true *regardless of any nondeterministic choices* that may occur during program execution can be interpreted in terms of demonic nondeterminism. For example, statements such as:

(i) 'α always starts from s'

(ii) 'α always terminates from s'

(iii) 'every final state of α from s has property Q'

(iv) 'every state $s \in P$ has a property Q'

can all be interpreted in terms of a demonic notion of nondeterminism because:

- if α sometimes or always does not start from s, then (i) is false,
- if α sometimes or always does not terminate from s, then (ii) is false,
- if some or every final state of α from s does not have property Q, then (iii) is false,
- if some or every state $s \in P$ does not have property P, then (iv) is false.

Dually, an *angelic* notion of nondeterminism assumes there is some angel controlling the nondeterminism in the sense that if there is at least one nondeterministic choice that will terminate cleanly then this choice will be chosen and the program will terminate cleanly. In other words, if anything can go *right* it will. Any statement about a program which must be true for *at least one nondeterministic choice* made during program execution can be interpreted in terms of angelic nondeterminism. For example, statements such as:

(i) 'α sometimes starts from s'

(ii) 'α sometimes terminates from s'

(iii) 'some final state of α from s has property Q'

(iv) 'some state $s \in P$ has a property Q'

can all be interpreted in terms of an angelic notion of nondeterminism because:

- if α sometimes or always starts from s, then (i) is true,
- if α sometimes or always terminates from s, then (ii) is true,
- if some or every final state of α from s has property Q, then (iii) is true,
- if some or every state $s \in P$ has property P, then (iv) is true.

Using execution method **(a)** (or **(c)**) to execute a nondeterministic program α *every* execution sequence (or exseq) of α is considered to be an execution of α. In a sense if anything can go wrong it will. Therefore execution methods **(a)** and **(c)** give rise to a *demonic* interpretation of nondeterminism (Dijkstra [1976], Main [1987], Jacobs and Gries [1985]). On the other hand using execution method **(b)** (or **(d)**) to execute a nondeterministic program α only the cleanly terminating execution sequences (or exseqs) of α are considered to be computations of α. In this sense if anything can go right it will. Therefore execution methods **(b)** and **(d)** give rise to an *angelic* interpretation of nondeterminism (Floyd [1967b], Main [1987], Jacobs and Gries [1985]).

In summary, the main ideas introduced in this chapter are:

- 15 execution properties listed in **Table 1**,
- 24 representation methods listed in **Table 3**, and
- 4 execution methods described in **Table 6**.

The central role of these ideas is reflected in the rest of this thesis.

Chapter 2

Dijkstra's Weakest Precondition Semantics

For many years, computer programming was considered a skill, in the sense that computer programmers combined a little commonsense and intuition with hours of debugging to make a program perform as expected. Such an approach is not conducive to well designed programs. Dissatisfied with its inadequacies, Hoare [1969] introduced a formal approach based on the notion of *correctness* of programs. However, formality alone leads to incomprehensibly detailed proofs, making it difficult to determine if an already existing program fulfils its purpose. During the early 1970's, the need to combine formality with commonsense and intuition *during* program development became apparent. The challenge was to separate the mathematical concerns (of *what* is produced by a program) from the operational concerns (of *how* a program produces a result on an abstract machine).

The insights of the 1970's into the nature of computer programming culminated in the *weakest precondition semantics* introduced in Dijkstra's original research paper [1975]. This paper was the forerunner of his classic monograph [1976] which made a significant impact on the methods of designing and proving the correctness of programs: Dijkstra ([1975], [1976]) advocated that a program should be proved correct not after but concurrently with its design. Subsequently, Dijkstra's approach has been presented in textbook form in Gries [1981] and offered as a program methodology in Backhouse [1986], Dromey [1989] and Morgan

[1990]. Dijkstra's weakest precondition semantics forms the core topic of this thesis. My aim in this chapter is to give a comprehensive yet comprehensible overview of the topic, and simultaneously to lay the foundation for work done in later chapters.

Hoare's [1969] approach was originally presented in the framework of *deterministic* programs (that is, programs for which there is at most one outcome from each initial state). However, Dijkstra ([1975], [1976]) introduced his weakest precondition semantics in the more general framework of *nondeterministic* programs: each program may have several possible outputs for at least one of its inputs. The semantics does not constrain which of the possibilities will actually be produced by a particular execution of the program.

Following McCarthy [1963], a *state* is taken to mean an assignment of some value to each program variable – hence, in effect, a sequence of values. Dijkstra ([1975], [1976]) assumed that the input of a program is reflected in the choice of an *initial state*, while a *final state* reflects the output of a program. The set of all possible states of programs is called the *state space*, denoted, here by $\mathcal{S}$. A *predicate*, in the first instance, is an interpreted formula in a first-order language. However, as in Dijkstra ([1976] p 14) it is common practice to equate a predicate Q with the set of all states in which Q is true, and I adopt this useful ambiguity without further mention. Predicates therefore are subsets of $\mathcal{S}$ (that is, elements of $\mathcal{P}(\mathcal{S})$). I will denote programs by α, β, $\gamma, \ldots$, states by s, t, $u, \ldots$ and predicates by P, Q, $R, \ldots$. (This notation differs from Dijkstra ([1975], [1976]), Gries [1981] and Dijkstra and Scholten [1990].)

The main idea is that the meaning of a (nondeterministic) program α is given by describing, for any predicate Q which is desired to be true upon termination of α, the set of all states s such that execution of α from s will result in α terminating in a state where Q is true. This set is called the *weakest precondition of α with respect to Q*, written $wp(\alpha, Q)$. Note that for any program α, '$wp(\alpha, -)$' is a mapping from predicates to predicates, that is $wp(\alpha, -) : \mathcal{P}(\mathcal{S}) \rightarrow \mathcal{P}(\mathcal{S})$. This mapping is called the *weakest precondition predicate transformer* for α. A predicate which characterises a set of final states of a program is called a *postcondition* for the program. A predicate which characterises a set of initial states of a program is called

a *precondition* for the program.

So a weakest precondition predicate transformer is a tool for describing the initial states in which a program must be started in order to produce a final state in a given postcondition, and for ignoring the intermediate states which may occur during program execution. Intuitively, we can think of a program as some sort of inexplicable leap from an initial state to a final state - that is, a *black box*. One virtue of this approach, as Dijkstra ([1976] p xiv) explains, is that we do not have to worry about the details of implementations and can therefore reason about properties even of programs with no realistic operational interpretation.

As the formal derivation of programs developed, Dijkstra's ([1975], [1976]) original method no longer provided adequate formalism. Furthermore, Dijkstra's ([1975], [1976]) approach was originally concerned with an input-output analysis of program behaviour. However, some properties of programs require a finer analysis of the program behaviour, for example, *unbounded nondeterminacy* (that is, there exists at least one state such that no finite upper bound on the number of possible next states can be given). Recently, Dijkstra and Scholten [1990] rejuvenated Dijkstra's ([1975], [1976]) weakest precondition semantics and presented a more formal self-contained approach to remedy these shortcomings.

2.1 Dijkstra's Guarded Command Language

At the outset, Dijkstra ([1976] p xiii) was faced with the question: 'Which programming language am I going to use?'. The complexity of a language with many features is likely to be reflected in the methodology used for specifying the semantics of programs written in the language. For this reason Dijkstra ([1975], [1976]) restricted his exposition to only the essential features of a programming language and defined a simple informal language. One virtue of such a simplification, as eloquently expounded in Dijkstra ([1976] p 215) is that programming can be presented as a discipline rather than a craft: 'a discipline that would assist us in keeping our program intelligible, understandable and manageable'. After all, programming is a way of thinking, the purpose of which 'is to reduce the detailed reasoning

needed to a doable amount' (Dijkstra [1976] p 216) and a programming language is a tool which has 'an influence on our thinking habits' (Dijkstra [1976] p xiii).

To avoid becoming immersed in details of formal definitions, the syntax of this programming language is defined, here as in Gries [1981], by appealing to the reader's knowledge of mathematics and programming. There are two special atomic programs, *skip* and *abort*. The former has the effect of doing nothing, while the latter always fails to reach a final state. Next, there is a whole class of atomic programs called *assignment statements*. These are of the form '$z := e$', where z is a program variable and e is some expression (for example, arithmetical) and intuitively understood as 'z becomes e' or 'assign to z the value of e'. A natural extension of the single assignment statement is the so-called *multiple assignment statement* of form '$z_1, z_2, \ldots, z_n := e_1, e_2, \ldots, e_n$' where the z_i's are distinct program variables and the e_i's are expressions. This program has the effect of simultaneously substituting the e_i's for the z_i's.

From the atomic programs, compound programs can be constructed in one of three ways. First, any two programs α and β may be *composed* into another program $\alpha;\beta$, intuitively understood as '*do* α, *then do* β'. Second, for any predicates $B_1, B_2, \ldots, B_n$ and programs $\alpha_1, \alpha_2, \ldots, \alpha_n$ where $n \geq 0$, there is an *alternative command* IF of the form:

$$if\ B_1 \rightarrow \alpha_1 \,[]\, B_2 \rightarrow \alpha_2 \,[]\, \ldots \,[]\, B_n \rightarrow \alpha_n\ fi$$

intuitively understood as 'select some true B_i and execute the corresponding α_i'. If either none of the B_i's evaluate to true or at least one B_i is not well-defined, the program will abort. Third, there is an *iterative command* DO of the form:

$$do\ B_1 \rightarrow \alpha_1 \,[]\, B_2 \rightarrow \alpha_2 \,[]\, \ldots \,[]\, B_n \rightarrow \alpha_n\ od$$

intuitively read as 'Repeat the following until no longer possible: select some true B_i and execute the corresponding α_i'; again $n \geq 0$. In the case where all the B_i's initially evaluate to false, the iterative command simply skips (Dijkstra [1976] p 39).

Note that each B_i ensures that the corresponding α_i is only executed under certain constraints and hence is called a *guard*. Each '$B_i \rightarrow \alpha_i$' is called a *guarded command*. For this

reason this programming language is sometimes referred to as a *guarded command language.* (Note that a command *is* a program.) Nondeterminacy can be introduced when at least two guards in an IF or DO construct are not mutually exclusive. (Note that nondeterminacy is not *always* introduced in this way because even if B_1 and B_2 are not mutually exclusive, if $\alpha_1 = \alpha_2$ then nondeterminacy is *not* introduced (by $if\ B_1 \rightarrow \alpha_1 \,[\!]\, B_2 \rightarrow \alpha_2\ fi$).)

Unlike an if or case statement in conventional programming languages (for example, Pascal, Algol 60), the alternative command IF in Gries ([1981] p 134) and Dijkstra and Scholten ([1990] p 145) is assumed to have no defaults. In other words the guards in an IF-construct must be jointly exhaustive, but they need not be mutually exclusive. Under this no-defaults assumption the empty guarded command, '$if\quad fi$', and the single guarded command, '$if\ B\ do\ \alpha$' (unless $B = \mathcal{S}$) are excluded. There is a danger of confusion here, since the definitions of IF in Gries ([1981] p 132) and Dijkstra and Scholten ([1990] p 139) do allow these two forms. The syntax of IF in Dijkstra ([1976] p 33) excludes '$if\quad fi$' but allows '$if\ B\ do\ \alpha$'. Dijkstra ([1976] p 34) simply remarks: 'If we allow the empty guarded command set as well, the statement '$if\quad fi$' is therefore semantically equivalent with the earlier statement *abort.*'

Both Dijkstra ([1976] p 34) and Gries ([1981] p 132) assume the well-definedness of all guards in the IF construct. Hence the only way in which an IF command can go wrong is if no guard is *true.* What happens when such a situation arises? If there are no defaults such a situation never occurs, nevertheless, Dijkstra/Gries do cater for it. In fact, in Dijkstra ([1976] p 26) and Dijkstra and Scholten ([1990] p 135) 'abort' means 'does not terminate', so what they are actually saying is that if no guard is true then IF is equivalent (in some sense) to a program which goes into an endless and unproductive loop. It is unclear what the virtues of such a notion are. Intuitively, it seems that if no guard is true nothing should happen. What will be made clear in Chapter 6, I trust, is that this intuitive insight (of nothing happening) can be made into a tidy treatment of the IF construct, both technically and conceptually. In this treatment, for any α the composition, 'IF;α' will always pass control from IF to α, even when no guard of IF is true.

To facilitate the exposition and minimise the technical details, I have effected some restrictions along the lines of Gries [1981]. First: there are no side effects of the evaluation of expressions and guards. This means that the evaluation of expressions and conditions may change no variable. For example, the execution of '$z := e$' may change only z, and the evaluation of e itself changes no variables. Consequently, expressions and guards can be considered as conventional mathematical entities (with properties such as associativity and commutativity of addition and logical laws). Second, I omit multiple assignment (Gries [1981] (9.2.1)), because it can be abbreviated to '$\overline{z} := \overline{e}$' where $\overline{z}$ represents an n-tuple $(z_1, \ldots, z_n)$ of program variables and $\overline{e}$ is an n-tuple $(e_1, \ldots, e_n)$ of expressions. Writing multiple assignment in this way does not change the (weakest precondition) semantics so, the results for multiple assignment statements are mere repetitions of those for single assignment statements. Third, I will assume the well-definedness of all expressions and guards.

Fourth, I restrict IF to at most two guards (that is, '$if\ B_1 \rightarrow \alpha_1 \,[]\, B_2 \rightarrow \alpha_2\ fi$') because a general IF command can be expressed a binary IF without changing the (weakest precondition) semantics. In particular,

$$\text{'}if\ B_1 \rightarrow \alpha_1 \,[]\, B_2 \rightarrow \alpha_2 \,[]\, \ldots \,[]\, B_n \rightarrow \alpha_n\ fi\text{', for } n \geq 2$$

can be written as:

$$\text{'}if\ B_1 \rightarrow \alpha_1 \,[]\, (B_2 \cup B_3 \ldots \cup B_n) \rightarrow \mathrm{IF}_2\ fi\text{'}$$

$$\text{where} \quad \mathrm{IF}_2 = if\ B_2 \rightarrow \alpha_2 \,[]\, \ldots \,[]\, B_n \rightarrow \alpha_n\ fi$$

Intuitively, '$if\ B_1 \rightarrow \alpha_1 \,[]\, (B_2 \cup B_3 \ldots \cup B_n) \rightarrow \mathrm{IF}_2\ fi$' can be understood as follows. Let $C_2 = B_2 \cup B_3 \ldots \cup B_n$. There are four cases: first, if B_1 is true and C_2 is false (that is, $B_2, \ldots, B_n$ are all false) then execute α_1, second if B_1 is false and C_2 is true (that is, at least one of $B_2, \ldots, B_n$ is true) then execute IF_2; third, if B_1 is true and C_2 is true then execute *one* of α_1 or IF_2 without knowing which; fourth, if both B_1 and C_2 are false then *abort*. Executing IF_2 can be understood as selecting some true B_i (for $i = 2, 3, \ldots, n$) and then executing the corresponding α_i. This corresponds to the intuitive semantics of '$if\ B_1 \rightarrow \alpha_1 \,[]\, B_2 \rightarrow \alpha_2 \,[]\, \ldots \,[]\, B_n \rightarrow \alpha_n\ fi$', for $n \geq 2$. Applying an analogous argument to

$\mathrm{IF}_i = if\ B_i \rightarrow \alpha_i \,[]\, \ldots \,[]\, B_n \rightarrow \alpha_n\ fi$ (for $i = 2, 3, \ldots, n$)

it follows that a general IF command can be expressed as a sequence of binary IF's without changing the semantics.
In particular,

$$\text{'}if\ B_1 \rightarrow \alpha_1 \ [] \ B_2 \rightarrow \alpha_2 \ [] \ \ldots \ [] \ B_n \rightarrow \alpha_n \ fi\text{'}, \text{ for } n \geq 2$$

can be written as:

$$\text{'}if\ B_1 \rightarrow \alpha_1 \ [] \ (B_2 \bigcup B_3 \ldots \bigcup B_n) \rightarrow \beta_2 \ fi\text{'}$$

where $\beta_i = if\ B_i \rightarrow \alpha_i \ [] \ (B_{i+1} \bigcup \ldots \bigcup B_n) \rightarrow \beta_{i+1} \ fi$ for $i = 2, 3, \ldots, n-1$

$\beta_n = \alpha_n$.

Since the formal definition of the general IF is a simple extension of binary IF it is a mere notational matter to extend the results obtained for two guards to n guards.

Also, if there are no defaults,

$$\text{'}if\ B\ do\ \alpha\text{'}$$

must be written as

$$\text{'}if\ B \rightarrow \alpha \ [] \ \neg B \rightarrow skip\ fi\text{'}.$$

Fifth, DO is restricted to *one* guard (that is, '$do\ B \rightarrow \alpha\ od$', or more familiarly '$while\ B\ do\ \alpha$') since, as Gries ([1981] p 139) and Dijkstra and Scholten ([1990] p 188) point out, in the presence of the general IF command the simple DO will suffice. That is,

$$\text{'}do\ B_1 \rightarrow \alpha_1 \ [] \ B_2 \rightarrow \alpha_2 \ [] \ \ldots \ [] \ B_n \rightarrow \alpha_n \ od\text{'}, \text{ for } n \geq 0$$

is equivalent to

$$\text{'}do\ BB \rightarrow if\ B_1 \rightarrow \alpha_1 \ [] \ B_2 \rightarrow \alpha_2 \ [] \ \ldots \ [] \ B_n \rightarrow \alpha_n \ fi\ od\text{'}$$

or

$$\text{'}do\ BB \rightarrow IF\ od\text{'} \text{ where } BB = B_1 \bigcup \ldots \bigcup B_n.$$

Intuitively, $do\ B \rightarrow \alpha\ od$ can be understood as: 'Repeat the following, until it has no effect: 'check whether B is true and if so do α'. Then execute *skip*'. Since DO *skips* when

$B_1 \bigcup \ldots \bigcup B_n$ is false, DO can be intuitively read as: 'Repeat the following, until it has no effect: 'check whether $B_1 \bigcup \ldots \bigcup B_n$ is true and if so: select some true B_i and execute the corresponding α_i'. Then execute *skip*'. In other words (from the intuitive semantics of IF): 'Repeat the following, until it has no effect: 'check whether $B_1 \bigcup \ldots \bigcup B_n$ is true and if so execute IF'. Then execute *skip*.' But this corresponds to the intuitive semantics of '*do* $B \rightarrow \alpha$ *od*' with B replaced by $B_1 \bigcup \ldots \bigcup B_n$ and α by IF. Hence intuitively, '*do* $BB \rightarrow IF$ *od*' is semantically equivalent to DO.

2.2 An Algebra of Weakest Preconditions

To specify the semantics of a language it is necessary to state what each construct in the language means. The Dijkstra/Gries methodology is to characterise programs (that is, constructs) by axiomatising the behaviour of weakest precondition predicate transformers. This means that the possible performance of a program is reflected by the behaviour of its associated predicate transformer.

One of Dijkstra's ([1975], [1976]) primary objectives was to present a set of rules to assist in the design of better programming languages. To this end beginning with a program α he postulated for the weakest precondition predicate transformer '$wp(\alpha, -)$' a set of axioms which underlie our reasoning about programs written in any programming language. In the following I use the numbering in Gries [1981].

For any program α and any predicates Q and R,

(7.3) **Law of the Excluded Miracle:** $wp(\alpha, \emptyset) = \emptyset$

(7.4) **Distributivity of Conjunction:** $wp(\alpha, Q) \bigcap wp(\alpha, R) = wp(\alpha, Q \bigcap R)$

(7.5) **Law of Monotonicity:** If $Q \subseteq R$ then $wp(\alpha, Q) \subseteq wp(\alpha, R)$

(7.6) **Distributivity of Disjunction:** $wp(\alpha, Q) \bigcup wp(\alpha, R) \subseteq wp(\alpha, Q \bigcup R)$

(7.7) For any deterministic program α, $wp(\alpha, Q) \bigcup wp(\alpha, R) = wp(\alpha, Q \bigcup R)$

(Note that the terminology in (7.4) and (7.6) is misleading: (7.4) and (7.6) really express the distributivity law (for the weakest precondition predicate transformer $wp(\alpha, -)$) *with respect*

to rather than *of* conjunction and disjunction, respectively. The laws (if they existed) for distibutivity *of* conjunction and disjunction could respectively be given by equations of the form: $wp(\alpha, Q) \bigcap wp(\beta, Q) = wp(\alpha \bigcap \beta, Q)$, and $wp(\alpha, Q) \bigcup wp(\beta, Q) = wp(\alpha \bigcup \beta, Q)$.) These axioms, which if successfully applied lead to a systematic derivation of implementable programs, were appropriately named *healthiness properties* (or criteria) in Hoare [1978]. (Note that (7.5) and is a consequence of (7.4) and (7.6) (or (7.7)) is in turn a consequence of (7.5). So as pointed out in Hoare ([1978] p 469) (7.5) and (7.6) (or (7.7)) could be ignored.) A fifth healthiness condition, expressing a continuity property for '$wp(\alpha, -)$', introduced by Dijkstra ([1976] p 74) is:

> **Law of Continuity:** For any increasing chain (under $\subseteq$) of predicates $\{P_i\}_{i \geq 0}$, $wp(\alpha, \bigcup_i P_i) = \bigcup_i wp(\alpha, P_i)$.

Postulating the Law of Continuity guarantees that the programs at issue are boundedly nondeterministic. For suppose α is unboundedly nondeterministic, that is suppose there is an initial state t from which α is guaranteed to terminate but from which an infinite number of different final states $s_1, s_2, s_3 \ldots$ each satisfying $\bigcup_i P_i$ are possible. Define $Q_0 = \bigcup_i P_i - \{s_1, s_2, s_3, \ldots\}$ and $Q_{i+1} = Q_i \bigcup \{s_{i+1}\}$ for $i \geq 0$. Then $\{Q_i\}_{i \geq 0}$ is an increasing chain of predicates with lub $(\bigcup_i P_i)$ (that is, $\bigcup_i Q_i = \bigcup_i P_i$). Now $t \in wp(\alpha, \bigcup_i Q_i)$ since $\bigcup_i Q_i$ is true in each of $s_1, s_2, s_3, \ldots$. But there is no $i \geq 0$ such that Q_i is true for all final states of α since only finitely many of them are added at each stage. Thus $t \notin \bigcup_i wp(\alpha, Q_i)$; hence $wp(\alpha, -)$ is not continuous at $\bigcup_i P_i$, a contradiction.

To fit the guarded command language (in §1), Dijkstra ([1975], [1976]) imposed some constraints on '$wp(\alpha, -)$' for each atomic program and introduced a set of axioms for constructing new weakest precondition predicate transformers in such a way that whatever can be constructed by applying them is a weakest precondition predicate transformer satisfying the healthiness conditions, provided each component weakest precondition predicate transformer does. In the following I use the numbering in Gries [1981].

(8.1) $wp(skip, Q) = Q$

(8.2) $wp(abort, Q) = \emptyset$

(8.3) $wp(\alpha; \beta, Q) = wp(\alpha, wp(\beta, Q))$

(9.1.3) $wp(\text{`}z := e\text{'}, Q) = Q_e^z$, where Q_e^z denotes the predicate which differs from Q only in that each occurrence of variable z is replaced by the value of the expression e.

(10.3b) $wp(if\ B_1 \rightarrow \alpha_1 \ [\!]\ B_2 \rightarrow \alpha_2\ fi, Q) = (B_1 \bigcup B_2) \bigcap (\neg B_1 \bigcup wp(\alpha_1, Q)) \bigcap (\neg B_2 \bigcup wp(\alpha_2, Q))$

(11.2) $wp(while\ B\ do\ \alpha, Q) = \bigcup_{n \geq 0} H_n(Q)$, where $H_0(Q) = \neg B \bigcap Q$, and $H_{n+1}(Q) = H_0(Q) \bigcup wp(if\ B\ do\ \alpha, H_n(Q))$

Since the axiomatisation is equational it is appropriate to think of it as an *algebra* and call it an *algebra of weakest preconditions*. It is this algebra to which the title of my thesis refers.

Under the assumption of bounded nondeterminacy each state of an execution of a program has only finitely many successor states. The set of all possible executions of a (boundedly nondeterministic) program from any given initial state will form a finitely branching tree (with respect to the next-state relation). If this tree contains infinitely many nodes (or states) then, by König's lemma (in for example, Smullyan ([1968] p 32)), there must be at least one infinite branch in the tree. This means that under the assumption of bounded nondeterminacy, the semantics of a program is based on the following inference: if from a given initial state s a program can produce an infinite number of different results then there must also be a nonterminating execution from s of the program. In particular, if a (boundedly nondeterministic) program is guaranteed to terminate from a given initial state, then each branch in the (finitely branching tree) will be finite. In this case, by Königs lemma, the tree itself must be finite and therefore there is only a finite number of different branches in the tree.

An iterative command is called *strongly terminating* if for each state s there is an integer n such that the loop is guaranteed to terminate in n or fewer iterations. (Termination that is not strong is called *weak* termination. (This terminology is used by Back [1980] who

attributes it to Dijkstra.) The predicate $H_n(Q)$ for $n \geq 0$ in (11.2) represents exactly the set of all states from which execution of *while* B *do* α terminates in n or fewer iterations with Q true. Dijkstra's weakest precondition $wp(while\ B\ do\ \alpha, Q)$ for the iterative command captures exactly the set of initial states from which execution terminates in a finite number of iterations with Q true. (This means that $wp(while\ B\ do\ \alpha, Q)$ formalises strong termination for *while* B *do* α.)

To illustrate these points, consider an implementation (in Dijkstra [1976] p 76) of a program which assigns to a variable x any positive integer:

α: $go\_on = true;\ x := 1;$
do $go\_on \longrightarrow x := x + 1$
$[]\ go\_on \longrightarrow false$
od

This loop has infinitely many different possible outputs, but termination is not guaranteed. If termination is enforced by replacing the first guarded command with
'$go\_on\ and\ x \leq N \longrightarrow x := x+1$' for some positive integer, then only finitely many different results can be produced.

Is the derivation of weakest preconditions practical for all programs? By way of example I will consider the $if\ B_1 \rightarrow \alpha_1\ []\ B_2 \rightarrow \alpha_2\ fi$ and *while* B *do* α constructs. (Recall from §1 that the results for the general IF and DO command are extensions of those for $if\ B_1 \rightarrow \alpha_1\ []\ B_2 \rightarrow \alpha_2\ fi$ and *while* B *do* α.) Gries ([1981] p 135) points out: 'often, we are not interested in the weakest precondition of an alternative command, but only in determining if a known precondition implies it'. Necessary and sufficient conditions under which $X \subseteq wp(if\ B_1 \rightarrow \alpha_1\ []\ B_2 \rightarrow \alpha_2\ fi, Q)$ is true are given in Dijkstra ([1975] p 456) as Theorem 1, in Dijkstra ([1976] p 37) as 'The Basic Theorem for the Alternative Construct' and in Gries ([1981] p 135) as Theorem (10.5). The theorem follows:

(1) **Theorem** *For any predicate Q,*

$X \subseteq wp(if\ B_1 \rightarrow \alpha_1 \ [\!]\ B_2 \rightarrow \alpha_2\ fi), Q)$ iff (i) $X \subseteq B_1 \bigcup B_2$

(ii) $X \bigcap B_1 \subseteq wp(\alpha_1, Q)$

(iii) $X \bigcap B_2 \subseteq wp(\alpha_2, Q)$ □

In general, it is difficult to determine the weakest precondition $wp(while\ B\ do\ \alpha,\ \mathcal{S})$ such that the iterative construct *will* terminate. Therefore, as Gries ([1981] p 140) puts it: 'we want to develop a theorem that allows us to work with a useful precondition of a loop (with respect to a postcondition) that is not the weakest precondition'.

This sought-after precondition is called an *invariant* of the loop: it is '... a predicate P that is true before and after each iteration of [the] loop' (Gries [1981] p 141). So the idea is that if $s \in P$ and $while\ B\ do\ \alpha$ is executed from s, then the final state is again an element of P.

In the Dijkstra/Gries formulation of invariants termination is not built in — it must be proved separately by a *bound function* t for a loop with respect to a given input. Let $t(s)$ denote the bound on a loop when started in state s. Then $t(s)$ provides an upper bound on the number of iterations of the loop still to be performed. The idea is that $t(s)$ is bounded below by 0 provided execution of $while\ B\ do\ \alpha$ has not terminated. Each iteration of $while\ B\ do\ \alpha$ decreases $t(s)$ by at least one so that termination is guaranteed to occur. For suppose an unbounded number of iterations were performed. Then $t(s)$ would be decreased below any limit which would lead to a contradiction.

Why is the notion of an invariant useful? The idea is that an invariant, as a precondition of a loop, is easier to obtain than the weakest precondition. Namely, for '$while\ B\ do\ \alpha$' and a given postcondition Q, if we can find an invariant I such that $\neg B \bigcap I \subseteq Q$, then I will be a precondition of the loop — that is, $I \subseteq wp(while\ B\ do\ \alpha, Q)$. The reasoning is that if execution is started in I it remains in I; upon termination B is false (hence $\neg B$ is true). But then any final state is in $\neg B \bigcap I$, hence in Q. This is essentially shown in Dijkstra ([1975] p 456) as Theorem 4, in Dijkstra ([1976] p 38) as 'The Basic Theorem for the Repetitive Construct' (also 'The Fundamental Invariance Theorem for Loops'), in Gries ([1981] p 144)

as Theorem (11.6) ('a theorem concerning a loop, an invariant and a bound function') and in Dijkstra and Scholten ([1990] p 180) as the 'Main Repetition Theorem'. That is, (when predicates are thought of as sets)

(2) **Theorem** *For any predicates I, B and Q, and any program* α,

(a) *if* $B \cap I \subseteq wp(\alpha, I)$, *and*

(b) *if for every state s in which both I and B are true, t is greater than zero, and*

(c) *if for every state s in which I and B are true and in which for* $t \leq t_1 + 1$ *for some variable* t_1, $wp(\alpha, t \leq t_1)$ *is true,*

then

$$I \subseteq wp(while\ B\ do\ \alpha, I \cap \neg B).$$

Dijkstra's weakest precondition semantics are not useful for the derivation of programs with unbounded nondeterminism. Dijkstra's ([1976] p 77) reason for excluding unbounded nondeterminism is that when using guarded commands it is impossible to implement a program which from a given initial state is guaranteed to terminate and also may produce inifinitely many different results. To emphasise the problem, Dijkstra ([1976] p 77) considered the following program:

$$\begin{array}{lll} \beta: & do\ x \neq 0 \longrightarrow & if\ x > 0 \longrightarrow x := x - 1 \\ & & [\!]\ x < 0 \longrightarrow \text{`}set\ x\ to\ any\ positive\ integer\text{'} \\ & & fi \\ & od & \end{array}$$

where execution of the program '*set x to any positive integer*' is guaranteed to terminate with x equal to some positive integer, but no *a priori* upper bound for the final value of x can be given. When initiated in a state in which $x = -1$, the set of all possible executions of β will form the infinitely branching tree in **Figure 5**:

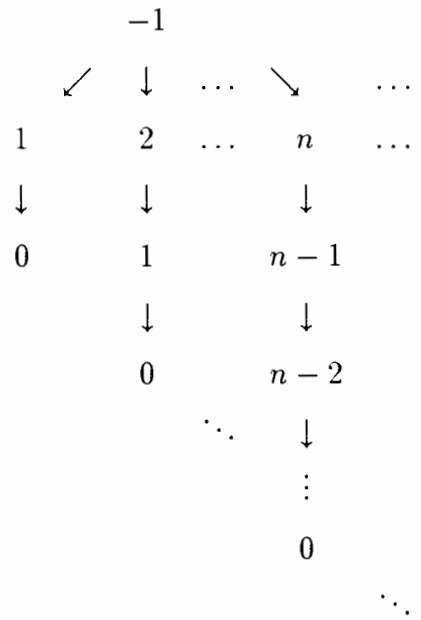


Figure 5

In this tree, -1 branches to infinitely many points – Königs lemma is no longer applicable. Each branch of this tree is finite, but for each integer n, there is a branch of length greater than n (the branch through $n+1$). This means (as first observed by Back [1980]) there is no integer k such that β is guaranteed to terminate in less than k iterations and hence for any state s in which $x = -1$, $s \notin wp(\beta, \mathcal{S})$ (because Dijkstra's [1976] weakest precondition semantics depends on strong termination (p 40) for loops). But '$s \notin wp(\beta, \mathcal{S})$' if and only if '$\beta$ never terminates from state s' or 'β terminates weakly from state s'. This means weakest precondition semantics cannot distinguish between these two execution properties. Our intuitive understanding (as represented in **Figure 5**) of the behaviour of β from a state in which $x = -1$ cannot be captured by the weakest precondition semantics of β.

Dijkstra ([1976] p 206) points out, program execution becomes significant when dealing with unbounded nondeterminism. In this connection a predicate which only guarantees that the program will not produce the wrong result but does not guarantee strong termination is more meaningful than the weakest precondition of α with respect to a given postcondition. Such a predicate for a program α and a given postcondition Q is called the *weakest liberal precondition of* α *with respect to* Q, written $wlp(\alpha, Q)$ in Dijkstra ([1976] p 21). This

approach, as presented in Dijkstra and Scholten [1990] is the subject of the next section. The semantics of unbounded nondeterminism has also been addressed in, for example, Back [1980], Boom [1982] and Hesselink [1990].

2.3 Dijkstra's Revised Mathematical Methodology

Dijkstra and Scholten [1990] give a self-contained algebraic-logical presentation of an (improved) axiomatic approach. The formal material, presented in the first author's usual cultured style, proves to be difficult reading. There are two major innovations (with respect to Dijkstra's ([1975], [1976]) earlier work) in the new approach: the inclusion of unbounded nondeterminism and the treatment of the semantics of the repetitive construct. To cater for these, Dijkstra and Scholten [1990] have revised the earlier [1976] mathematical methodology and effected some notational adaptations of predicate calculus.

My intention in this section is to a give a preview of a few terminological and notational conventions adopted throughout Dijkstra and Scholten [1990] before outlining (in §4) its new approach to program semantics via predicate transformers.

A *state space* is defined to be a nonempty Cartesian product space, components of which are thought of as values of program variables. A total function defined on the state space is called a *structure*. (For example, integer- and boolean matrices, etc) can all be treated as structures.) In particular, a *boolean structure* is a total boolean-valued function defined on the state space. A predicate is therefore a boolean structure and a mapping between boolean structures corresponds to a predicate transformer. (Dijkstra and Scholten's [1990] approach is based on a calculus of boolean structures and on mappings between boolean structures.) There are two special boolean structures, *true* and *false* which are called *boolean scalars*. These boolean scalars always exist since the state space is nonempty. The symbols '$=$', '$\Rightarrow$' and '$\Leftarrow$' denote mappings from a pair of boolean structures to a boolean structure:

For any boolean structures P and Q,

- '$P = Q$' means P and Q are *equal* as functions

(that is, P and Q both map a state to *true* (or to *false*)), and

- '$P \Rightarrow Q$' means P *implies* Q as functions
  (that is, no state is mapped to *true* by P and to *false* by Q), and
- '$P \Leftarrow Q$' means P *follows from* Q as functions
  (that is, no state is mapped to *false* by P and to *true* by Q)

A *square bracket* notation ,'[ ]', is introduced to denote a mapping from boolean structures to boolean scalars such that for any boolean structure X, $[X]$ = *true* if and only if X, as a function, maps every state to *true*. In particular, [*true*] = *true* and [*false*] = *false*. If X is a boolean structure which is true for some states and false for the others then [X] = *false*. (Dijkstra and Scholten [1990] did not consider boolean structures which are undefined for some states.) The mapping '[ ]' is called an '*everywhere*' *operator* (Dijkstra and Scholten [1990] p 8). It is the identity function for boolean scalars (that is, boolean scalars are those boolean structures solving the equation $X : [[X] \equiv X]$) and it is idempotent (that is, for any boolean structure Y, $[[[Y]] \equiv [Y]]$). The convention introduced in Dijkstra and Scholten ([1990] p 9) is that square brackets '[ ]' are used to express 'complete' equality between two operands that might be structures. For example, the Rule of Leibnitz traditionally written as:

$$x = y \Rightarrow f(x) = f(y)$$

is written (in Dijktra and Scholten ([1990] p 9) as:

$$[x = y] \Rightarrow [f(x) = f(y)]$$

The first pair of '[]' is needed because the arguments x and y could be structures; the second pair of '[ ]' is needed because f may be structure-valued.

For example, suppose '+' is an infix operation on integers and '$\geq$' denotes an ordering on integers. Then for integer structures x and y, '$x + y$' is an integer structure, '$x + y \geq 2$' is a boolean structure and '$[x + y \geq 2]$' is a boolean scalar. When treating boolean matrices as structures, a pointwise comparison of the components of two matrices yields a boolean structure and for any boolean matrix X, $[X]$ = *true* if and only if every component of X

gets mapped to *true*. In particular, if A is the matrix

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$$

and B is the matrix

$$\begin{bmatrix} 7 & 6 \\ 5 & 4 \end{bmatrix}$$

then '$A = B$' is the boolean structure

$$\begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix}$$

(where 0 corresponds to *false* and 1 to *true*) and '$[A = B]$' is the boolean scalar *false*.

Throughout the book a new proof annotation format is adopted: a proof of $P = R$ is written in the form:

P
$= \{$ hint why $P = Q$ $\}$
Q
$= \{$ hint $\}$
$\vdots$
$= \{$ hint $\}$
R

A proof of '$P \Rightarrow R$' (or '$P \Leftarrow R$') is written similarly, except at least one step may use '$\Rightarrow$' (or '$\Leftarrow$') instead of '$=$'. Only the most essential hints should be included so as not to detract from the proof. This proof format is reminiscent of the 'intermediate assertion' method of program proving in the early days of Floyd [1967a].

Two functions from pairs of boolean structures to boolean structures are postulated: *equivalence* (Dijkstra and Scholten [1990] p 32) and *disjunction* (Dijkstra and Scholten [1990] p 35) denoted respectively by '$\equiv$' and '$\vee$'. (Note that '$\equiv$' is an alternative for '$=$' used only when associativity holds (Dijkstra and Scholten [1990] p 10, 32)). (Note that what Dijkstra and Scholten [1990] are doing is nothing new: they are using the fact from set-theory that

given any set X the collection of functions from X to {F, T} can have a boolean algebra structure defined on it. In particular, the function '$\vee$' from a pair of boolean structures to a boolean structure is postulated to be symmetric, associative, idempotent, distributive over $\equiv$, distributive over itself and have the boolean scalar *true* as zero element.)

A novel idea of Dijkstra and Scholten ([1990] p 37) is to define *conjunction*, written '$\wedge$', in terms of equivalence and disjunction. For this they use a relation called 'The Golden Rule' (Dijkstra and Scholten [1990] (5,18)):

$$[X \wedge Y \equiv X \equiv Y \equiv X \vee Y]$$

In conjunction with associativity of $\equiv$ this formula can be parsed as:

either $[(X \wedge Y) \equiv ((X \equiv Y) \equiv (X \vee Y))]$,

or $[((X \wedge Y) \equiv (X \equiv Y)) \equiv (X \vee Y)]$,

or $[(X \wedge Y) \equiv (X \equiv Y \equiv (X \vee Y))]$,

or $[((X \wedge Y) \equiv X) \equiv (Y \equiv (X \vee Y))]$,

or $[((X \wedge Y) \equiv X \equiv Y) \equiv (X \vee Y)]$,

or $[X \wedge (Y \equiv X \equiv Y \equiv (X \vee Y))]$.

Conjunction is a function from pairs of boolean structures to boolean structures with properties that follow from those postulated for '$\equiv$' and '$\vee$'. Another such function, called *implication* and denoted by '$\Rightarrow$', is defined by (Dijkstra and Scholten [1990] (5,37)):

$$[X \Rightarrow Y \equiv X \vee Y \equiv Y]$$

Analogously, a function denoted by '$\Leftarrow$', read as '*follows from*', is defined by (Dijkstra and Scholten [1990] (5,44)):

$$[X \Leftarrow Y \equiv X \wedge Y \equiv Y]$$

Negation, denoted by '$\neg$', is a function between boolean structures with properties postulated with respect to '$\equiv$' and '$\vee$'(Dijkstra and Scholten [1990] p 51).

The properties of these functions can be summarised as follows: $(\mathcal{S}, \Rightarrow)$ is a partially ordered set (that is, a poset) with a maximum element *true* and a minimum element *false* in which

any two boolean structures P and Q have a greatest lower bound, $P \wedge Q$, and a least upper bound, $P \vee Q$, where $\vee$ and $\wedge$ distribute over each other. Furthermore any boolean structure P has a complement, $\neg P$, such that $P \vee \neg P = \mathit{true}$ and $P \wedge \neg P = \mathit{false}$. This means that boolean structures form a boolean algebra, ($\mathcal{S}$, $\vee$, $\wedge$, $\neg$, *true*, *false*).

Finally, with a view to characterising the semantics of the repetitive constructs, Dijkstra and Scholten ([1990] p 76) show that every chain of boolean structures under the ordering $\Rightarrow$ has a greatest lower bound (and also a least upper bound). So $(\mathcal{S}, \Rightarrow)$ is a poset with a minimum element *false* in which chains have least upper bounds (that is, $(\mathcal{S}, \Rightarrow)$ is a *domain* (in for example, Manes and Arbib [1986] p 149)).

To cater for an assignment statement, Dijkstra and Scholten ([1990] p 116) define a (prefix) predicate transformer for substitution:

$$(z := e)(Q)$$

intuitively understood as: 'Q with z replaced by the value of e' (or 'whatever Q says about z is true of the value of e').

In order to remedy the shortcoming concerning programs with unbounded nondeterminacy, encountered in Dijkstra [1976], Dijkstra and Scholten ([1990] Chapter 9) define the semantics of the repetitive construct in terms of solutions of equations of the form:

$$Y = b(X, Y)$$

where X is a predicate, that is, a boolean structure, and b is a total boolean-valued function over $\mathcal{S}$, that is, a structure over $\mathcal{S}$ (written in Dijkstra and Scholten ([1990] (8,0)) as Y : $[b.X.Y]$). For this Dijkstra and Scholten ([1990] (8,1)) consider simpler equations of the form $Y = b(Y)$ (written Y : $[b.Y]$), that is, *fixed points* of b. It is shown (in Dijkstra and Scholten ([1990] p 148,149)) that for every boolean-valued function b there is precisely one element $X \subseteq \mathcal{S}$ such that $b(X) = X$ and for each $Y \subseteq \mathcal{S}$ with $b(Y) = Y$, $X \subseteq Y$, namely $X = \bigcap\{Y \mid Y = b(Y)\}$ (provided it exists). This is called the *strongest solution* of $Y = b(Y)$ in Dijkstra and Scholten [1990]. (The ordinary terminology used is: X is the

least fixed point of b (Manes and Arbib [1986] p 153) (de Bakker [1980] p 70).) Likewise, the *weakest solution* of $Y = b(Y)$ (conventionally, the *greatest fixed point* of b) (if it exists) is the unique element $X = \bigcup\{Y \mid Y = b(Y)\} \subseteq \mathcal{S}$ such that $b(X) = X$ and for each $Y \subseteq \mathcal{S}$ with $b(Y) = Y$, $Y \subseteq X$.

The usefulness of least fixed points in the characterisation of semantics of the repetitive construct is pointed out in Manes and Arbib ([1986] §6.2). In this connection Dijkstra and Scholten [1990] prove two theorems (analogous to those in Manes and Arbib [1986]) concerning the existence of fixed points. A least fixed point is characterised as an infinite meet in the first and as an infinite join in the second. The first theorem deals with functions f over $(\mathcal{S}, \subseteq)$ with the property: if $X \subseteq Y$ then $f(X) \subseteq f(Y)$, that is, *monotonic functions.* This is:

> **Knaster-Tarski Theorem** (Dijkstra and Scholten ([1990] (8,25) p 154)
> *For any monotonic function f over $(\mathcal{S}, \subseteq)$, the strongest solution of $Y = f(Y)$ is $X = \bigcap\{Y \mid f(Y) \subseteq Y\}$ (and the weakest solution is $X = \bigcup\{Y \mid f(Y) \supseteq Y\}$).*

However, this theorem is not particularly useful in program semantics. Another characterisation is obtained by considering *continuous functions* over $(\mathcal{S}, \subseteq)$ – that is, functions f such that for any chain (under $\subseteq$) $\{P_i\}_{i \geq 0}$ of subsets of $\mathcal{S}$, $f(\bigcup_{i \geq 0} P_i) = \bigcup_{i \geq 0} f(P_i)$. This is:

> **Kleene's Fixed Point Theorem** (Dijkstra and Scholten ([1990] (8,60) p 163)
> *For any continuous function f over $(\mathcal{S}, \subseteq)$, the strongest solution of $Y = f(Y)$ is $X = \bigcup_{i \geq 0} f^i(\emptyset)$ (and the weakest solution is $X = \bigcap_{i \geq 0} f^i(\mathcal{S})$).*

Unfortunately, Dijkstra and Scholten ([1990] p 158) use some quite nondescript notations for the strongest and weakest solutions of equations of the form '$Y = f(X, Y)$', namely '$g.X$' and '$h.Y$' – these are commonly denoted respectively by 'μ' and 'ν' (de Bakker and de Roever [1973]). In particular, the least fixed point of $f(X, Y) = Y$ is denoted by $\mu X.[f(X, Y) = Y]$ and the greatest fixed point by $\nu X.[f(X, Y) = Y]$.

A final notion introduced to help describe bound functions for loops is that of well-foundedness (Enderton [1977] p 241-242), which is used instead of the ordering on natural numbers. This

notion of well-foundedness applies to posets and the relevant facts are accessible to anyone with knowledge about induction. In particular,

For any poset $(D, \subseteq)$, the following are equivalent:

(a) *D is well-founded*

(b) *D is well-ordered (that is, every nonempty subset of D has a minimal element)* (Dijkstra and Scholten ([1990] (9,16))

(c) *Course-of-values induction over D is valid*
(Dijkstra and Scholten ([1990] (9,20))

(d) *All decreasing chains in D are finite* (Dijkstra and Scholten ([1990] (9,22))

It is impressive how neatly well-foundedness, in particular its equivalence to course-of-values induction, fits into Dijkstra's 'Fundamental Invariance Theorem for Loops'.

2.4 The Weakest Liberal Precondition

Finally the preliminaries are over, now we come to the Dijkstra and Scholten [1990] program semantics which, as in §2, I present equationally. (Note that *true* and *false* correspond respectively to $\mathcal{S}$ and $\emptyset$.) Consistent with the methodology established in Dijkstra ([1975], [1976]), Dijkstra and Scholten [1990] postulate predicate transformers and healthiness conditions. The former notation '$wp(\alpha, Q)$' and '$wlp(\alpha, Q)$' are replaced with '$wp.\alpha.Q$' and '$wlp.\alpha.Q$' where the '.' denotes function application (as in lambda calculus (Hindley and Seldin [1986])). In this thesis I use the former notation.

They postulate for every program α a predicate transformer $wlp(\alpha, -)$ and a predicate $wp(\alpha, \mathcal{S})$ (rather than the predicate transformer $wp(\alpha, -)$) and then define $wp(\alpha, -)$ as (Dijkstra and Scholten [1990] (7,2)):

$$wp(\alpha, Q) = wp(\alpha, \mathcal{S}) \bigcap wlp(\alpha, Q)$$

for all predicates Q. This definition is consistent with Dijkstra's ([1975], [1976]) postulation of $wp(\alpha, -)$. In the first instance, $wlp(\alpha, Q)$ characterises the set of all states s such that

if execution of α from s terminates then it does so in Q. Secondly, $wp(\alpha, \mathcal{S})$ characterises the set of all states from which α terminates. Then $wp(\alpha, \mathcal{S}) \bigcap wlp(\alpha, Q)$ is the set of all initial states s such that every execution of α from s will terminate and does so in a state in Q. And this is exactly the weakest precondition of α with respect to Q as postulated in Dijkstra ([1975], [1976]).

The new set of healthiness conditions is:

For any program α:
(R0) $wlp(\alpha, \bigcap_i P_i) = \bigcap_i wlp(\alpha, P_i)$ for all $P_i \subseteq \mathcal{S}$
(R1) $wp(\alpha, \emptyset) = \emptyset$

Condition (R0) implies (as shown in Dijkstra and Scholten ([1990] p 132)):

$wlp(\alpha, \emptyset) = \emptyset$ and
$wp(\alpha, \bigcap_i P_i) = \bigcap_i wp(\alpha, P_i)$ for all $P_i \subseteq \mathcal{S}$.

The intended effect of (R1) is to eliminate states from which a program α sometimes terminates and sometimes does not terminate (that is termination/nontermination property B(2) in Chapter 1, p 3). It also eliminates states from which a program α does not always start (that is initialisation property A(2) in Chapter 1, p 3) (Programs which sometimes start and sometimes do not start from a state are called *partial programs* in Nelson [1989]). It should be noted that the original four healthiness conditions (§2, p 38) (as postulated by Dijkstra [1975]) are direct consequences of (R0) and (R1) and hence are not explicitly required. These are as follows: (The proofs are in Dijkstra and Scholten [1990].)

For any program α and any predicates Q and R,

(7.3) **Law of the Excluded Miracle:**

$wlp(\alpha, \mathcal{S}) = \mathcal{S}$

$wp(\alpha, \emptyset) = \emptyset$

(7.4) **Distributivity of Conjunction:**

$wlp(\alpha, Q) \bigcap wlp(\alpha, R) = wlp(\alpha, Q \bigcap R)$

$wp(\alpha, Q) \bigcap wp(\alpha, R) = wp(\alpha, Q \bigcap R)$

(7.5) **Law of Monotonicity:**

If $Q \subseteq R$ then $wlp(\alpha, Q) \subseteq wlp(\alpha, R)$

If $Q \subseteq R$ then $wp(\alpha, Q) \subseteq wp(\alpha, R)$

(7.6) **Distributivity of Disjunction:**

$wlp(\alpha, Q) \bigcup wlp(\alpha, R) \subseteq wlp(\alpha, Q \bigcup R)$

$wp(\alpha, Q) \bigcup wp(\alpha, R) \subseteq wp(\alpha, Q \bigcup R)$

(7.7) For any deterministic program α,

$wlp(\alpha, Q) \bigcup wlp(\alpha, R) = wlp(\alpha, Q \bigcup R)$

$wp(\alpha, Q) \bigcup wp(\alpha, R) = wp(\alpha, Q \bigcup R)$

The fifth healthiness condition (the Law of Continuity, (§2.2, p 8)) has been excluded since, as Dijkstra and Scholten ([1990] p 125) explain, 'in order to reason in a trustworthy manner about abstract programs one has to know how to cope with unbounded nondeterminism'.

Dijkstra's ([1975], [1976]) guarded command language is extended with the introduction of an (unboundedly nondeterministic) program called *havoc*. This program has the effect of terminating in an unpredictable state from a given initial state. Now the intention in Dijkstra and Scholten [1990] is to capture the semantics of this program and those in §1 by imposing conditions on '$wlp(\alpha, -)$' and '$wp(\alpha, \mathcal{S})$'. These are the following: (I use the numbering of Dijkstra and Scholten [1990].)

(7.16) $wlp(skip, Q) = Q$

(7.17) $wp(skip, \mathcal{S}) = \mathcal{S}$

(7.13) $wlp(abort, Q) = \mathcal{S}$

(7.14) $wp(abort, \mathcal{S}) = \emptyset$

(7.10) $wlp(havoc, Q) = \mathcal{S}$, if $Q \neq \emptyset$

(7.11) $wp(havoc, \mathcal{S}) = \mathcal{S}$

(7.23) $wlp(\alpha; \beta, Q) = wlp(\alpha, wlp(\beta, Q))$

(7.24) $wp(\alpha; \beta, \mathcal{S}) = wp(\alpha, wp(\beta, \mathcal{S}))$

(7.19) $wlp(\text{`}z := e\text{'}, Q) = (z := e)(Q)$

(7.20) $wp(\text{`}z := e\text{'}, \mathcal{S}) = \mathcal{S}$

(7.27) $wlp(if\ B_1 \rightarrow \alpha_1 \ [] \ B_2 \rightarrow \alpha_2\ fi, Q) = (\neg B_1 \bigcup wlp(\alpha_1, Q)) \bigcap (\neg B_2 \bigcup wlp(\alpha_2, Q))$

(7.28) $wp(if\ B_1 \rightarrow \alpha_1 \ [] \ B_2 \rightarrow \alpha_2\ fi, \mathcal{S}) = (B_1 \bigcup B_2) \bigcap (\neg B_1 \bigcup wp(\alpha_1, \mathcal{S})) \bigcap$
$(\neg B_2 \bigcup wp(\alpha_2, \mathcal{S}))$

These definitions are indeed valid since as shown in Dijkstra and Scholten [1990] the commands they define satisfy (R0) and (R1). It is worth noting that there is no theorem corresponding to Dijkstra's theorem 'The Basic Theorem for the Alternative Construct' and Gries's ([1981] p 135) Theorem (10.5) (that is, Theorem (2.1) in this thesis) because using the new approach the problem of determining $wp(if\ B_1 \rightarrow \alpha_1 \ [] \ B_2 \rightarrow \alpha_2\ fi, Q)$ is broken into two smaller problems, namely, determining $wlp(if\ B_1 \rightarrow \alpha_1 \ [] \ B_2 \rightarrow \alpha_2\ fi, Q)$ and determining $wp(if\ B_1 \rightarrow \alpha_1 \ [] \ B_2 \rightarrow \alpha_2\ fi, \mathcal{S})$.

Although the iterative construct is somewhat more complicated to handle than the other constructs, if the command α is deterministic then so is DO. In the case of the repetitive construct, the issue of termination or rather nontermination must be considered with care. In particular, termination of DO is guaranteed for all initial states in which $wp(DO, \mathcal{S})$ holds. What follows is a brief account of the semantics of the simple *while* B *do* α.

The idea is that $wp(while\ B\ do\ \alpha, Q)$ is the set of all those states s such that execution of *while* B *do* α from s *will* terminate in a state Q. Then in Dijkstra and Scholten ([1990]

(9,2)) $wp(while\ B\ do\ \alpha, Q)$ is defined to be the strongest solution of the mapping

$$f(Y) = (B \bigcup Q) \bigcap (\neg B \bigcup wp(\alpha, Y))$$

$(= (B \cup Q) \cap (\neg B \cup (\neg B \cup wp(\alpha, Y)) = (B \cup Q) \cap (\neg B \cup B) \cap (\neg B \cup (\neg B \cup wp(\alpha, Y))) =$
$(B \cup Q) \cap (\neg B \cup wp(if\ B \rightarrow \alpha\ fi, Y)))$. That is,
$wp(while\ B\ do\ \alpha, Q) = \bigcap\{Y \mid Y = f(Y)\}$ where $f(Y) = (B \cup Q) \cap (\neg B \cup wp(if\ B \rightarrow \alpha\ fi, Y))$
So, here, initially no upper bound on the number of execution steps is needed.

Analogously, $wlp(while\ B\ do\ \alpha, Q)$ is the set of all those states s such that if execution of $while\ B\ do\ \alpha$ from s terminates then it does so in a final state in Q. Then $wlp(while\ B\ do\ \alpha, Q)$ is defined in Dijkstra and Scholten ([1990] (9,1)) to be the weakest solution of the mapping

$$g(Y) = (B \bigcup Q) \bigcap (\neg B \bigcup wlp(\alpha, Y))$$

$(= (B \cup Q) \cap (\neg B \cup (\neg B \cup wlp(\alpha, Y)) = (B \cup Q) \cap (\neg B \cup wlp(if\ B \rightarrow \alpha\ fi, Y)))$. That is,
$wlp(while\ B\ do\ \alpha, Q) = \bigcup\{Y \mid Y = g(Y)\}$ where $g(Y) = (B \cup Q) \cap (\neg B \cup wlp(if\ B \rightarrow \alpha\ fi, Y))$

Finally, Dijkstra and Scholten ([1990] (9,26)) formulate a new version of Dijkstra's [1976] theorem for the iterative construct. When predicates are thought of as sets, this theorem is as follows:

> **Theorem** *For any program α, any predicates I and B and any arithmetic expression t,*
>
> (a) *if for every state s in which both I and B are true, t is greater than zero, and*
>
> (b) *if for every state s in which I and B are true and in which t has value t_1 for some variable t_1, $wp(\alpha, I \cap t < t_1)$ is true,*
>
> *then*
>
> $$I \subseteq wp(while\ B\ do\ \alpha, I \cap \neg B).$$

Replacing (c) in Theorem (2.1) (of this thesis) by
(c)' for every state s in which I and B are true and in which t has value t_1 for some variable t_1, $wp(\alpha, t < t_1)$ is true,

and using Gries ([1981] (7.4)) to combine (2.1)(a) and $(c)'$, it turns out that Gries's formulation of Dijkstra's [1976] theorem for the iterative construct reduces to that of Dijkstra and Scholten [1990].

2.5 Categorising the State Space

A major concern in the study of program semantics is to develop a framework and language in which we can provide a helpful classification of executions. To maintain consistency with the Dijkstra/Gries type exposition I will assume that:

(i) α always starts from s, and

(ii) nontermination is equated with messy termination.

The fundamental question is: What do we want to be able to say of a program α, an initial state s, and a postcondition Q? Recall from Chapter 1 the key to answering this question lies in the execution and final state properties of a program from a state.

First concerning the execution of α from state s we ask: Does α always, sometimes, sometimes not or never terminate from s? So we want the termination/nontermination property of a program α started from a state s. There are eight possibilities arising from the termination/nontermination properties (B(1),B(2) and B(3) in Chapter 1, p 3). Namely, α neither terminates nor does not terminate from s; B(1) α always terminates from s; B(2) α sometimes terminates from s; B(3) α never terminates from s; [B(1) and B(3)] α sometimes terminates and sometimes does not terminate from s; [B(1) and B(2)] α sometimes or always terminates from s; [B(2) and B(3)] α sometimes or always does not terminate from s; and [(B(1), B(2) and B(3)] there is an execution of α from s.

Secondly concerning the final states of α from state s we ask: Does every or some final state of α from s have a given property Q? or Does every or some final state of α from s not have a given property Q? So we want the final state properties of a program α from a state s. There are eight possibilities arising from final state properties (D(1), D(2) and D(3) in Chapter 1, p 3). Namely: there is no final state of α from s; D(1) every final state of α from

s has property Q; D(2) some final state of α from s has property Q; D(3) no final state of α from s has property Q; [D(1) and D(2)] some or every final state of α from s has property Q; [D(1) and D(3)] every or no final state of α from s has property Q; [D(2) and D(3)] some or every final state of α from s does not have property Q; [D(1), D(2) and D(3)] there is a final state of α from s.

From these possibilities we can identify eight categories (a) - (h) of the state space $\mathcal{S}$. Namely:

(a) $\{s \mid \alpha$ always terminates from $s\}$

(b) $\{s \mid \alpha$ sometimes terminates from $s\}$

(c) $\{s \mid \alpha$ sometimes does not terminate from $s\}$

(d) $\{s \mid \alpha$ never terminates from $s\}$

(e) $\{s \mid$ every final state of α from s has property $Q\}$

(f) $\{s \mid$ some final state of α from s has property $Q\}$

(g) $\{s \mid$ some final state of α from s has property $\neg Q\}$

(h) $\{s \mid$ every final state of α from s has property $\neg Q\}$

corresponding respectively to B(1), [B(1) and B(2)], [B(2) and B(3)], B(3), D(1), [D(1) and D(2)], [D(2) and D(3)], and D(3).

Knowing that the Dijkstra/Gries language of '*wp*' and '*wlp*' is a language for reasoning about programs, we enquire whether it captures each of the above eight cases.

So my goal is to capture the above eight sets by formulating conditions on executions and states. For this the interpretation of a program as a *black box* is inadequate since the details of executions are ignored. Instead I will view a program (operationally) as a *sequence* of atomic steps. Recall from Chapter 1 (p 24, 25) since programs are nondeterministic, any

given initial state gives rise to an *execution tree* (or *extree* for short). I will assume that the meaning of a program is given by the set of all its possible execution trees.

The set Seq($\mathcal{S}$) of exseqs is defined as follows:

$\mathcal{S}^+$ denotes the set of all finite non-empty sequences of elements of $\mathcal{S}$.

$\mathcal{S}^\infty$ denotes the set of all infinite sequences of elements of $\mathcal{S}$.

$\text{Seq}(\mathcal{S}) = \mathcal{S}^+ \bigcup \mathcal{S}^\infty$.

I denote exseqs by $\mathbf{x}$, $\mathbf{y}$, $\mathbf{z}$, ... etc, the idea being that when I write '$\mathbf{x} = (x_1, x_2, x_3, \ldots)$' it is left open whether or not $\mathbf{x}$ is finite. But '$\mathbf{x} = (x_1, x_2, \ldots, x_n)$' means $\mathbf{x}$ *is* finite and has x_n as last element, where $x_n \in \mathcal{S}$. (Note that it can never be the case that some $x_i \notin \mathcal{S}$ (for $i = 1, 2, \ldots, n$) because by (ii) there is no state of messy termination.)

I now define some operations on exseqs. For any $\mathbf{x} \in \text{Seq}(\mathcal{S})$, say $\mathbf{x} = (x_1, x_2, x_3, \ldots)$:

$$first(\mathbf{x}) = x_1$$

$$last(\mathbf{x}) = \begin{cases} x_n & \text{if } \mathbf{x} \in \mathcal{S}^+ \text{ and } \mathbf{x} = (x_1, x_2, \ldots, x_n) \\ \text{undefined} & \text{if } \mathbf{x} \in \mathcal{S}^\infty \ . \end{cases}$$

We need a notation for the set of all execution sequences of a program from an initial state. For any program α and any initial state s

$$\mathbf{extree}(\alpha, s) = \{\mathbf{x} \in \text{Seq}(\mathcal{S}) \mid first(\mathbf{x}) = s \text{ and } \mathbf{x} \text{ is an exseq of } \alpha\}$$

Then $\mathbf{extree}(\alpha, s) \subseteq \text{Seq}(\mathcal{S})$ and divides into two subsets as follows:

$\mathbf{fin}(\alpha, s)$ denotes the set of all finite exseqs of α from s.

$\mathbf{infin}(\alpha, s)$ denotes the set of all infinite exseqs of α from s.

Thus $\mathbf{extree}(\alpha, s) = \mathbf{fin}(\alpha, s) \bigcup \mathbf{infin}(\alpha, s)$.

Note that '$\mathbf{extree}(\alpha, s) = \emptyset$' would mean that α does not start from s. Recall from p 57 that we have assumed that every program is defined for every state. (That is, for every program α and every state s, $\mathbf{extree}(\alpha, s) \neq \emptyset$.) Also '$\mathbf{extree}(\alpha, s) = \mathbf{fin}(\alpha, s)$' means 'every execution of α from s is finite' or ' every execution of α from s terminates'. Recall from

the earlier discussion on bounded/unbounded nondeterminacy (§2) that in the particular case of a *boundedly* nondeterministic program α, '$\mathbf{extree}(\alpha, s) = \mathbf{fin}(\alpha, s)$' is equivalent to '$\mathbf{extree}(\alpha, s)$ is finite' by König's lemma. (Note that a deterministic program α can proceed in exactly one way from any state s, so in this case $\mathbf{extree}(\alpha, s)$ is a singleton.)

Finally I introduce a notation for the set of all final states of a program from a given initial state. For any program α and any initial state s

$$out(\alpha, s) = \{t \mid (\exists \mathbf{x} \in \mathbf{extree}(\alpha, s))\, [t = last(\mathbf{x})]\} = \{last(\mathbf{x}) \mid \mathbf{x} \in \mathbf{extree}(\alpha, s)\}$$

(Note that I am not using sequences in the sense that the limit of an infinite exseq of α from s is an output for α from s. So $out(\alpha, s)$ captures the final states of cleanly terminating executions (that is, exseqs in $\mathbf{fin}(\alpha, s)$) of α from s.) Then for any predicate Q, '$out(\alpha, s) \subseteq Q$' means 'every final state of every (terminating) computation of α from s is one in which Q is true'.

Using the above notation we can easily express the eight sets (a)-(h):

(a) $\{s \mid \alpha \text{ always terminates from } s\} = \{s \mid \mathbf{infin}(\alpha, s) = \emptyset\}$

(b) $\{s \mid \alpha \text{ sometimes terminates from } s\} = \{s \mid \mathbf{fin}(\alpha, s) \neq \emptyset\}$

(c) $\{s \mid \alpha \text{ sometimes does not terminate from } s\} = \{s \mid \mathbf{infin}(\alpha, s) \neq \emptyset\}$

(d) $\{s \mid \alpha \text{ never terminates from } s\} = \{s \mid \mathbf{fin}(\alpha, s) = \emptyset\}$

(e) $\{s \mid \text{every final state of } \alpha \text{ from } s \text{ has property } Q\} = \{s \mid out(\alpha, s) \subseteq Q\}$

(f) $\{s \mid \text{some final state of } \alpha \text{ from } s \text{ has property } Q\} = \{s \mid out(\alpha, s) \bigcap Q \neq \emptyset\}$

(g) $\{s \mid \text{some final state of } \alpha \text{ from } s \text{ has property } \neg Q\} = \{s \mid out(\alpha, s) \bigcap \neg Q \neq \emptyset\}$

(h) $\{s \mid \text{every final state of } \alpha \text{ from } s \text{ has property } \neg Q\} = \{s \mid out(\alpha, s) \subseteq \neg Q\}$

Note that there are variations. For example, provided α is defined for state s,
(d) $\{s \mid \alpha \text{ never terminates from } s\} = \{s \mid out(\alpha, s) = \emptyset\}$

In fact, we can capture all these eight sets by defining two primitive functions:

at: programs $\longrightarrow$ predicates, defined by

$at(\alpha) = \{s \mid \alpha$ always terminates from $s\}$

and

af: programs $\times$ predicate $\longrightarrow$ predicates, defined by

$af(\alpha, Q) = \{s \mid$ every final state of α from s has property $Q\}$

I use terminology 'af' for '*a*ll *f*inal states' and 'at' for '*a*lways *t*erminates'. (Note that $af(\alpha, \mathcal{S}) = \mathcal{S}$ and if $Q \subseteq R$ then $af(\alpha, Q) \subseteq af(\alpha, R)$ (that is, $af(\alpha, -)$ is monotonic over predicates). Now,

(a) $\{s \mid \alpha$ always terminates from $s\} = at(\alpha)$

(b) $\{s \mid \alpha$ sometimes terminates from $s\} = \neg af(\alpha, \emptyset)$

(c) $\{s \mid \alpha$ sometimes does not terminate from $s\} = \neg at(\alpha)$

(d) $\{s \mid \alpha$ never terminates from $s\} = af(\alpha, \emptyset)$

(e) $\{s \mid$ every final state of α from s has property $Q\} = af(\alpha, Q)$

(f) $\{s \mid$ some final state of α from s has property $Q\} = \neg af(\alpha, \neg Q)$

(g) $\{s \mid$ some final state of α from s has property $\neg Q\} = \neg af(\alpha, Q)$

(h) $\{s \mid$ every final state of α from s has property $\neg Q\} = af(\alpha, \neg Q)$

Using the two primitive functions at and af (that is, implicitly, the eight sets above) we can easily express the Dijkstra/Gries predicate transformers wp and wlp. Namely, for any program α:

$wp(\alpha, -)$: predicates $\longrightarrow$ predicates, defined by

$wp(\alpha, Q) = at(\alpha) \cap af(\alpha, Q)$

(Note that $wp(\alpha, \mathcal{S}) = at(\alpha) \cap af(\alpha, \mathcal{S}) = at(\alpha)$.)

and

$wlp(\alpha, -)$: predicates $\longrightarrow$ predicates, defined by

$wlp(\alpha, Q) = af(\alpha, Q)$

Note that $wp(\alpha, Q) = wp(\alpha, \mathcal{S}) \cap wlp(\alpha, Q)$. Hence Dijkstra and Scholten [1990] take $wlp(\alpha, -)$ and $wp(\alpha, \mathcal{S})$ as primitives. (The latter really is a mapping $wp(-, \mathcal{S})$: programs $\longrightarrow$ predicates.) These are exactly our primitives *af* and *at*. Therefore,

(a) $\{s \mid \alpha$ always terminates from $s\} = wp(\alpha, \mathcal{S})$

(b) $\{s \mid \alpha$ sometimes terminates from $s\} = \neg wlp(\alpha, \emptyset)$

(c) $\{s \mid \alpha$ sometimes does not terminate from $s\} = \neg wp(\alpha, \mathcal{S})$

(d) $\{s \mid \alpha$ never terminates from $s\} = wlp(\alpha, \emptyset)$

(e) $\{s \mid$ every final state of α from s has property $Q\} = wlp(\alpha, Q)$

(f) $\{s \mid$ some final state of α from s has property $Q\} = \neg wlp(\alpha, \neg Q)$

(g) $\{s \mid$ some final state of α from s has property $\neg Q\} = \neg wlp(\alpha, Q)$

(h) $\{s \mid$ every final state of α from s has property $\neg Q\} = wlp(\alpha, \neg Q)$

This shows that, given a few assumptions on definedness, the Dijkstra/Gries language of '*wp*' and '*wlp*' really does capture the eight cases.

2.6 Discussion

In conclusion, I provide a critical assessment of the original Dijkstra ([1975],[1976]) approach and the current mature Dijkstra and Scholten [1990] approach to programming language semantics. In my opinion there are a number of respects in which neither is entirely adequate for characterising the semantics of programs in the sense required by practical considerations. (The list is not meant to be in order of priority.)

First, Dijkstra ([1975], [1976]) and Dijkstra and Scholten [1990] consider only one way in which a program can 'go wrong': abortion due to no guard of an IF evaluating to true. But (as I mentioned in Chapter 1) in practice there are several ways in which (an execution of) a program may fail, for example, 'overflow', 'underflow' (due to a value being out of range), 'break' (due to a deliberate break in program execution), 'undefinedness' (due to, say, division by zero), etc. In order to cater for the other failures which may arise, Gries [1981], imposed conditions under which the assignment and alternative commands do not fail. However, along with Dijkstra ([1975], [1976]) (and Dijkstra and Scholten 1990]), for ease of exposition, Gries [1981] also assumed the well-definedness of expressions used in assignment commands and guards.

Second, the law of excluded miracle only allows the construction of programs which can be started from every possible initial state. This means that every state is a possible initial state of any program and hence Dijkstra ([1975], [1976]) and Dijkstra and Scholten [1990] implicitly consider only *total* programs. However it may be reasonable (in some context) to consider a notion of *partial* programs (that is, programs which may not always start). Nelson [1989] simplified Dijkstra's [1976] healthiness conditions by dropping this law for which he could see no practical significance.

Third, although recursive procedures play a central role in computer science, Dijkstra ([1975], [1976]) chose to omit them from his language of guarded commands. His primary concern was to select the most effective constraints on programming languages rather than to consider 'as most universal a programming language as possible' (Dijkstra [1976] p 213). One of the reasons for the omission, as Dijkstra ([1976] p xvi) explains is:

> The point is that I felt no need for them in order to get my message across, viz. how a carefully chosen separation of concerns is essential for the design of in all respects, high-quality programs: the modest tools of the mini-language gave us already more than enough latitude for nontrivial, yet satisfactory designs.

Nelson [1989] generalised the weakest precondition semantics, constituted in Dijkstra [1976], to include the semantics of recursion.

Fourth, the treatment of a single guarded command '$if\ B \rightarrow \alpha\ fi$' is somewhat unclear. Although this case is not explicitly mentioned in Dijkstra ([1975], [1976]), it would seem that the familiar command, '$if\ B\ do\ \alpha$' corresponds to '$if\ B \rightarrow \alpha\ fi$'. However, due to the lack of defaults assumed in Gries ([1981] p 134) such a command corresponds to '$if\ B \rightarrow \alpha \ [] \ \neg B \rightarrow skip\ fi$'. Does this mean that there is *no* one-guarded alternative command (unless $B = \mathcal{S}$ in which case the guard does nothing)? The same discrepancy arises in Dijkstra and Scholten ([1990] p 138) where '$if\ B \rightarrow \alpha\ fi$' is allowed but '$if\ B\ do\ \alpha$' corresponds to '$if\ B \rightarrow \alpha \ [] \ \neg B \rightarrow skip\ fi$' (Dijkstra and Scholten ([1990] p 145). However, Dijkstra and Scholten ([1990] p 193) observe that '$if\ B \rightarrow \alpha\ fi$', when initiated in a state in which B holds, corresponds to one iteration of '$while\ B\ do\ \alpha$'. To see this recall, from §4, the definition of the semantics of the repetitive command. In particular, for any predicate Q,

$$\begin{aligned} wlp(while\ B\ do\ \alpha, Q) &= (B \bigcup Q) \bigcap (\neg B \bigcup wlp(\alpha, wlp(DO, Q)) \\ &= (B \bigcup Q) \bigcap (\neg B \bigcup wlp(\alpha; while\ B\ do\ \alpha, Q) \\ &= wlp(`if \neg B \rightarrow skip \ [] \ B \rightarrow \alpha; while\ B\ do\ \alpha\ fi', Q) \end{aligned}$$

and

$$\begin{aligned} wp(while\ B\ do\ \alpha, Q) &= (B \bigcup Q) \bigcap (\neg B \bigcup wp(\alpha, wp(while\ B\ do\ \alpha, Q)) \\ &= (B \bigcup Q) \bigcap (\neg B \bigcup wp(\alpha; while\ B\ do\ \alpha, Q) \\ &= wp(`if \neg B \rightarrow skip \ [] \ B \rightarrow \alpha; while\ B\ do\ \alpha\ fi', Q) \end{aligned}$$

This means that '$if \neg B \rightarrow skip \ [] \ B \rightarrow \alpha; while\ B\ do\ \alpha\ fi$' is semantically equivalent to '$while\ B\ do\ \alpha$'. However, operationally, an execution of '$while\ B\ do\ \alpha$' consists of a (finite or infinite) sequence of executions of '$if\ B \rightarrow \alpha\ fi$' (or simply α) under the constraint that B is true before each of these executions. Execution of '$while\ B\ do\ \alpha$' terminates if and only if B does not hold after a finite number (possibly zero) of these executions of '$if\ B \rightarrow \alpha\ fi$' (or simply α). So, in general, '$while\ B\ do\ \alpha$' consists of a (finite or infinite) sequence of executions of '$if \neg B \rightarrow skip \ [] \ B \rightarrow \alpha; while\ B\ do\ \alpha\ fi$' (or of '$if\ B \rightarrow \alpha\ fi$'). Therefore the weakest precondition semantics of one or zero iterations of '$while\ B\ do\ \alpha$' is the same as the weakest precondition semantics of '$if\ B \rightarrow \alpha\ fi$'.

Chapter 3

Correctness

This chapter is concerned with a major problem confronting all programmers: whether or not a program behaves as intended. This is the problem of the *correctness* of a program. Important issues in this regard are the meanings of 'correctness', different notions of correctness and methods of increasing one's confidence that a program is correct. The first part of this chapter gives some background and justification as to why program correctness is an important area of study. In the second part I introduce what I take to be important factors influencing a program's correctness. I consider different possible constraints on a state in which a program is activated: 2 initialisation constraints and 26 execution constraints. These lead to 52 notions of correctness.

Since the advent of the first computers a significant amount of programmer effort has been deployed in fixing errors in programs. There are two main methods of producing programs without errors or with fewer errors: testing and proving. At first much emphasis was placed on the former. Testing a program involves executing it, either mechanically or manually, on a particular set of data. An important feature of testing is that the effect of the program is discovered only for a particular chosen set of input data by *inductive reasoning.* Such a technique can be performed without much thought and reveals only the presence of errors, not their absence.

The idea of proving properties of programs (for example, correctness) can be traced back to

the early 1960's. For example, in a paper concerning a mathematical theory of programs, McCarthy [1963] suggested that 'instead of trying out computer programs on test cases until they are debugged, one should prove that they have the desired properties'. Thereafter much attention was devoted to the *operational proofs of program correctness.* Such proofs are based on a model of computation, and the effects of executing a program on a machine are considered. It was only in the second half of the sixties that significant research on formal proofs of program correctness evolved.

The idea of formally proving program correctness emerged through the work of Naur [1966] and Floyd [1967a]. Naur [1966] provided an informal technique for specifying program proofs and, in a seminal paper [1967a], Floyd proposed that an adequate definition of a programming language could be obtained from the specification of proof techniques. Building on Naur's and Floyd's work, Hoare [1969] laid the foundations for much of the work in programming methodology, program proving and programming language design. Hoare's approach, currently known as an *axiomatic approach* (also an *algebraic approach*), was to define programming language constructs in terms of how programs containing them can be proved correct, rather than in terms of how they are to be executed. (The definition consisted of a logical system of axioms and inference rules. This logical system is called *Hoare logic* and sometimes *Floyd-Hoare logic.*) The main idea is that properties of a program and the effect of executing a program on a set of data are established from the program text by means of *deductive reasoning.*

Gradually (during the 1970's) it was recognised that a program and its proof should be developed concurrently, with the main emphasis on the idea of correctness. That is, the concerns of correctness and methods used in proving correctness influence the development of a program from the beginning. It was Dijkstra ([1975], [1976]) who clarified how this could be achieved via his calculus for the derivation of programs (that is, the algebra of weakest preconditions in Chapter 2.2). Dijkstra's algebra of weakest preconditions is primarily an algebra for rigorous program derivation rather than an algebra for post hoc verification. Hoare [1969] introduced *sufficient* conditions such that a program will produce the correct result if it terminates; while Dijkstra ([1975], [1976]) introduced *necessary and sufficient*

conditions (or weakest preconditions) such that a program is guaranteed to terminate and produce the correct result.

Variants of Hoare's approach include: specification languages and programming logics. Specification languages, of which a variety have been developed, are languages for writing programs which are to be verified. Many employ a set of axioms and inference rules based on first-order predicate calculus which can be used to give a precise statement of the effect that a program is required to achieve. Specific examples include Euclid (Lampson *et al* [1977]), the Vienna Development Method or VDM (also Vienna Development Language or VDL) (Jones [1986]) and Z (Spivey [1989], Norcliffe and Slater [1990]). Specification languages are useful for illustrating the practical consequences of attempting to design a language for which proofs are feasible.

Programming logics such as *dynamic logic* (Pratt [1976], [1979a], Harel [1979a]), *process logic* (Pratt [1979b]), *temporal logic* (Emerson [1990]) and *action logic* (Pratt [1990]) are further logical systems motivated by program correctness. The purpose of a programming logic is to provide a mathematical framework useful for specifying and verifying the correctness of programs. Programs are not actually written within the mathematical framework — it serves as a tool for describing certain behavioural patterns of programs. For example, dynamic logic can be used to describe the input-output behaviour of programs; while process logic and temporal logic can used to say what happens during an execution.

An overview of Hoare's [1969] approach and subsequent developments is provided in §1. In §2 I use the analysis of execution properties in Chapter 1 to introduce various notions of program correctness. Of these, however, only a few have previously been investigated. These are identified in §4. §5 is devoted to concepts of correctness which cater for both nontermination and messy termination — to cater for the other ways in which a program can go wrong. A comparison of the various notions of correctness for programs is given in §6.

3.1 Correctness of Programs

To begin with I expound the notion of correctness based on Hoare's axiomatic approach. One meaning of 'correctness' of programs arises by putting some constraints on the input to a program and the expected relation between its inputs and outputs. A constraint characterising the values of program variables before initialisation of a program is called an *input assertion* (or a *precondition*), while that characterising the values program variables should assume after execution of the program is called an *output assertion* (or a *postcondition*). An *assertion* is a statement about a program's state which is either true or false. Then the idea is that a notion of correctness of a program should reflect the relationship among an input assertion, an output assertion and the program. Hoare's [1969] original notation for expressing a notion of correctness for a program α with respect to an input assertion P and an output assertion Q is '$P\{\alpha\}Q$'. ('$\{P\}\alpha\{Q\}$' is now more widely used.) Such a triple is called a *Hoare triple* in Jacobs and Gries [1985].

A *proof of correctness* for a program involves showing formally that a program satisfies its specification. In order to achieve this, an axiom or inference rule is associated with each atomic or compound program of a programming language. These axioms or inference rules state what may be asserted after execution of the program in terms of what was true beforehand. So to prove a program is correct with respect to an input assertion P and an output assertion Q we assume the truth of P and use (like a proof in logic) the axioms and inference rules to try to establish the truth of Q. Eventually every atomic program in the complete program appears between two assertions, called *intermediate assertions.* In general, the input assertions used must describe all possible inputs for a program not only a finite (small) collection of specific inputs. Then a proof of correctness of a program is a proof over *all* the program inputs. A program together with assertions between each pair of statements is called a *proof outline.* Placing assertions in a program is called *annotating* the program and the final program is called an *annotated program.* For a complex language with many features the correctness proofs for programs are likely to be elaborate and this will be reflected in the complexity of the underlying axioms and rules.

Mathematical logic notations can be used to express assertions. If a proof of a program does not rely on special properties of the symbols used but only those required for all inputs then the proof is valid for *every* specific input. If special properties of the symbols (for example, $x \geq 0, x > 0$) must be asssumed to construct a proof then an exhaustive case analysis constitutes a complete proof.

The practice of proving the correctness of programs should not be regarded as a purely theoretical exercise. First, a proof of correctness for a program is a rigorous method of formulating the purpose of a program and conditions under which it will perform as intended. In this respect correctness proofs play an increasingly significant role in the *documentation* of programs. Second, correctness proofs using machine-independent axioms and inference rules reveal the machine-dependent features of programs and therefore can be used in establishing the *portability* of programs. Third, the axiomatic approach is an indispensible tool for writing programs which are simple and clear since programming language constructs are defined with a view to making proofs involving them easier to understand instead of making their execution easy. Fourth, the *reliability* of programs can be increased because correctness proofs can be used to detect errors and force the programmer to make explicit in the program text information for understanding and maintenance of the program.

However, due to the size and complexity of programs it may be difficult to develop input and output assertions against which to verify an already existing program and it may be impossible to show that, if found, the input and output assertions do in fact reflect the intentions of the programmer. To overcome such difficulties Dijkstra ([1975], [1976]) proposed a more goal-directed approach in which a program and its proof are developed simultaneously.

It seems natural to ask whether such an axiomatic approach provides an adequate alternative to the existing 'forward' development of programs. Dijkstra [1976] provided many examples to illustrate the extreme importance of the output assertion (instead of the input assertion) in developing programs. Suppose we want to find a program α such that $\{P\}\alpha\{Q\}$. Dijkstra's goal-directed approach requires that an input assertion P be *derived* from a program and an output assertion; while the 'historical' approach involves developing α from P only (without

reference to Q). The chance of producing (under the latter approach) a correct program which solves a given problem is remote. (See also Gries [1979a].)

The connection between Hoare's approach and Dijkstra's approach to program correctness is simple. For a program α, a precondition P and a postcondition Q, triples of the form $\{P\}\alpha\{Q\}$ are the basis of the approach to program correctness via Hoare logic. Dijkstra's ([1975], [1976]) approach to program correctness of a program α with respect to a postcondition Q, is to specify the least constrained precondition P, written $wp(\alpha, Q)$, such that $\{P\}\alpha\{Q\}$ holds (that is, $\{wp(\alpha, Q)\}\alpha\{Q\}$ holds and for any P such that $\{P\}\alpha\{Q\}$ holds $P \subseteq wp(\alpha, Q)$). Therefore, $\{P\}\alpha\{Q\}$ iff $P \subseteq wp(\alpha, Q)$. This means that the algebra of weakest preconditions can be written using triples of the form $\{P\}\alpha\{Q\}$. A complete list of the equivalent formulations for the formulae in the algebra of weakest preconditions is given in, for example, Gries [1979b] and Apt [1984].

It is interesting to note that Dijkstra and Scholten [1990] introduce in direct analogy to a proof outline, a proof format (Chapter 2.3): a proof together with hints (or assertions) between each pair of steps. One virtue of this idea is to give the reader the opportunity of familiarising himself with the methodology whereby programs and their specifications are developed together with correctness proofs.

The response to Dijkstra's ([1975], [1976]) approach has been lively, and there is now an extensive and still growing literature on Dijkstra's algebra of weakest preconditions. What was at first (or so it seemed) a purely theoretical exercise has grown into a programming methodology, expounded in three recent books: Backhouse [1986], Dromey [1989] and Morgan [1990] . The textbook of Gries [1981] is one of the first attempts to establish the respectability of the whole weakest precondition enterprise and to convince programmers that developing programs and proofs hand-in-hand often leads to correct programs which are shorter and clearer than those previously produced.

3.2 Notions of Correctness

My aim in this section is to introduce different possible meanings of the statement '*a program behaves as intended*'. Recall from Chapter 1 that we deal with two basic concepts: executions and states. These are treated on different levels of generality.

On the first level we have:

I (a) Execution properties (that is, initialisation and clean-, messy and nontermination properties) of a program started from a *given* initial state.

(b) Properties of final state(s) of a program started from a *given* initial state.

On the second level we have:

II (a) Execution properties (that is, initialisation and clean-, messy and nontermination properties) of a program started from *all* or *only some* states with a given property.

(b) Properties of final state(s) of a program started from *all* or *some* states with a given property.

In Chapter 1 I only considered the first level; now I consider both.

Recall from §1 that a notion of correctness should reflect some relationship between a precondition, a program and a postcondition. A meaning of correctness arises by putting constraints on all or only some states in which a program is activated. Now for a precondition P, a program α and a postcondition Q, the following phenomena need investigation:

(1) Constraints on a state s with property P in which a program α is activated. These arise from:

(i) the initialisation properties (A(1), A(2), A(3) in Chapter 1, p 3) of α from s, (that is, α always, sometimes or never starts from s),

and/or (ii) the termination/nontermination properties (B(1), B(2), B(3) in Chapter 1, p 3) of α from s, (that is, α always, sometimes or never terminates from s),

and/or (iii) the clean-/messy termination properties (C(1), C(2), C(3) in Chapter 1, p 3) of α from s, (that is, if α terminates from s, α always, sometimes or never terminates cleanly from s),

and/or (iv) the final state properties (D(1), D(2), D(3) in Chapter 1, p 3) of α from s (that is, every, some or no final state of α from s has a property Q).

and (2) Do all or only some initial states with property P satisfy the chosen constraints?

In this section I will assume that nontermination is equivalent to messy termination, so if a program starts from a state s it either terminates cleanly or does not terminate at all from s. Therefore I need only consider (1)(i), (1)(ii) and (2). My investigation is divided into two parts: in the first part I only consider the final states of cleanly terminating executions; in the second part I also consider the 'state of nontermination'.

Let $\mathcal{S}$ be state space and assume predicates are subsets of $\mathcal{S}$. I now consider possible constraints on a state s with property P in which a program α is activated.

First it must be decided which initialisation properties (A(1), A(2), A(3)) of a program α activated in a state s we wish to capture by a notion of correctness. Recall (from Chapter 1, p 8, 9) there are eight possibilities. Namely: we can capture either no executions or only executions which always, sometimes or never start from s (that is, respectively A(1), A(2) or A(3)); or only executions which always or sometimes start from s (that is, [A(1) and A(2)]); or only executions which sometimes or never start from s (that is, [A(2) and A(3)]); only executions which always or never start from s (that is, [A(1) and A(3)]) or all executions from s (that is, [A(1), A(2) and A(3)]).) Of these possibilities only three were considered for initialisation property representations in Chapter 1 (p 8): A(1), [A(1) and A(2)] and [A(1), A(2) and A(3)]. However the latter may only be interesting in a context where it is the normal behavioural pattern of programs never to start from some states. Here I will only consider A(1) and [A(1) and A(2)]. These two properties give rise to two constraints on a

state s with property P in which a program α is activated. Namely, we could choose:

C_1 Initialisation constraints

either (1) Only states from which α *always* starts are captured as initial states. (That is, a state from which α sometimes or always *does not start* is disregarded as an initial state.)

or (2) All states from which α *sometimes or always* starts are captured as initial states. (That is, only states from which α *never* starts are disregarded as initial states.)

(Constraint $\mathbf{C_1}(1)$ corresponds to the assumption (i) in Chapter 2.5 (p 56).)

Second, it must be decided which termination/nontermination properties for the program α started from a state s with a property P are to be captured by a notion of correctness. In Chapter 2.5 (p 57), we identified four categories (a) – (d). Namely:

(a) $\{s \mid \alpha$ always terminates from $s\}$

(b) $\{s \mid \alpha$ sometimes terminates from $s\}$

(c) $\{s \mid \alpha$ sometimes does not terminate from $s\}$

(d) $\{s \mid \alpha$ never terminates from $s\}$

corresponding respectively to B(1); [B(1) and B(2)]; [B(2) and B(3)] and B(3).

Recall from Chapter 1 (p 2) that negating 'α always terminates from s' yields 'α sometimes terminates from s'. Therefore category (c) is the complement of category (a). Likewise category (d) is the complement of category (b). Categories (a) and (b) give rise to two constraints on a state s with property P in which a program α is activated. Namely, we could choose:

C_2 Termination Constraints

either (1) Only states from which a program *always* terminates are captured as initial states. (That is, a state from which a program sometimes or always *does not* terminate is disregarded as an initial state.)

or (2) All states from which a program *sometimes or always* terminates are captured as initial states. (That is, only states from which a program *never* terminates are disregarded as initial states.)

(Then by negation two further constraints corresponding to categories (c) and (d) arise.)

Thirdly, the expected property Q of all or only some final states of a program α started from a state s leads to further constraints on states in which α is activated. It must be decided which (if any) of the properties D(1), D(2) and D(3) (in Chapter 1, p 3) are to be captured. In Chapter 2.5 (p 57) we identified four categories (e) - (h). Namely:

(e) $\{s \mid$ every final state of α from s has property $Q\}$

(f) $\{s \mid$ some final state of α from s has property $Q\}$

(g) $\{s \mid$ some final state of α from s does not have property $Q\}$

(h) $\{s \mid$ every final state of α from s does not have property $Q\}$

corresponding respectively to D(1); [D(1) and D(2)]; [D(2) and D(3)] and D(3).

Recall again from Chapter 1 (p 2) that negating 'every final state of α from s has property Q' yields 'some final state of α from s does not have property Q'. Therefore category (g) is the complement of category (e). Likewise category (h) is the complement of category (f). Categories (e) and (f) give rise to two constraints on a state s with property P in which a program α is activated. Namely, we could choose:

C_3 Outcome constraints

either (1) Only states such that *every* final state of α from s has property Q are captured as initial states. (That is, a state s such that some or every final state of α from s *does not* have property Q is disregarded as an initial state.)

or (2) All states such that *some or every* final state of α from s has property Q are captured as initial states. (That is, only states such that every final state of α from s *does not* have property Q are disregarded as initial states.)

(Then by negating (1) and (2) two further constraints corresponding respectively to categories (g) and (h) arise.)

Combining the two initialisation constraints ($\mathbf{C_1}(1)$ and $\mathbf{C_1}(2)$) and the two termination constraints ($\mathbf{C_2}(1)$ and $\mathbf{C_2}(2)$) we obtain the following four constraints.

$\mathbf{C_4}$ Initialisation and termination constraints

(1) $\mathbf{C_1}(1)$ and $\mathbf{C_2}(1)$:

Only states from which α always starts and always terminates are captured as initial states.

(2) $\mathbf{C_1}(1)$ and $\mathbf{C_2}(2)$:

Only states from which α always starts and sometimes or always terminates are captured as initial states.

(3) $\mathbf{C_1}(2)$ and $\mathbf{C_2}(1)$:

Of the states from which α sometimes or always starts only those from which α always terminates are captured as initial states.

(4) $\mathbf{C_1}(2)$ and $\mathbf{C_2}(2)$:

Of the states from which α sometimes or always starts only those from which α sometimes or always terminates are captured as initial states.

(Note: by negating the second conjunct (or the entire conjunction) in each of $\mathbf{C_4}(x)$ (for x = 1, 2, 3, 4) eight further constraints arise. For example, ($\mathbf{C_1}(1)$ and negation of $\mathbf{C_2}(1)$); (negation of $\mathbf{C_1}(1)$ or negation of $\mathbf{C_2}(1)$).)

Combining the two initialisation constraints ($\mathbf{C_1}(1)$ and $\mathbf{C_1}(2)$) and the two outcome constraints ($\mathbf{C_3}(1)$ and $\mathbf{C_3}(2)$) we obtain the following four constraints.

$\mathbf{C_5}$ Initialisation and outcome constraints

(1) $\mathbf{C_1}(1)$ and $\mathbf{C_2}(1)$:

Only states from which α always starts and from which every final state of α has a property Q are captured as initial states.

(2) $\mathbf{C_1}(1)$ and $\mathbf{C_2}(2)$:

Only states from which α always starts and from which some or every final state of α has a property Q are captured as initial states.

(3) $\mathbf{C_1}(2)$ and $\mathbf{C_2}(1)$:

Of the states from which α sometimes or always starts only those from which every final state of α has a property Q are captured as initial states.

(4) $\mathbf{C_1}(2)$ and $\mathbf{C_2}(2)$:

Of the states from which α sometimes or always starts only those from which some or every final state of α has a property Q are captured as initial states.

(Note: by negating the second conjunct (or the entire conjunction) in each of $\mathbf{C_5}(x)$ (for x = 1, 2, 3, 4) eight further constraints arise. For example, ($\mathbf{C_1}(1)$ and negation of $\mathbf{C_3}(1)$); (negation of $\mathbf{C_1}(1)$ or negation of $\mathbf{C_3}(1)$).)

Combining the two termination constraints ($\mathbf{C_2}(1)$ and $\mathbf{C_2}(2)$) and the two outcome constraints ($\mathbf{C_3}(1)$ and $\mathbf{C_3}(2)$) we obtain the following four constraints.

$\mathbf{C_6}$ Termination and outcome constraints

(1) $\mathbf{C_2}(1)$ and $\mathbf{C_3}(1)$:

Only states from which α always terminates and from which every final state of α has a property Q are captured as initial states.

(2) $\mathbf{C_2}(1)$ and $\mathbf{C_3}(2)$:

Only states from which α always terminates and from which some or every final state of α has a property Q are captured as initial states.

(3) $\mathbf{C_2}(2)$ and $\mathbf{C_3}(1)$:

Of the states from which α sometimes or always terminates only those from which every final state of α has a property Q are captured as initial states.

(4) $\mathbf{C_2}(2)$ and $\mathbf{C_3}(2)$:

Of the states from which α sometimes or always terminates only those from which

some or every final state of α has a property Q are captured as initial states.

(Note: by negating one or both conjuncts (or the entire conjunction) in each of $\mathbf{C_6}(x)$ (for x = 1, 2, 3, 4) sixteen further constraints arise. For example, ($\mathbf{C_2}(1)$ and negation of $\mathbf{C_3}(1)$); (negation of $\mathbf{C_2}(1)$ and $\mathbf{C_3}(1)$); (negation of $\mathbf{C_1}(1)$ and negation of $\mathbf{C_2}(1)$); (negation of $\mathbf{C_1}(1)$ or negation of $\mathbf{C_2}(1)$).)

Finally combining the two initialisation constraints ($\mathbf{C_1}(1)$ and $\mathbf{C_1}(2)$), the two termination constraints ($\mathbf{C_2}(1)$ and $\mathbf{C_2}(2)$) and the two outcome constraints ($\mathbf{C_3}(1)$ and $\mathbf{C_3}(2)$) we obtain another eight constraints. Namely:

$\mathbf{C_7}$ Initialisation, termination and outcome constraints

(1) $\mathbf{C_1}(1)$ and $\mathbf{C_6}(1)$:

Only states from which α always starts, from which α always terminates and from which every final state of α has a property Q are captured as initial states.

(2) $\mathbf{C_1}(1)$ and $\mathbf{C_6}(2)$:

Only states from which α always starts, from which α always terminates and from which some or every final state of α has a property Q are captured as initial states.

(3) $\mathbf{C_1}(1)$ and $\mathbf{C_6}(3)$:

Of the states from which α always starts and sometimes or always terminates only those from which every final state of α has a property Q are captured as initial states.

(4) $\mathbf{C_1}(1)$ and $\mathbf{C_6}(4)$:

Of the states from which α always starts and sometimes or always terminates only those from which some or every final state of α has a property Q are captured as initial states.

(5) $\mathbf{C_1}(2)$ and $\mathbf{C_6}(1)$:

Of the states from which α sometimes or always starts only those states from which α always terminates and from which every final state of α has a property Q are captured as initial states.

(6) $\mathbf{C_1}(2)$ and $\mathbf{C_6}(2)$:

Of the states from which α sometimes or always starts only those states from which α always terminates and from which some or every final state of α has a property Q are captured as initial states.

(7) $\mathbf{C_1}(2)$ and $\mathbf{C_6}(3)$:

Of the states from which α sometimes or always starts only those states from which α sometimes or always terminates and from which every final state of α has a property Q are captured as initial states.

(8) $\mathbf{C_1}(2)$ and $\mathbf{C_6}(4)$:

Of the states from which α sometimes or always starts only those states from which α sometimes or always terminates and from which some or every final state of α has a property Q are captured as initial states.

(Note: further constraints arise by considering negations.)

The above constraints and their negations are all constraints on states in which a program α is activated and hence allow us to formulate notions of correctness for a program α with respect to *a state* s and a postcondition Q. For the purposes of my investigation in this thesis I will not consider the negations of the constraints in $\mathbf{C_k}$ (for $\mathbf{k} = 1, 2, \ldots, 7$). Now to formulate notions of correctness for a program α with respect to *a precondition* P and a postcondition Q we need constraints on all or only some states *with a property* P in which a program α is activated. We could choose either:

$\mathbf{C_8}$ Initial state constraints

(1) Every state with property P in which α is activated satisfies a constraint in $\mathbf{C_k}$ (for $\mathbf{k} = 1, 2, \ldots, 7$).

or (2) Some or every state with property P in which α is activated satisfies a constraint in $\mathbf{C_k}$ (for $\mathbf{k} = 1, 2, \ldots, 7$).

or (3) Some states with property P in which α is activated satisfy a constraint in $\mathbf{C_k}$ (for $\mathbf{k}$

$= 1, 2, \ldots, 7$) and some other states with property P in which α is activated satisfy another constraint in $\mathbf{C_k}$ (for $\mathbf{k} = 1, 2, \ldots, 7$).

Of these only $\mathbf{C_8}(1)$ has been covered in the literature. Although $\mathbf{C_8}(2)$ and $\mathbf{C_8}(3)$ have not been investigated they may also be of interest. What is at issue is a notion of 'not being (completely) incorrect'; rather than a notion of 'correctness'. For instance, a programmer may find that for a program α, a precondition P and a postcondition Q, α is correct with respect to a subset P' of P, and Q. It seems that a notion of 'not being (completely) incorrect' would be useful to express such a relationship among α, P and Q. Such a notion formulated using $\mathbf{C_8}(2)$ could indicate the possibility of an error; while those formulated using $\mathbf{C_8}(3)$ could distinguish (say) possible errors concerning termination from those concerning final states in a particular context. For the purposes of this thesis I will not consider the third constraint.

The above constraints can be interpreted in terms of the two dual notions (demonic and angelic) of nondeterminism introduced in Chapter 1 (p 28, 29). Each constraint $\mathbf{C_k}(1)$ (for $\mathbf{k} = 1, 2, 3$) characterises, for a program α, a set of initial states s from which *every* execution of α establishes a particular property. Recall from Chapter 1 that this means these constraints refer to a demonic interpretation of nondeterminism. As a reference in subsequent sections I will call such constraints *demonic constraints.* On the other hand each constraint $\mathbf{C_k}(2)$ (for $\mathbf{k} = 1, 2, 3$) characterises, for a program α, a set of initial states s from which *at least one* execution of α establishes a particular property. Recall from Chapter 1 that this means these constraints refer to an angelic interpretation of nondeterminism. As a reference in subsequent sections I will call such constraints *angelic constraints.* Also any conjunct of the form '$\mathbf{C_i}(1)$ and $\mathbf{C_j}(1)$' (for $\mathbf{i}, \mathbf{j} = 1, 2, 3$) refers to a demonic interpretation of nondeterminism; while any conjunct of the form '$\mathbf{C_i}(2)$ and $\mathbf{C_j}(2)$' (for $\mathbf{i}, \mathbf{j} = 1, 2, 3$) (or '$\mathbf{C_i}(1)$ and $\mathbf{C_j}(2)$' (for $\mathbf{i}, \mathbf{j} = 1, 2, 3$)) refers to an angelic interpretation of nondeterminism.

So the 2 initial state constraints ($\mathbf{C_8}(1)$ and $\mathbf{C_8}(2)$) and the 26 constraints (in $\mathbf{C_k}$ for $\mathbf{k} = 1, 2, \ldots 7$) on states in which a program α is activated lead to 52 notions of correctness. Namely:

Notions for initialisation

Notion $\mathbf{N_1}(\mathbf{C_1}(\mathbf{x}))$: $\mathbf{C_8}(1)$ and $\mathbf{C_1}(\mathrm{x})$:

Every state s with property P is captured by constraint $\mathbf{C_1}(\mathrm{x})$

Notion $\mathbf{N_2}(\mathbf{C_1}(\mathbf{x}))$: $\mathbf{C_8}(2)$ and $\mathbf{C_1}(\mathrm{x})$:

Some or every state s with property P is captured by constraint $\mathbf{C_1}(\mathrm{x})$

where x = 1, 2.

Notions for termination

Notion $\mathbf{N_1}(\mathbf{C_2}(\mathbf{x}))$: $\mathbf{C_8}(1)$ and $\mathbf{C_2}(\mathrm{x})$:

Every state s with property P is captured by constraint $\mathbf{C_2}(\mathrm{x})$

Notion $\mathbf{N_2}(\mathbf{C_2}(\mathbf{x}))$: $\mathbf{C_8}(2)$ and $\mathbf{C_2}(\mathrm{x})$:

Some or every state s with property P is captured by constraint $\mathbf{C_2}(\mathrm{x})$

where x = 1, 2.

Notions for outcomes

Notion $\mathbf{N_1}(\mathbf{C_3}(\mathbf{x}))$: $\mathbf{C_8}(1)$ and $\mathbf{C_3}(\mathrm{x})$:

Every state s with property P is captured by constraint $\mathbf{C_3}(\mathrm{x})$

Notion $\mathbf{N_2}(\mathbf{C_3}(\mathbf{x}))$: $\mathbf{C_8}(2)$ and $\mathbf{C_3}(\mathrm{x})$:

Some or every state s with property P is captured by constraint $\mathbf{C_3}(\mathrm{x})$

where x = 1, 2.

Notions for initialisation and termination

Notion $\mathbf{N_1}(\mathbf{C_4}(\mathbf{x}))$: $\mathbf{C_8}(1)$ and $\mathbf{C_4}(\mathrm{x})$:

Every state s with property P is captured by constraint $\mathbf{C_4}(\mathrm{x})$

Notion $\mathbf{N_2}(\mathbf{C_4}(\mathbf{x}))$: $\mathbf{C_8}(2)$ and $\mathbf{C_4}(\mathrm{x})$:

Some or every state s with property P is captured by constraint $\mathbf{C_4}(\mathrm{x})$

where x = 1, 2, 3, 4.

Notions for initialisation and outcomes

Notion $\mathbf{N_1}(\mathbf{C_5}(\mathbf{x}))$: $\mathbf{C_8}(1)$ and $\mathbf{C_5}(\mathrm{x})$:

Every state s with property P is captured by constraint $\mathbf{C_5}(\mathrm{x})$

Notion $N_2(C_5(x))$: $C_8(2)$ and $C_5(x)$:

Some or every state s with property P is captured by constraint $C_5(x)$ where x = 1, 2, 3, 4.

Notions for termination and outcomes

Notion $N_1(C_6(x))$: $C_8(1)$ and $C_6(x)$:

Every state s with property P is captured by constraint $C_6(x)$

Notion $N_2(C_6(x))$: $C_8(2)$ and $C_6(x)$:

Some or every state s with property P is captured by constraint $C_6(x)$ where x = 1, 2, 3, 4.

Notions for initialisation, termination and outcomes

Notion $N_1(C_7(x))$: $C_8(1)$ and $C_7(x)$:

Every state s with property P is captured by constraint $C_7(x)$

Notion $N_2(C_7(x))$: $C_8(2)$ and $C_7(x)$:

Some or every state s with property P is captured by constraint $C_7(x)$ where x = 1, 2, 3, 4, 5, 6, 7, 8.

I now come to the second part of this section. Each of the notions of correctness formulated in the first part assumes the existence of final states because they are based on the input-output semantics of programs. However, this assumption is not always warranted because there are programs which are continuously operating, and ideally nonterminating (for example, operating systems). The normal behavioural pattern of such programs (or systems) is an arbitrarily long execution sequence. Since there are in general no final states we need to formulate notions of correctness to capture the 'state of nontermination'. The solution to this problem lies in the way in which we view a program. In this part I look at how the correctness of such programs can be specified.

In the preceding discussion we assumed that predicates are subsets of $\mathcal{S}$ and did not require an explicit erroneous state for nontermination. As a solution to the abovementioned problem Jacobs and Gries [1985] suggested the introduction of a special symbol to denote the 'state of

nontermination'. I will use the special symbol, '∞', to indicate the 'state of nontermination' and allow postconditions to contain ∞. (Jacobs and Gries [1985] used '$\perp$'.) Then treating ∞ as a possible 'final' state of a program it is naturally interesting to see whether we can obtain notions of program correctness (based on input-output semantics) which capture continuously operating programs as well. Now the set of all possible initial states is $\mathcal{S}$; while the set of all possible final states is $\mathcal{S} \bigcup \{\infty\}$.

The initialisation contraints $\mathbf{C_1}(1)$ and $\mathbf{C_1}(2)$ are the same as on p 72 because the set of all possible initial states for programs is unchanged. However, the termination/nontermination properties (B(1), B(2) and B(3) in Chapter 1, p 3) of a program α from a state s can now be described as follows. Let Q be the set of all final states of α from state s. Then

(B) **Termination/Nontermination**

either (1) $Q \subseteq \mathcal{S}$ which means α always terminates from s,

or (2) $Q \subseteq \mathcal{S} \bigcup \{\infty\}$ which means α sometimes terminates from s,

or (3) $Q = \{\infty\}$ which means α never terminates from s.

Therefore the termination constraints in $\mathbf{C_2}$ (p 72, 73) become:

either (1) The set of final states of α from s is a subset of $\mathcal{S}$.

or (2) The set of final states of α from s is a subset of $\mathcal{S} \bigcup \{\infty\}$.

The outcome contraints $\mathbf{C_3}(1)$ and $\mathbf{C_3}(2)$ are the same as on p 73 only now a property Q of final states can describe the state of nontermination as well.

Hence if '∞' denotes the state of nontermination we can define all the notions of correctness discussed in the first part of this section, only now using the modified termination and outcome constraints. Namely, $(\mathbf{N_j}(\mathbf{C_k}(\mathbf{x}))$ (for $\mathbf{j} = 1, 2$; $\mathbf{k} = 1, 2, 3$; $\mathbf{x} = 1, 2$); $\mathbf{N_j}(\mathbf{C_k}(\mathbf{x}))$ (for $\mathbf{j} = 1, 2$; $\mathbf{k} = 4, 5, 6$; $\mathbf{x} = 1, 2, 3, 4$)) $\mathbf{N_j}(\mathbf{C_7}(\mathbf{x}))$ (for $\mathbf{j} = 1, 2$; $\mathbf{x} = 1, 2, \ldots, 8$).

3.3 Formulating Notions of Correctness

Knowing (from §1) that a notion of correctness expresses a relationship between a program, a precondition and a postcondition, we enquire whether any of the statements $(\mathbf{N_j(C_k(x))}$ (for $\mathbf{j} = 1, 2$; $\mathbf{k} = 1, 2, 3$; $\mathbf{x} = 1, 2$); $\mathbf{N_j(C_k(x))}$ (for $\mathbf{j} = 1, 2$; $\mathbf{k} = 4, 5, 6$; $\mathbf{x} = 1, 2, 3, 4$); $\mathbf{N_j(C_7(x))}$ (for $\mathbf{j} = 1, 2$; $\mathbf{x} = 1, 2, \ldots, 8$)) expresses a meaningful notion of correctness. The aim of this section is to express notions of correctness set-theoretically.

I assume that:

(i) every state is an initial state for every program, (that is, that constraint $\mathbf{C_1}(1)$ holds),

and (ii) nontermination is equated with messy termination.

So I will only consider the notions $(\mathbf{N_j(C_k(x))}$ (for $\mathbf{j} = 1, 2$; $\mathbf{k} = 2, 3$; $\mathbf{x} = 1, 2$)) and the notions $(\mathbf{N_j(C_6(x))}$ (for $\mathbf{j} = 1, 2$; $\mathbf{x} = 1, 2, 3, 4$)) (introduced in §2). My investigation is divided into two parts: in the first the state space is $\mathcal{S}$ and I express notions of correctness set-theoretically using the apparatus defined in Chapter 2.5; in the second the set of all possible final states of programs is extended to $\mathcal{S} \bigcup \{\infty\}$ and postconditions may contain ∞.

To begin with let $\mathcal{S}$ be the state space. In Chapter 2.5, (where (i) and (ii) above were assumed), I introduced (p 60, 61) the two primitive functions *at* and *af* to classify executions. The first defines for a program α the set of initial states from which α always terminates; the second defines for a program α and a postcondition Q the set of initial states of α from which every final state of α has the property Q. These are defined using the auxillary functions *last* (which defines the final state of some execution of α from state s) and *out* (which defines the set of all final states of α from a state s). Recall from Chapter 2.5 (p 60, 61) that for any program α and any postcondition Q, $wlp(\alpha, Q) = af(\alpha, Q)$ and $wp(\alpha, Q) = at(\alpha) \bigcap af(\alpha, Q)$. Now taking $at(\alpha)$, $af(\alpha, Q)$ and P as primitives we obtain (by simple boolean combinations) the following characterisations:

Notions for termination

$\mathbf{N_1(C_2(1))}$	for every $s \in P$, α always terminates from s:	$P \subseteq at(\alpha)$
$\mathbf{N_1(C_2(2))}$	for every $s \in P$, α sometimes terminates from s:	$P \subseteq af(\alpha, \emptyset)'$
$\mathbf{N_2(C_2(1))}$	for some $s \in P$, α always terminates from s:	$P \cap at(\alpha) \neq \emptyset$
$\mathbf{N_2(C_2(2))}$	for some $s \in P$, α sometimes terminates from s:	$P \cap af(\alpha, \emptyset)' \neq \emptyset$

(Note that using the equivalences

$P \subseteq at(\alpha)$ iff $\sim(P \cap at(\alpha)' \neq \emptyset)$ and

$P \subseteq af(\alpha, \emptyset)'$ iff $\sim(P \cap af(\alpha, \emptyset) \neq \emptyset)$

we can easily express notions for the negation of termination constraints. For example, the negation of $\mathbf{N_2(C_2(1))}$:

for every $s \in P$, α sometimes does not terminate from s, or

for every $s \in P$, it is not true that α always terminates from s

is expressed by $\sim(P \cap at(\alpha) \neq \emptyset)$ (that is, $P \subseteq at(\alpha)'$).)

Notions for outcomes

$\mathbf{N_1(C_3(1))}$	for every $s \in P$, every final state of α from s has property Q: $P \subseteq af(\alpha, Q)$
$\mathbf{N_1(C_3(2))}$	for every $s \in P$, some final state of α from s has property Q: $P \subseteq af(\alpha, Q')'$
$\mathbf{N_2(C_3(1))}$	for some $s \in P$, every final state of α from s has property Q: $P \cap af(\alpha, Q) \neq \emptyset$
$\mathbf{N_2(C_3(2))}$	for some $s \in P$, some final state of α from s has property Q: $P \cap af(\alpha, Q')' \neq \emptyset$

(Note that using the equivalence $P \subseteq af(\alpha, Q)$ iff $\sim(P \cap af(\alpha, Q)' \neq \emptyset)$ we can easily express notions for the negations of the outcome constraints. For example, the negation of $\mathbf{N_1(C_3(2))}$:

for some $s \in P$, every final state of α from s does not have property Q, or

for some $s \in P$, it is not true that some final state of α from s has property Q.

is expressed by $\sim(P \subseteq af(\alpha, Q')')$ (that is, $P \cap af(\alpha, Q') \neq \emptyset$).)

Notions for termination and outcomes

$\mathbf{N_1(C_6(1))}$ for every $s \in P$, α always terminates from s and every final state of α from s has property Q:
$P \subseteq at(\alpha) \bigcap af(\alpha, Q)$

$\mathbf{N_1(C_6(2))}$ for every $s \in P$, α always terminates from s and some or every final state of α from s has property Q:
$P \subseteq at(\alpha) \bigcap af(\alpha, Q')'$

$\mathbf{N_1(C_6(3))}$ for every $s \in P$, α sometimes or always terminates from s and every final state of α from s has property Q:
$P \subseteq af(\alpha, \emptyset)' \bigcap af(\alpha, Q)$

$\mathbf{N_1(C_6(4))}$ for every $s \in P$, α sometimes or always terminates from s and some or every final state of α from s has property Q:
$P \subseteq af(\alpha, \emptyset)' \bigcap af(\alpha, Q')'$

$\mathbf{N_2(C_6(1))}$ for some $s \in P$, α always terminates from s and every final state of α from s has property Q:
$P \bigcap at(\alpha) \bigcap af(\alpha, Q) \neq \emptyset$

$\mathbf{N_2(C_6(2))}$ for some $s \in P$, α always terminates from s and some or every final state of α from s has property Q:
$P \bigcap at(\alpha) \bigcap af(\alpha, Q')' \neq \emptyset$

$\mathbf{N_2(C_6(3))}$ for some $s \in P$, α sometimes or always terminates from s and every final state of α from s has property Q:
$P \bigcap af(\alpha, \emptyset))' \bigcap af(\alpha, Q) \neq \emptyset$

$\mathbf{N_2(C_6(4))}$ for some $s \in P$, α sometimes or always terminates from s and some or every final state of α from s has property Q:
$P \bigcap af(\alpha, \emptyset)' \bigcap af(\alpha, Q')' \neq \emptyset$

(Note that using the equivalence $P \subseteq X \bigcap Y$ iff $\sim (P \bigcap (X \bigcap Y)' \neq \emptyset)$ we can easily express notions for the negations of the termination and outcomes constraints. For example, the negation of $\mathbf{N_1(C_6(1))}$:

'for some $s \in P$, it is not true that (α always does not terminate from s and Q is true in every final state of α from s)'
can be expressed as $\sim(P \subseteq (at\,(\alpha) \cap af\,(\alpha, Q)))$ (that is, $P \cap (at\,(\alpha) \cap af\,(\alpha, Q))' \neq \emptyset$).)

These notions are not independent. For example, for a deterministic program α, $out(\alpha, s)$ is a singleton for any state s. Thus

(1) $P \subseteq af(\alpha, Q')'$ iff $P \subseteq af(\alpha, \emptyset)' \cap af\,(\alpha, Q)$.

Proof Left to right: Take any $s \in af(\alpha, Q')'$. Then $out(\alpha, s) \cap Q \neq \emptyset$ so $out(\alpha, s) \neq \emptyset$ and $out(\alpha, s) \subseteq Q$ (since α is deterministic). Hence $s \in af(\alpha, \emptyset)' \cap af\,(\alpha, Q)$.
Right to left: Take any $s \in af(\alpha, \emptyset)' \cap af(\alpha, Q)$. Then $out(\alpha, s) \neq \emptyset$ and $out(\alpha, s) \subseteq Q$. But $out(\alpha, s)$ is a singleton; hence $out(\alpha, s) \cap Q \neq \emptyset$, showing $s \in af(\alpha, Q')'$. □

Now let $\mathcal{S} \cup \{\infty\}$ be the set of all possible final states and suppose postconditions may contain ∞. Recall from Chapter 2.5 (p 58) that *last* is a partial operation on exseqs: $last(\mathbf{x})$ is undefined if $\mathbf{x}$ is an infinite sequence. Hence *last* cannot be used to reason about the 'state of nontermination'. For this reason *af* cannot be used to express notions of correctness for nonterminating programs. To achieve this we require *last* to be total.

With the introduction of '∞' as a possible final state we can define a total operation $last_\infty$ (from *last* in Chapter 2.5 (p 58)) as follows: For any $\mathbf{x} \in \mathrm{Seq}(\mathcal{S})$, say $\mathbf{x} = (x_1, x_2, x_3, \ldots)$:

$$last_\infty(\mathbf{x}) = \begin{cases} x_n & \text{if } \mathbf{x} \in \mathcal{S}^+ \text{ and } \mathbf{x} = (x_1, x_2, \ldots, x_n) \\ \infty & \text{if } \mathbf{x} \in \mathcal{S}^\infty (= \mathcal{S}^\perp)\ . \end{cases}$$

Using this definition we can define a function af_∞: program × predicate ⟶ predicate by

$af_\infty(\alpha, Q) = \{s \mid out_\infty(\alpha, s) \subseteq Q\}$
where $out_\infty(\alpha, s) = \{last_\infty(\mathbf{x}) \mid \mathbf{x} \text{ is an exseq of } \alpha \text{ from } s\}$

(The subscript '∞' indicates that predicates may contain ∞.)

The characterisation of the notions of correctness for outcomes are the same as given on page 83, only using 'af_∞' instead of *af*.

3.4 Notions of Correctness in the Literature

My aim in this section is to identify which of the notions of correctness $\mathbf{N_j(C_k(x))}$ (for $\mathbf{j} = 1, 2$; $\mathbf{k} = 2, 3$; $\mathbf{x} = 1, 2$) and $\mathbf{N_j(C_6(x))}$ (for $\mathbf{j} = 1, 2$; $\mathbf{x} = 1, 2, 3, 4$) (expressed set-theoretically in terms of *at*, *af* (or af_∞) and P) have previously been investigated in the literature and relate them to the two dual notions (*demonic* and *angelic*) of nondeterminism (defined in Chapter 1, p 29). First, suppose $\mathcal{S}$ is the state space.

I Partial Correctness

Consider the notion of correctness $\mathbf{N_1(C_3(1))}$. $P \subseteq af(\alpha, Q)$ is true iff for every state s with property P if α starts from s, and if it terminates, it does so in a state with property Q. However, if $P \subseteq af(\alpha, \emptyset)$ (that is, if α never terminates from any state with property P) then $P \subseteq af(\alpha, Q)$ (by monotonicity of '$af(\alpha, -)$'). This means it is possible to establish the truth of $P \subseteq af(\alpha, Q)$ for a program which never terminates from any state in P. My conclusion is that for a state s to be in $af(\alpha, Q)$ it is *not* necessary for the execution of α from s to terminate at all. In this sense, $P \subseteq af(\alpha, Q)$ is a *partial specification*. However the terminating executions of α from s *must* terminate in a state with property Q. Recall from §2 (p 78) that $\mathbf{C_3(1)}$ is a demonic constraint, so $af(\alpha, Q)$ can be interpreted in terms of a demonic interpretation of nondeterminism.

Define for a program α, a precondition P and a postcondition Q,

(1) $\{P\}\alpha\{Q\}_{(Pd)}$ iff $P \subseteq af(\alpha, Q)$

where the subscripts 'P' and 'd' indicate respectively *partial* specification and *demonic* nondeterminism.

In fact, Hoare [1969] originally introduced the notion of correctness expressed by $\{P\}\alpha\{Q\}_{(Pd)}$ (which he called *conditional correctness*) in the framework of deterministic programs. It was later named *partial correctness* by Manna [1969] who also discussed it for nondeterministic programs. Another point to note is that (1) shows the connection between partial correctness and the Dijkstra/Gries predicate transformer '$wlp(\alpha, -)$'. In particular,

if in $\{P\}\alpha\{Q\}_{(Pd)}$ α and Q are fixed then (by (1) and definition of $wlp(\alpha, -)$ in Chapter 2.5, p 61) $wlp(\alpha, Q)$ is the least constrained predicate P such that the triple is true. (That is, $\{P\}\alpha\{Q\}_{(Pd)}$ iff $P \subseteq wlp(\alpha, Q)$.) The predicate $wlp(\alpha, Q)$ (or $af(\alpha, Q)$) is also called the *(demonic) weakest liberal precondition* of α with respect to Q in Jacobs and Gries [1985] and simply the *weakest liberal precondition* of α with respect to Q in Dijkstra ([1975], [1976]) and Dijkstra and Scholten [1990].

Now consider the notion of correctness $\mathbf{N_1(C_3(2))}$. $P \subseteq af(\alpha, Q')'$ is true iff for every state s with property P at least one execution of α from s terminates in a state which has property Q. $P \subseteq af(\alpha, Q')'$ therefore eliminates programs which never terminate from any state with property P (since if $P \subseteq af(\alpha, \emptyset)$ then $P \subseteq af(\alpha, Q')$ so $P \cap af(\alpha, Q')' = \emptyset$). However, for a state s to be in $af(\alpha, Q')'$ it is *not* necessary for every execution of α from s to terminate in a state in which Q is true. This means that $P \subseteq af(\alpha, Q')'$ is a *partial* specification. Recall from §2 (p 78) that $\mathbf{C_3(2)}$ is angelic constraint, so $af(\alpha, Q')'$ can be interpreted in terms of an angelic interpretation of nondeterminism.

Define for a program α, a precondition P and a postcondition Q:

(2) $\{P\}\alpha\{Q\}_{(Pa)}$ iff $P \subseteq af(\alpha, Q')'$

where the subscripts 'P' and 'a' indicate respectively *partial* specification and *angelic* nondeterminism.

In fact, the predicate $af(\alpha, Q')'$ is exactly the *(angelic) weakest liberal precondition* of α with respect to Q, $wlpa(\alpha, Q)$, in Jacobs and Gries [1985]. So $\{P\}\alpha\{Q\}_{(Pa)}$ iff $P \subseteq wlpa(\alpha, Q)$.

II **Total Correctness**

To get concepts of correctness which capture both partial correctness and termination I consider the termination and outcome constraints $\mathbf{C_6}$(x) (for x = 1, 2, 3, 4).

First, consider the notion of correctness $\mathbf{N_1(C_6(1))}$. Now $P \subseteq at(\alpha) \cap af(\alpha, Q)$ is true iff for every state s with property P *every* execution of α from s is guaranteed to terminate in

a state t which has property Q. Recall from §2 (p 78) that $\mathbf{C_6(1)}$ is a demonic constraint, so the predicate $af(\alpha) \bigcap af(\alpha, Q)$ can be interpreted in terms of a demonic interpretation of nondeterminism .

Define for a program α, a precondition P and a postcondition Q:

(3) $\{P\}\alpha\{Q\}_{(Td)}$ iff $P \subseteq at(\alpha) \bigcap af(\alpha, Q)$

where the subscripts 'T' and 'd' indicate respectively *total* specification and *demonic* nondeterminism.

$\{P\}\alpha\{Q\}_{(Td)}$ captures the notion of *total correctness* for nondeterministic programs introduced by Manna and Pnueli [1974] in an extension of Hoare's [1969] approach as well as the notion of *weak total correctness* for nondeterministic programs in Apt [1984]. Also, (3) shows the connection between total correctness and the Dijkstra/Gries predicate transformer '$wp(\alpha, -)$'. In particular, if in $\{P\}\alpha\{Q\}_{(Td)}$ α and Q are fixed then (by definition of $wp(\alpha, -)$ in Chapter 2.5, p 60) $wp(\alpha, Q)$ is the least constrained predicate P such that the triple is true. (That is, $\{P\}\alpha\{Q\}_{(Td)}$ iff $P \subseteq wp(\alpha, Q)$.) The predicate $wp(\alpha, Q)$ (or $at(\alpha) \bigcap af(\alpha, Q)$) is also called the *(demonic) total correctness weakest precondition* in Jacobs and Gries [1985] and simply *weakest precondition* of α with respect to Q in Dijkstra ([1975], [1976]) and Dijkstra and Scholten [1990].

Second, consider the notion of correctness $\mathbf{N_1(C_6(3))}$. Now $P \subseteq af(\alpha, \emptyset)' \bigcap af(\alpha, Q)$ is true iff for every state s with property P at least one execution of α from s is guaranteed to terminate in a state with property Q. Recall from §2 (p 78) that $\mathbf{C_6(3)}$ is an angelic constraint, so the predicate $af(\alpha, \emptyset)' \bigcap af(\alpha, Q)$ can be interpreted in terms of an angelic interpretation of nondeterminism.

Define for a program α, a precondition P and a postcondition Q:

(4) $\{P\}\alpha\{Q\}_{(Ta)}$ iff $P \subseteq af(\alpha, \emptyset)' \bigcap af(\alpha, Q)$

where the subscripts 'T' and 'a' indicate respectively *total* specification and *angelic* nondeterminism.

In fact, the predicate $af(\alpha,\emptyset)' \bigcap af(\alpha,Q)$ is consistent with the definition of the *(angelic) total correctness weakest precondition* of α with respect to Q, $wpa(\alpha,Q)$, in Jacobs and Gries [1985]. So $\{P\}\alpha\{Q\}_{(Ta)}$ iff $P \subseteq wpa(\alpha,Q)$.

In summary, for any program α, any precondition $P \subseteq \mathcal{S}$, and postcondition $Q \subseteq \mathcal{S}$ we can define:

Partial Correctness

$\mathbf{N_1(C_3(1))}$: $\{P\}\alpha\{Q\}_{(Pd)}$ iff $P \subseteq af(\alpha,Q)$

$\mathbf{N_1(C_3(2))}$: $\{P\}\alpha\{Q\}_{(Pa)}$ iff $P \subseteq af(\alpha,Q')'$

Total Correctness

$\mathbf{N_1(C_6(1))}$: $\{P\}\alpha\{Q\}_{(Td)}$ iff $P \subseteq at(\alpha) \bigcap af(\alpha,Q)$

$\mathbf{N_1(C_6(3))}$: $\{P\}\alpha\{Q\}_{(Ta)}$ iff $P \subseteq af(\alpha,\emptyset)' \bigcap af(\alpha,Q)$

(where the 'd' and 'a' in the subscripts denote respectively demonic and angelic nondeterminism). (Recall from Chapter 2.5 (p 60, 61) that $wlp(\alpha,Q) = af(\alpha,Q)$ and $wp(\alpha,Q) = at(\alpha) \bigcap af(\alpha,Q)$.)

Now let $\mathcal{S} \bigcup \{\infty\}$ be the set of all possible final states for programs. Here I consider the notions of correctness expressed in terms of postconditions that may contain ∞ and the function af_∞ (defined in §3, p 85).

There are three cases to consider. We could assume:

either (i) some postconditions Q contain ∞,

or (ii) every postcondition Q contains ∞,

or (iii) no postcondition Q contains ∞.

(i) **General Correctness**

Suppose postconditions Q may contain ∞. For any program α, any precondition P and any postcondition Q (which may contain ∞) consider the notion of correctness: $\mathbf{N_1(C_3(1))}$: $P \subseteq af_\infty(\alpha,Q)$. Now $P \subseteq af_\infty(\alpha,Q)$ is true iff for every state s with

property P, every execution of α from s sometimes terminates from s (but it is not guaranteed to) and every final state of α from s has property Q. The predicate $af_\infty(\alpha, Q)$ can be interpreted in terms of a demonic interpretation of nondeterminism (because $\mathbf{C_3(1)}$ is a demonic constraint (§2, p78)). In fact, $af_\infty(\alpha, Q)$ is consistent with the definition of the *(demonic) general correctness weakest precondition*, '$gwp(\alpha, Q)$', in Jacobs and Gries [1985].

Define for any program α, any precondition P and any postcondition Q (which may contain ∞):

(5) $\{P\}\alpha\{Q\}_{(Gd)}$ iff $P \subseteq af_\infty(\alpha, Q)$

where the subscripts 'G' and 'd' indicate respectively *general* specification and *demonic* nondeterminism. So $\{P\}\alpha\{Q\}_{(Gd)}$ iff $P \subseteq gwp(\alpha, Q)$.

Now for any program α, any precondition P and any postcondition Q (which may contain ∞) consider the notion of correctness: $\mathbf{N_1(C_3(2))}$: $P \subseteq af_\infty(\alpha, Q')'$. $P \subseteq af_\infty(\alpha, Q')'$ is true iff for every state s with property P, at least one execution of α from s either terminates from s in a final state with property Q or does not terminate at all (if $\infty \in Q$). The predicate $af_\infty(\alpha, Q')'$ can be interpreted in terms of an angelic interpretation of nondeterminism (because $\mathbf{C_3(2)}$ is an angelic constraint (§2, p 78)). In fact, $af_\infty(\alpha, Q')'$ is consistent with the definition of the *(angelic) general correctness weakest precondition*, '$gwpa(\alpha, Q)$', in Jacobs and Gries [1985].

Define for any program α, any precondition P and any postcondition Q (which may contain ∞):

(6) $\{P\}\alpha\{Q\}_{(Ga)}$ iff $P \subseteq af_\infty(\alpha, Q')'$

where the subscripts 'G' and 'a' indicate respectively *general* specification and *angelic* nondeterminism. So $\{P\}\alpha\{Q\}_{(Ga)}$ iff $P \subseteq gwpa(\alpha, Q)$.

(ii) **Partial Correctness**

Suppose postconditions Q must contain ∞. Then $\{P\}\alpha\{Q\}_{(Gd)}$ resembles the triple

$\{P\}\alpha\{Q\}_{(Pd)}$ (in (1)).

(In particular, take any $R = Q \bigcup \{\infty\}$ where $Q \subseteq \mathcal{S}$. Then $af_\infty(\alpha, R) = af_\infty(\alpha, Q \bigcup \{\infty\}) \supseteq af_\infty(\alpha, Q) \bigcup af_\infty(\alpha, \{\infty\}) = af(\alpha, Q) \bigcup af_\infty(\alpha, \{\infty\})$ (by monotonicity of af_∞). Hence if $P \subseteq af(\alpha, Q)$ then $P \subseteq af_\infty(\alpha, Q \bigcup \{\infty\})$. On the other hand, suppose $P \subseteq af_\infty(\alpha, Q \bigcup \{\infty\})$. Then $\forall s \in P$, $out_\infty(\alpha, s) \subseteq Q \bigcup \{\infty\}$, so $out(\alpha, s) \subseteq Q$, hence $P \subseteq af(\alpha, Q))$. Hence $\{P\}\alpha\{Q\}_{(Pd)}$ iff $P \subseteq af_\infty(\alpha, Q \bigcup \{\infty\})$ iff $P \subseteq wlp(\alpha, Q)$.) Similarly, $P \subseteq af_\infty(\alpha, Q')'$ captures the notion of partial correctness expressed in $\{P\}\alpha\{Q\}_{(Pa)}$ (in (2)).

Hence if postconditions must contain ∞ we can capture notions of partial correctness.

(iii) **Total correctness**

Suppose postconditions Q never contain ∞. Then $\{P\}\alpha\{Q\}_{(Gd)}$ resembles the triple $\{P\}\alpha\{Q\}_{(Td)}$ (in (3)).

(In particular, take any $R = Q \bigcap \mathcal{S}$ where $Q \subseteq \mathcal{S} \bigcup \{\infty\}$. Then $af_\infty(\alpha, R) = af_\infty(\alpha, Q \bigcap \mathcal{S}) = af_\infty(\alpha, \mathcal{S}) \bigcap af_\infty(\alpha, Q) = at(\alpha) \bigcap af_\infty(\alpha, Q)$ (by monotonicity of af_∞). Hence $\{P\}\alpha\{Q\}_{(Td)}$ iff $P \subseteq af_\infty(\alpha, Q \bigcap \mathcal{S})$ iff $P \subseteq wp(\alpha, Q)$.) Similarly, the triple $\{P\}\alpha\{Q\}_{(Ga)}$ resembles the triple $\{P\}\alpha\{Q\}_{(Ta)}$ (in (4)).

Hence if postconditions never contain ∞ we can capture notions of total correctness.

It is evident from this discussion that the inclusion of '∞' into the set-theoretic framework introduced in Chapter 2.5, provides a uniform framework in which to view and relate the various notions of correctness. In summary, for any program α, any precondition $P \subseteq \mathcal{S}$, and any postcondition $Q \subseteq \mathcal{S} \bigcup \{\infty\}$ we can define:

Partial Correctness

$\mathbf{N_1(C_3(1))}$: $\{P\}\alpha\{Q\}_{(Pd)}$ iff $P \subseteq af_\infty(\alpha, Q \bigcup \{\infty\})$

$\mathbf{N_1(C_3(2))}$: $\{P\}\alpha\{Q\}_{(Pa)}$ iff $P \subseteq af_\infty(\alpha, (Q \bigcup \{\infty\})')'$

Total Correctness

$\mathbf{N_1(C_3(1))}$: $\{P\}\alpha\{Q\}_{(Td)}$ iff $P \subseteq af_\infty(\alpha, Q \bigcap \mathcal{S})$

$\mathbf{N_1(C_3(2))}$: $\{P\}\alpha\{Q\}_{(Ta)}$ iff $P \subseteq af_\infty(\alpha, (Q \bigcap \mathcal{S})')'$

General Correctness

$\mathbf{N_1(C_3(1))}$: $\{P\}\alpha\{Q\}_{(Gd)}$ iff $P \subseteq af_\infty(\alpha, Q)$

$\mathbf{N_1(C_3(2))}$: $\{P\}\alpha\{Q\}_{(Ga)}$ iff $P \subseteq af_\infty(\alpha, Q')'$

(where the 'd' and 'a' in the subscripts denote respectively demonic and angelic nondeterminism).

I conclude this section by mentioning that although *general correctness* is also based on input-output semantics, it expresses *at least one* correctness property for continuously operating programs: what happens before and after an execution that terminates. However, it is of little use for reasoning about what happens *during* an execution. In a more recent development, Emerson [1990] utilises a temporal logic framework to provide an alternative formalisation of correctness based on next-state rather than input-output semantics. The formalisation defined is simple and provides an adequate foundation for reasoning about *all* correctness properties of continuously operating programs.

3.5 Correctness and Messy Termination

In this section I take the negation of 'terminates cleanly' to mean 'something went wrong'. The standard view (as adopted in §2, §3 and §4) of equating 'something went wrong' with 'nontermination' thus implicitly selects nontermination as the most important manner in which a program can 'go wrong'. But as indicated in Chapters 1 and 2 there are others. So even if we have notions of correctness which specifically cater for nontermination it would be useful to cater for messy termination as well — to cater for the other ways in which a program can 'go wrong'. My view in this section is: if a program α starts from a state s it either terminates or does not terminate from s. If it terminates it does so either cleanly or messily.

There are at least two ways to define (in the present context) notions of correctness for programs. The first is analogous to the approach adopted in the first part of §2. That is, I

could formulate all possible notions of correctness using constraints on a state s in which a program α is activated arising from

(i) the initialisation properties (A(1), A(2), A(3) in Chapter 1, p 3) of α from s,

and/or (ii) the clean-, messy- and nontermination properties (in list E Chapter 1, p 4) of α from s,

and/or (iii) the properties (D(1), D(2), D(3) in Chapter 1, p 3) of some or every final state of α from s.

The second is analogous to the approach adopted in the second part of §2. That is, I could introduce a special symbol '$\perp$' to denote the final state of messily terminating executions and allows postconditions to contain $\perp$. Then messy termination can be modelled by having the exseq for that execution ending in a special symbol '$\perp$', with $\perp \notin \mathcal{S}$. So $Seq(\mathcal{S}) = \mathcal{S}^+ \bigcup \mathcal{S}^\perp \bigcup \mathcal{S}^\infty$, where $\mathcal{S}^\perp$ denotes the set of all finite nonempty sequences with $\perp$ as the last component and elements of $\mathcal{S}$ as the first and (if any) intermediate components.

I choose the latter. As in §2 my investigation is divided into two parts: in the first part, I only consider the final states of terminating executions (these include cleanly- and messily terminating executions); in the second part I also consider the 'state of nontermination.'

Let $\mathcal{S} \bigcup \{\perp\}$ be the set of final states of programs. Now the definition of *last* on exseqs (Chapter 2.5, p 58) becomes

$$last_\perp(\mathbf{x}) = \begin{cases} \mathbf{x_n} & \text{if } \mathbf{x} \in \mathcal{S}^+ \bigcup \mathcal{S}^\perp \text{ and } \mathbf{x} = (x_1, \ldots x_n) \\ \text{undefined} & \text{if } \mathbf{x} \in \mathcal{S}^\infty \end{cases}$$

Using $last_\perp$ I can define a function $af_\perp$: programs $\times$ predicates $\longrightarrow$ predicates by

$$af_\perp(\alpha, Q) = \{s \mid out_\perp(\alpha, s) \subseteq Q\}$$
where $out_\perp(\alpha, s) = \{last_\perp(\mathbf{x}) \mid \mathbf{x} \text{ is an exseq of } \alpha \text{ from } s\}$.

(The subscript $\perp$ indicates that predicates may contain $\perp$.)

For any predicate Q containing $\perp$, $P \subseteq af_{\perp}(\alpha, Q)$ captures the notion of partial correctness expressed in $\{P\}\alpha\{Q\}_{(Pd)}$. Analogous reasoning to that in §4 (p 90, 91) for predicates which contain ∞, only replacing '∞' with '$\perp$' can be used to justify this statement. Similarly, $P \subseteq af_{\perp}(\alpha, Q')'$ captures the notion of partial correctness expressed in $\{P\}\alpha\{Q\}_{(Pa)}$. Note that for $Q \subseteq \mathcal{S}$, $P \subseteq af_{\perp}(\alpha, Q')'$ is a suitable characterisation of correctness to capture the error of messy termination. In fact, this is precisely Blikle's [1981] notion of *global correctness* introduced for this purpose. It is also called *possible correctness* in Hoare [1978].

However the preservation of the meaning of total correctness in the presence of messy termination is not so simple. Dijkstra's ([1976] p 16) definition of the notion of $wp(\alpha, Q)$ is:

> We shall use the notation $wp(\alpha, Q)$ to denote the weakest precondition for the initial state of the system such that activation of α is guaranteed to lead to a properly terminating activity leaving the system in a final state satisfying the postcondition Q.

It seems that the particular notion of correctness Dijkstra ([1975], [1976]) had in mind is one which guarantees that a program always terminates *cleanly* (that is, never terminates messily and never goes into an endless unproductive loop) and produces the correct result.

Now the fundamental question is: If the effects of messy termination are taken into account, does

$$\{P\}\alpha\{Q\}_{(Td)} \text{ iff } P \subseteq at(\alpha) \bigcap af(\alpha, Q)$$

capture the Dijkstra/Gries total correctness for any program α and postcondition $Q \subseteq \mathcal{S}$? My approach to answering this question follows (and occasionally adapts) that suggested in Harel [1979a].

The key point (as observed in Harel [1979a] p 59) is that the definition of a notion of total correctness for a program depends on the particular execution method one has in mind. Recall the four methods described in Chapter 1 **Table 6**,

(a) Depth-first execution method

(b) Depth-first execution method with backtracking

(c) Breadth-first execution method

(d) Breadth-first execution method with backtracking

My aim now is to define (using the primitive functions at and af) notions of total correctness dependent on these execution methods. For a program α to be totally correct with respect to a precondition P and a postcondition Q (which may contain $\perp$) we require α to be guaranteed to produce a state $t \in \mathcal{S}$ from every $s \in P$ and $Q \bigcap \mathcal{S}$ to be true in every final state of α from every $s \in P$.

In the context of this section (where nontermination is different from messy termination) 'terminates' means 'terminates cleanly or messily'. So

- $at(\alpha)$ eliminates states from which α sometimes or always does not terminate.
- $af(\alpha, \mathcal{S})$ eliminates states from which α sometimes or always terminates messily.
- $af(\alpha, \emptyset)'$ includes all initial states from which α terminates at least once.
- If $\perp \in Q$ then $af(\alpha, Q)$ captures the initial states of terminating executions of α with final state in Q, and
- if $\perp \notin Q$ then $af(\alpha, Q)$ captures the initial states of cleanly terminating executions of α with final state in Q.

From Chapter 1 **Table 7** I conclude the following:

(a) Using execution method **(a)** a program α activated in a state s is guaranteed to produce a result only if α always terminates cleanly from s (that is, if $s \in at(\alpha)$ and $s \in af(\alpha, \mathcal{S})$).

(b) Using execution method **(b)** a program α activated in a state s is guaranteed to produce a result only if α sometimes terminates cleanly from s and α never does not terminate from s (that is, only if $s \in af(\alpha, \emptyset)'$ and $s \in at(\alpha)$).

(c) Using execution method **(c)** a program α activated in a state s is guaranteed to produce a state $t \in \mathcal{S}$ as its result only if α sometimes terminates cleanly from s and α never terminates messily from s (that is, only if $s \in af(\alpha, \emptyset)'$ and $s \in af(\alpha, \mathcal{S})$).

(d) Using execution method **(d)** a program α activated in a state s is guaranteed to produce a state $t \in \mathcal{S}$ as its result only if α sometimes terminates cleanly from s (that is, only if $s \in af(\alpha, \emptyset)'$).

Consider any program α, any precondition P and any postcondition $Q \subseteq \mathcal{S} \cup \{\infty\}$. Then we can define a notion of total correctness $\{P\}\alpha\{Q\}_{\mathbf{x}}$ for each execution method $\mathbf{x}$ (where $\mathbf{x} =$ **(a)** - **(d)**) as follows:

(1) $\{P\}\alpha\{Q\}_{\mathbf{(a)}}$ iff $P \subseteq at(\alpha) \cap af(\alpha, Q \cap \mathcal{S})$
$\{P\}\alpha\{Q\}_{\mathbf{(b)}}$ iff $P \subseteq af(\alpha, \emptyset)' \cap at(\alpha) \cap af(\alpha, Q)$
$\{P\}\alpha\{Q\}_{\mathbf{(c)}}$ iff $P \subseteq af(\alpha, \emptyset)' \cap af(\alpha, Q \cap \mathcal{S})$
$\{P\}\alpha\{Q\}_{\mathbf{(d)}}$ iff $P \subseteq af(\alpha, \emptyset)' \cap af(\alpha, Q)$

From (1) we can define four *independent* notions of weakest precondition of a program. Namely:

(2) $wp_{\mathbf{(a)}}(\alpha, Q) = at(\alpha) \cap af(\alpha, Q \cap \mathcal{S})$
$wp_{\mathbf{(b)}}(\alpha, Q) = af(\alpha, \emptyset)' \cap at(\alpha) \cap af(\alpha, Q)$
$wp_{\mathbf{(c)}}(\alpha, Q) = af(\alpha, \emptyset)' \cap af(\alpha, Q \cap \mathcal{S})$
$wp_{\mathbf{(d)}}(\alpha, Q) = af(\alpha, \emptyset)' \cap af(\alpha, Q)$

where $wp_{\mathbf{x}}(\alpha, Q)$ (for $\mathbf{x} =$ **(a)** - **(d)**) defines the weakest precondition of α with respect to Q for execution method $\mathbf{x}$. Note that for execution method $\mathbf{x}$,

$\{P\}\alpha\{Q\}_{\mathbf{x}}$ iff $P \subseteq wp_{\mathbf{x}}(\alpha, Q)$

and $wp_{\mathbf{x}}(\alpha, Q)$ denotes the set of all those initial states s from which execution of α *using execution method* $\mathbf{x}$ is guaranteed to terminate cleanly in a state in which Q is true. Therefore each $wp_{\mathbf{x}}(\alpha, Q)$ satisfies Dijkstra's ([1975], [1976]) definition of weakest precondition.

This raises the following question: Does Dijkstra's notion of weakest precondition depend on a particular execution method? To answer this question I need to determine which of the three notions of weakest precondition satisfies Dijkstra's healthiness properties and is also consistent with the way in which Dijkstra's guarded command language is defined. Instead of giving a complete analysis of the notions of weakest precondition $wp_{\mathbf{x}}(\alpha, Q)$ (for $\mathbf{x} = \mathbf{(a)}$ - $\mathbf{(d)}$) (as done in Harel [1979a]), I will simply by a process of elimination show that only one of these four notions, namely $wp_{\mathbf{(a)}}(\alpha, Q)$, can be consistent with Dijkstra's definition of weakest precondition.

In particular, for each of $wp_{\mathbf{(b)}}$, $wp_{\mathbf{(c)}}$ and $wp_{\mathbf{(d)}}$ there are programs α and β such that the Dijkstra/Gries law for sequential composition (Chapter 2.2 (8.3), p 40) is not satisfied.

Take α: if $true \longrightarrow x := 1 \ [] \ true \longrightarrow x := 2$ fi and postcondition Q: $true$.

(b) For execution method **(b)**, take β: if $x = 1 \longrightarrow x := x$ fi. Then

$$wp_{\mathbf{(b)}}(\alpha;\beta, Q) = af(\alpha;\beta, \emptyset)' \cap at(\alpha;\beta) \cap af(\alpha;\beta, Q) = \mathcal{S}$$

since $\alpha;\beta$ terminates cleanly at least once, namely when α assigns 1 to x, $\alpha;\beta$ never loops forever and using execution method **(b)**, Q is true in every final state produced by $\alpha;\beta$. On the other hand,
$wp_{\mathbf{(b)}}(\alpha, wp_{\mathbf{(b)}}(\beta, Q)) = af(\alpha, \emptyset)' \cap at(\alpha) \cap af(\alpha, wp_{\mathbf{(b)}}(\beta, Q))$ and for every state s, $out(\alpha, s) = \{t \mid x = 1 \text{ in state } t\} \cup \{t \mid x = 2 \text{ in state } t\}$ but $\{t \mid x = 2 \text{ in state } t\} \subseteq \{t \mid out(\beta, t) = \{\perp\}\} \not\subseteq \{t \mid out(\beta, t) \subseteq Q\} = af(\beta, Q)$. Thus there is no state s such that $out(\alpha, s) \subseteq af(\beta, Q)$, so $af(\alpha, af(\beta, Q)) = \emptyset$; hence $af(\alpha, wp_{\mathbf{(b)}}(\beta, Q)) = \emptyset$. Hence $wp_{\mathbf{(b)}}(\alpha, wp_{\mathbf{(b)}}(\beta, Q)) = \emptyset$ which shows that $wp_{\mathbf{(b)}}(\alpha;\beta, Q)) \not\subseteq wp_{\mathbf{(b)}}(\alpha, wp_{\mathbf{(b)}}(\beta, Q))$. This result, I believe supports my contention (Chapter 1, p 27) that execution method **(b)** is not suitable for defining a notion of total correctness consistent with that of Dijkstra.

(c) For execution method **(c)**, take γ: do $x = 1 \longrightarrow x := x$ od. Then

$$wp_{(\mathbf{c})}(\alpha;\gamma,Q) = af(\alpha;\gamma,\emptyset)' \cap af(\alpha;\gamma,Q \cap \mathcal{S}) = \mathcal{S}$$

since $\alpha;\gamma$ terminates cleanly at least once, namely when α assigns 1 to x, $\alpha;\gamma$ never terminates messily and using execution method **(c)**, Q is true in every final state produced by $\alpha;\gamma$. On the other hand,
$wp_{(\mathbf{c})}(\alpha, wp_{(\mathbf{c})}(\gamma,Q)) = af(\alpha,\emptyset)' \cap af(\alpha, wp_{(\mathbf{c})}(\gamma,Q) \cap \mathcal{S})$ and for every state s, $out(\alpha,s)$ $= \{t \mid x = 1 \text{ in state } t\} \cup \{t \mid x = 2 \text{ in state } t\}$ but $\{t \mid x = 2 \text{ in state } t\} \subseteq \{t \mid \mathbf{infin}(\gamma,t) \neq \emptyset\} = af(\gamma,\emptyset)$. Thus there is no state s such that $out(\alpha,s) \subseteq af(\gamma,\emptyset)'$, so $af(\alpha,\ af(\gamma,\emptyset)') = \emptyset$; hence $af(\alpha,(wp_{(\mathbf{c})}(\gamma,Q) \cap \mathcal{S})) = \emptyset$. Thus $wp_{(\mathbf{c})}(\alpha, wp_{(\mathbf{c})}(\gamma,Q)) = \emptyset$ which shows that $wp_{(\mathbf{c})}(\alpha;\gamma,Q)) \not\subseteq wp_{(\mathbf{c})}(\alpha, wp_{(\mathbf{c})}(\gamma,Q))$.

(d) For execution method **(d)**, take either β or γ. Analogous reasoning shows that $wp_{(\mathbf{d})}(\alpha;\beta,Q)) \not\subseteq wp_{(\mathbf{d})}(\alpha, wp_{(\mathbf{d})}(\beta,Q))$ and $wp_{(\mathbf{d})}(\alpha;\gamma,Q)) \not\subseteq wp_{(\mathbf{d})}(\alpha, wp_{(\mathbf{d})}(\gamma,Q))$.

My conclusion is that execution method **(a)** gives rise to a definition of weakest precondition consistent with Dijkstra's ([1975], [1976]) definition of weakest precondition in the sense that it captures the same meaning of correctness as that of Dijkstra. It is therefore best suited for determining whether a program is totally correct. Recall from Chapter 1 execution method **(a)** gives rise to a demonic interpretation of nondeterminism. This means that the notion of weakest precondition formulated in Dijkstra ([1975], [1976]) presupposes execution method **(a)** and hence a demonic interpretation of nondeterminism. In fact, Harel's [1979a] approach, via tree traversal algorithms, leads to the same result. It is interesting to note that this is also essentially shown in the independent investigations of de Bakker [1976], Plotkin (as described in de Roever [1976]) and Hoare [1978]. I have therefore answered both my earlier questions (p 94, 97).

Note also that execution method **(d)** gives rise to a definition of weakest precondition which resembles '$wpa(\alpha,Q)$' (defined in §3 (5)). Recall from Chapter 1 execution method **(d)** gives rise to a demonic interpretation of nondeterminism. Therefore even if the effects of

messy termination are taken into account $wpa(\alpha, Q)$ is suitable for determining the total correctness of a program under an angelic interpretation of nondeterminism.

In summary, for any program α, any precondition $P \subseteq \mathcal{S}$ and postcondition $Q \subseteq \mathcal{S} \cup \{\perp\}$, we can define:

Partial Correctness

$\mathbf{N_1(C_3(1))}$: $\{P\}\alpha\{Q\}_{(Pd)}$ iff $P \subseteq af_\perp(\alpha, Q \cup \{\perp\})$

$\mathbf{N_1(C_3(2))}$: $\{P\}\alpha\{Q\}_{(Pa)}$ iff $P \subseteq af_\perp(\alpha, (Q \cup \{\perp\})')'$

Total Correctness

$\mathbf{N_1(C_3(1))}$: $\{P\}\alpha\{Q\}_{(Td)}$ iff $P \subseteq af_\perp(\alpha, Q \cap \mathcal{S})$

$\mathbf{N_1(C_3(2))}$: $\{P\}\alpha\{Q\}_{(Ta)}$ iff $P \subseteq af_\perp(\alpha, (Q \cap \mathcal{S})')'$

(where the 'd' and 'a' in the subscripts indicate respectively demonic and angelic nondeterminism).

I now come to the second part of this section. Let the set of all possible final states be $\mathcal{S} \cup \{\infty\} \cup \{\perp\}$ and assume postconditions may contain ∞ and/or $\perp$. The definition of *last* on exseqs (Chapter 2.5, p 58) becomes

$$last_u(\mathbf{x}) = \begin{cases} \mathbf{x_n} & \text{if } \mathbf{x} \in \mathcal{S}^+ \cup \mathcal{S}^\perp \text{ and } \mathbf{x} = (x_1, \ldots x_n) \\ \infty & \text{if } \mathbf{x} \in \mathcal{S}^\infty \end{cases}$$

Using $last_u$ I can define a function af_u: programs $\times$ predicates $\longrightarrow$ predicates by

$af_u(\alpha, Q) = \{s \mid out_u(\alpha, s) \subseteq Q\}$

where $out_u(\alpha, s) = \{last_u(\mathbf{x}) \mid \mathbf{x} \text{ is an exseq of } \alpha \text{ from } s\}$.

(The subscript u indicates that postconditions may contain $\perp$ and/or ∞.)
Now there **four** cases to consider for a postcondition $Q \subseteq \mathcal{S} \cup \{\infty\} \cup \{\perp\}$.

(i) Suppose $\perp \in Q$. Now:

- if $\infty \in Q$ then $P \subseteq af_u(\alpha, Q)$ and $P \subseteq af_u(\alpha, Q')'$ capture notions of partial correctness in §4 (1) and (2), and
- if $\infty \notin Q$ then none of the notions of correctness mentioned in §4 are captured.

(ii) Suppose $\perp \notin Q$. Now:

- if $\infty \in Q$ then $P \subseteq af_u(\alpha, Q)$ and $P \subseteq af_u(\alpha, Q')'$ capture notions of general correctness in §4 (5) and (6), and
- if $\infty \notin Q$ then $P \subseteq af_u(\alpha, Q)$ and $P \subseteq af_u(\alpha, Q')'$ capture notions of total correctness in §4 (3) and (4)

Analogous reasoning to that (in §4) for postconditions which may contain ∞ and that (in the first part of this section) for postconditions which may contain $\perp$ can be used to establish these claims.

In summary, for any program α, any precondition $P \subseteq \mathcal{S}$ and any postcondition $Q \subseteq \mathcal{S} \cup \{\infty\} \cup \{\perp\}$ we can define:

Partial Correctness

$\mathbf{N_1(C_3(1))}$: $\{P\}\alpha\{Q\}_{(Pd)}$ iff $P \subseteq af_u(\alpha, Q \cup \{\infty\} \cup \{\perp\})$

$\mathbf{N_1(C_3(2))}$: $\{P\}\alpha\{Q\}_{(Pa)}$ iff $P \subseteq af_u(\alpha, (Q \cup \{\infty\} \cup \{\perp\})')'$

Total Correctness

$\mathbf{N_1(C_3(1))}$: $\{P\}\alpha\{Q\}_{(Td)}$ iff $P \subseteq af_u(\alpha, Q \cap \mathcal{S})$

$\mathbf{N_1(C_3(2))}$: $\{P\}\alpha\{Q\}_{(Ta)}$ iff $P \subseteq af_u(\alpha, (Q \cap \mathcal{S})')'$

General Correctness

$\mathbf{N_1(C_3(1))}$: $\{P\}\alpha\{Q\}_{(Gd)}$ iff $P \subseteq af_u(\alpha, Q)$

$\mathbf{N_1(C_3(2))}$: $\{P\}\alpha\{Q\}_{(Ga)}$ iff $P \subseteq af_u(\alpha, Q')'$

Global Correctness

$\mathbf{N_1(C_3(2))}$: $\{P\}\alpha\{Q\}_{(G)}$ iff $P \subseteq af_u(\alpha, (Q \cap \mathcal{S})')'$

(where the '*d*' and '*a*' in the subscripts indicate respectively demonic and angelic nondeterminism).

3.6 A Comparison of Notions of Correctness

In §3, §4 and §5 I characterised and identified notions of partial, total-, global- and general correctness. In this section I discuss the relationships between these concepts of correctness

using the definitions obtained in the second part of §5.

(1) **Theorem** *For any program* α, *any precondition* $P \subseteq \mathcal{S}$ *and any postcondition* $Q \subseteq \mathcal{S} \cup \{\infty\} \cup \{\perp\}$,

(a) $\{P\}\alpha\{Q\}_{(Td)}$ *implies* $\{P\}\alpha\{Q\}_{(Pd)}$

(b) $\{P\}\alpha\{Q\}_{(Td)}$ *implies* $\{P\}\alpha\{Q\}_{(G)}$

(c) $\{P\}\alpha\{Q\}_{(Td)}$ *implies* $\{P\}\alpha\{Q\}_{(Gd)}$

(d) $\{P\}\alpha\{Q\}_{(Gd)}$ *implies* $\{P\}\alpha\{Q\}_{(Pd)}$

Analogous relationships hold for the notions of correctness using angelic constraints.

Proof I will consider the demonic case.

(a) For any $Q \subseteq \mathcal{S} \cup \{\infty\} \cup \{\perp\}$, $Q \cap \mathcal{S} \subseteq Q \subseteq Q \cup \{\infty\} \cup \{\perp\}$. So by monotonicity of af_u, $af_u(\alpha, Q \cap \mathcal{S}) \subseteq af_u(\alpha, Q \cup \{\infty\} \cup \{\perp\})$. Hence if $P \subseteq af_u(\alpha, Q \cap \mathcal{S})$ then $P \subseteq af_u(\alpha, Q \cup \{\infty\} \cup \{\perp\})$ (that is, $\{P\}\alpha\{Q\}_{(Td)}$ implies $\{P\}\alpha\{Q\}_{(Pd)}$). However, this implication cannot be reversed. For example, consider the program α : *abort* and any postcondition $Q \subseteq \mathcal{S} \cup \{\infty\} \cup \{\perp\}$. Now $af_u(abort, Q) = \mathcal{S}$ but $af_u(abort, Q \cap \mathcal{S}) = \emptyset$. So $\{P\}\alpha\{Q\}_{(Pd)}$ holds but $\{P\}\alpha\{Q\}_{(Td)}$ does not hold.

(b) Suppose $\{P\}\alpha\{Q\}_{(Td)}$ holds. Then $P \subseteq af_u(\alpha, Q \cap \mathcal{S})$. Thus $\forall s \in P$, $out_u(\alpha, s) \subseteq Q \cap \mathcal{S}$ and hence $\forall s \in P$, $out(\alpha, s) \cap (Q \cap \mathcal{S}) \neq \emptyset$. Thus $P \subseteq af_u(\alpha, (Q \cap \mathcal{S})')'$, so $\{P\}\alpha\{Q\}_{(G)}$ holds. Hence $\{P\}\alpha\{Q\}_{(Td)}$ implies $\{P\}\alpha\{Q\}_{(G)}$. However, this implication cannot be reversed. For example consider the program α: if *true* $\longrightarrow$ *abort* [] *true* $\longrightarrow$ *skip* fi and postcondition $Q = \mathcal{S}$. Now $\forall s \in \mathcal{S}$, $out_u(\alpha, s) = \{s, \perp\}$, so $af_u(\alpha, (Q \cap \mathcal{S})')' = \mathcal{S}$ but $af_u(\alpha, Q \cap \mathcal{S}) = \emptyset$. So for any $P \subseteq \mathcal{S}$, $P \subseteq af_u(\alpha, (Q \cap \mathcal{S})')'$, but $P \not\subseteq af_u(\alpha, Q \cap \mathcal{S})$. Hence $\{P\}\alpha\{Q\}_{(G)}$ holds but $\{P\}\alpha\{Q\}_{(Td)}$ does not.

(c) For any predicate $Q \subseteq \mathcal{S} \cup \{\infty\} \cup \{\perp\}$, $Q \cap \mathcal{S} \subseteq Q \cap (\mathcal{S} \cup \{\infty\})$. So by the monotonicity of af_u, $af_u(\alpha, Q \cap \mathcal{S}) \subseteq af_u(\alpha, Q \cap (\mathcal{S} \cup \{\infty\}))$. Hence if $P \subseteq af_u(\alpha, Q \cap \mathcal{S})$ then $P \subseteq af_u(\alpha, Q \cap (\mathcal{S} \cup \{\infty\}))$ (that is, $\{P\}\alpha\{Q\}_{(Td)}$ implies $\{P\}\alpha\{Q\}_{(Gd)}$). However, the reverse implication does not hold. For example, consider the program α: do *true* $\longrightarrow$ *skip* od and postcondition $Q = \mathcal{S} \cup \{\infty\} \cup \{\perp\}$. Now $\forall s \in \mathcal{S}$,

$out_u(\alpha, s) = \{\infty\}$, so $af_u(\ _{\circ}Q \cap \mathcal{S}) = \emptyset$ but $af_u(\alpha, Q \cap (\mathcal{S} \cup \{\infty\})) = \mathcal{S}$. Hence for any $P \subseteq \mathcal{S}$, $P \subseteq af_u(\alpha, Q \cap (\mathcal{S} \cup \{\infty\}))$ but $P \not\subseteq af_u(\alpha, Q \cap \mathcal{S})$. So $\{P\}\alpha\{Q\}_{(Gd)}$ holds but $\{P\}\alpha\{Q\}_{(Td)}$ does not.

(d) For any predicate $Q \subseteq \mathcal{S} \cup \{\infty\} \cup \{\perp\}$, $Q \cap (\mathcal{S} \cup \{\infty\}) \subseteq Q \cup \{\infty\} \cup \{\perp\}$. So by the monotonicity of $af_u(\alpha, Q \cap (\mathcal{S} \cup \{\infty\})) \subseteq af_u(\alpha, (Q \cup \{\infty\} \cup \{\perp\}))$. Hence if $P \subseteq af_u(\alpha, Q \cap (\mathcal{S} \cup \{\infty\}))$ then $P \subseteq af_u(\alpha, (Q \cup \{\infty\} \cup \{\perp\}))$ (that is, $\{P\}\alpha\{Q\}_{(Gd)}$ implies $\{P\}\alpha\{Q\}_{(Pd)}$). However, the reverse implication does not hold. For example, consider the program α: *abort* and postcondition $Q = \mathcal{S}$. Now $\forall s \in \mathcal{S}$, $out_u(\alpha, s) = \{\perp\}$, so $af_\infty(\alpha, (Q \cup \{\infty\} \cup \{\perp\})) = \mathcal{S}$ but $af_u(\alpha, Q \cap (\mathcal{S} \cup \{\infty\})) = \emptyset$. Thus $\forall P \subseteq \mathcal{S}$, $P \subseteq af_\infty(\alpha, (Q \cup \{\infty\} \cup \{\perp\}))$ but $P \not\subseteq af_u(\alpha, Q \cap (\mathcal{S} \cup \{\infty\}))$. So $\{P\}\alpha\{Q\}_{(Pd)}$ holds but $\{P\}\alpha\{Q\}_{(Gd)}$ does not. □

For any nondeterministic program α, any precondition and any postcondition, the relationships between partial-, total- , global- and general correctness are given in **Figure 6**,

$$\begin{array}{ccccccccc}
\{P\}\alpha\{Q\}_{(Td)} & \Longrightarrow & \{P\}\alpha\{Q\}_{(Pd)} & & \{P\}\alpha\{Q\}_{(Ta)} & \Longrightarrow & \{P\}\alpha\{Q\}_{(Pa)} \\
\Downarrow & \searrow & \Uparrow & & \Downarrow & \searrow & \Uparrow \\
\{P\}\alpha\{Q\}_{(G)} & & \{P\}\alpha\{Q\}_{(Gd)} & & \{P\}\alpha\{Q\}_{(G)} & & \{P\}\alpha\{Q\}_{(Ga)}
\end{array}$$

Figure 6

where '$\Longrightarrow$' means 'implies'. For example, and arrow from $\{P\}\alpha\{Q\}_{(Td)}$ to $\{P\}\alpha\{Q\}_{(Pd)}$ means '$\{P\}\alpha\{Q\}_{(Td)}$ implies $\{P\}\alpha\{Q\}_{(Pd)}$' interpreted as 'if α is totally correct with respect to P and Q, then α is partially correct with respect to P and Q.'

Note that, in general,

(i) $\{P\}\alpha\{Q\}_{(G)}$ does not imply $\{P\}\alpha\{Q\}_{(Gd)}$

(ii) $\{P\}\alpha\{Q\}_{(Gd)}$ does not imply $\{P\}\alpha\{Q\}_{(G)}$

(iii) $\{P\}\alpha\{Q\}_{(G)}$ does not imply $\{P\}\alpha\{Q\}_{(Pd)}$

(iv) $\{P\}\alpha\{Q\}_{(Pd)}$ does not imply $\{P\}\alpha\{Q\}_{(G)}$

As a counterexample for (i) consider the program α: if $true \longrightarrow abort \;[]\; true \longrightarrow skip$ fi and any postcondition $Q = \mathcal{S}$. Now $\forall s \in \mathcal{S}$, $out_u(\alpha, s) = \{s, \perp\}$, so $af_u(\alpha, (Q \bigcap \mathcal{S})')' = \mathcal{S}$ but $af_u(\alpha, (Q \bigcap (\mathcal{S} \bigcup \{\infty\})) = \emptyset$. Hence $\{P\}\alpha\{Q\}_{(G)}$ holds but $\{P\}\alpha\{Q\}_{(Gd)}$ does not hold.

As a counterexample for (ii) consider the program β : do $true \longrightarrow skip$ od. Now $\forall s \in \mathcal{S}$ $out_u(\alpha, s) = \{\infty\}$, so $af_u(\beta, (Q \bigcap (\mathcal{S} \bigcup \{\infty\})) = \mathcal{S}$ but $af_u(\beta, (Q \bigcap \mathcal{S})')' = \emptyset$. Hence $\{P\}\beta\{Q\}_{(Gd)}$ holds but $\{P\}\beta\{Q\}_{(G)}$ does not hold.

As a counterexample for (iii) consider the program α: if $true \longrightarrow x := 1 \;[]\; true \longrightarrow x := 2$ fi and the postcondition $Q = \{s \mid x$ has value 1 in state $s\}$. Now $\forall s \in \mathcal{S}$, $out_u(\alpha, s) = \{s \mid x$ has value 1 in state $s\} \bigcup \{s \mid x$ has value 2 in state $s\}$. So $af_u(\alpha, Q \bigcup \{\infty\} \bigcup \{\perp\}) = \emptyset$ but $af_u(\alpha, (Q \bigcap \mathcal{S})')' = \mathcal{S}$. Hence $\{P\}\alpha\{Q\}_{(G)}$ holds but $\{P\}\alpha\{Q\}_{(Pd)}$ does not hold.

As a counterexample for (iv) consider the program β: *abort* and any postcondition Q. Now $\forall s \in \mathcal{S}$, $out_u(\alpha, s) = \{\perp\}$, so $af_u(\alpha, Q \bigcup \{\infty\} \bigcup \{\perp\}) = \mathcal{S}$ but $af_u(\alpha, (Q \bigcap \mathcal{S})')' = \emptyset$. Hence $\{P\}\alpha\{Q\}_{(Pd)}$ holds but $\{P\}\alpha\{Q\}_{(G)}$ does not hold.

In this chapter I have approached the definition of different notions of correctness from one point of view by determining constraints on initial states for programs. Noting that the specification, written $\{P\}\alpha\{Q\}$ (p 68), of a notion of correctness for a program α expresses a relationship between a set P of initial states for α and a set Q of final states for α, notions of correctness can also be defined by determining constraints on the final states for programs.

Chapter 4

A Relational Model

This chapter is concerned with a model of programs in which all reasoning about programs is based upon their input-output behaviour. I will critically examine two ideas:

(i) That programs can be modelled as binary relations on states; operations on programs as operations on relations and the calculus of weakest preconditions as the calculus of binary relations.

(ii) That it may be possible to do so equationally.

The intuition behind idea (i) is that an input-output pair (s, t) of states is related by a program α iff program α *sometimes* reaches final state t from state s. Since programs, in general, are nondeterministic, any intial state s may be related by a program to a number of possible final states and the meaning of a program is given by the collection of all its possible input-output pairs. This collection is a binary (input-output) relation on states.

The initial attraction of the relational model lies in facts such as the following.

(a) There already exists a calculus of binary relations (such as that of Tarski [1941]).

(b) Details of implementations are suppressed.

(c) Proofs are often simple.

However, it has several shortcomings.

(a) It abstracts from the many properties of the computations performed in transforming states. (Blikle [1981]) For example, it permits a study of correctness and termination (as I show in §4 and §5) but not of properties of the progressive behaviour of programs. (For example, it fails to handle assertions like 'the variable x assumes the value 0 at some point during the computation' or 'a property P holds throughout every execution of a program from some state s'.)

(b) Continuously operating programs where no output at all is expected cannot be analysed. (For example, operating systems.) (Emerson [1990])

(c) It falters on an important point: Gries [1981] (8.3) is not satisfied when composition of programs is interpreted as composition of relations. (Gordon [1989a])

(d) Some relational operations (for example, complementation) have rather artificial interpretations in the context of programs.

I will conclude in §5 that a more sophisticated model is required to capture our intuition about the behaviour of nondeterministic programs (including those which are continuously operating).

In order to investigate idea (i) it is necessary to find a relational model for nondeterministic programs. In §2 I apply the analysis of representation methods made in Chapter 1 to the relational model, and draw the attention of the reader to twenty-four methods of representing programs as binary (input-output) relations on states. In §3 I investigate the suitability of the various representation methods for (each of) the notion(s) of correctness identified in Chapter 3.4. However, I demonstrate in §4 that models based on such representation methods do not comfortably accommodate Dijkstra's weakest precondition semantics, and so we do not obtain some perspicuous *standard* model for nondeterministic programs by just considering binary relations. We need to specialise to a certain set of relations which I call *execution relations*. Using execution relations I verify, in §5, all the formulae (except that for the iterative command) in Dijkstra's algebra of weakest preconditions. Of those verified I conclude that all, except the law for the assignment command, can be established equationally.

4.1 Background

Before investigating a relational model it is essential to have a clear picture of the developments which motivated the research into the relevance of binary relations to theoretical computer science.

One of the earliest approaches to capturing the semantics of a program α was to consider the final state in which α terminates as a *function* of the initial state from which α was started. The '*functional approach*' to programming has been extensively followed in, for example, Hoare and Lauer [1974] — but it has a few shortcomings. One is that the final state, considered as a function of the initial state, may not be defined for all states in the state space, because there may be some initial state s such that either α fails to start from s, or if α starts from s it may terminate messily from s or it may not terminate from s.

To deal with such situations there are at least three methods of representing a program as a function of states. Recall (from Chapter 1, p 8) the two options **R1**(1) and **R1**(3) for representing initialisation properties of programs: either (1) only states from which α always starts are included as initial states; or (3) all states in which a program α is activated are included as initial states. For the latter option recall also (from Chapter 1, p 11) the options **R2**(4)(a) and **R2**(4)(b) for representing execution properties of programs: either (a) one special symbol could be used to simultaneously denote the final state of nonterminating and messily terminating executions; or (b) two special symbols could be used to denote respectively the final state of nonterminating and messily terminating executions. There are then three alternative ways of representing a program as a function on states. Namely, we could choose either **R1**(1); [**R1**(3) and **R2**(4)(a)] or [**R1**(3) and **R2**(4)(b)]. First, if **R1**(1) is adopted, a program α could be viewed as a *partial function* $\alpha : \mathcal{S} \longrightarrow \mathcal{S}$. This method (which corresponds to representation method $\mathbf{R_1(1)}$ in Chapter 1 **Table 3**) has the disadvantage that it cannot distinguish between a program which can start from s but not terminate cleanly, and one which cannot start from s at all. (Of course making this distinction may not be important in a particular context.) Second, if options **R1**(3) and **R2**(4)(a) are chosen then a program can be viewed as a *total function* $\alpha : \mathcal{S} \longrightarrow \mathcal{S} \cup \{\perp\}$

where '$\alpha(s) = \{\perp\}$' means 'α is not defined at s' which can be taken as saying 'α does not start from s', 'α terminates messily from s' or 'α does not terminate from s'. Third, if options **R1**(3) and **R2**(4)(b) are chosen then a program can be viewed as a *total function* $\alpha : \mathcal{S} \longrightarrow \mathcal{S} \cup \{\perp, \infty\}$ where '$\alpha(s) = \{\perp\}$' means 'α is not defined at s' which can be taken as saying 'α does not start from s' or 'α terminates messily from s', and '$\alpha(s) = \{\infty\}$' means 'α does not terminate from s'. However, the latter two approaches (which correspond respectively to representation methods $\mathbf{R_3(7)}$ and $\mathbf{R_3(8)}$ in Chapter 1 **Table 3**) have the disadvantage that they destroy the homogeneity of the state space. (Dijkstra and Scholten ([1990] p 126).

Perhaps the main shortcoming of viewing a program as a (partial- or total) function is that it caters only for *deterministic programs* — nondeterministic programs do not fit naturally into such a functional framework. This has led to a number of alternative mathematical frameworks for treating nondeterministic programs. One such approach is to introduce functions from states to subsets of states — that is, *multifunctions* (or *multiple-valued functions* ((Enderton [1977] p 44), (Manes and Arbib [1986] p 21–26)). By analogy with the discussion in Chapter 1 concerning representation methods for nondeterministic programs we could enumerate twenty-four representation methods in terms of multifunctions corresponding to the twenty-four methods (in terms of pictorial representations) listed in Chapter 1 **Table 3**. For example, if representation method $\mathbf{R_1(1)}$, or $\mathbf{R_2(1)}$ is adopted, a program α could be viewed as a *partial multifunction* $\alpha : \mathcal{S} \longrightarrow \mathcal{P}(\mathcal{S})$. If representation method $\mathbf{R_3(7)}$: **R1**(3) and **R2**(4)(a) is adopted, a program α can be viewed as a *total multifunction* $\alpha : \mathcal{S} \longrightarrow \mathcal{P}(\mathcal{S} \cup \{\perp\})$, where for every $s \in \mathcal{S}$, $\alpha(s) \neq \emptyset$ and $\alpha(s) \subseteq \mathcal{S} \cup \{\perp\}$.

Another approach is to view a program as a binary (input-output) relation on states. On such an approach, an initial state and a final state for a program are treated at the same level.

Now I summarise the basic operations on relations that I need in the sequel. A binary relation R with respect to a set S is a subset of the Cartesian product $S \times S$. The following operations will be used:

(a)	Null relation	$\emptyset$ (which is the empty subset of $S \times S$)
	Identity relation	$I = \{(x,x) \mid x \in S\}$
	Universal relation	$S \times S$.
(b)	Conversion	$R^{\smile} = \{(x,y) \mid (y,x) \in R\}$
	Complementation	$R' = \{(x,y) \mid (x,y) \notin R\}$
(c)	Composition	$R;S = \{(x,y) \mid (\exists z \in S)[(x,z) \in R \text{ and } (z,y) \in S]\}$
	Union	$R \cup S = \{(x,y) \mid (x,y) \in R \text{ or } (x,y) \in S\}$
	Intersection	$R \cap S = \{(x,y) \mid (x,y) \in R \text{ and } (x,y) \in S\}$
(d)	Infinite union	$\bigcup_{n \geq 0} R^n = \{(x,y) \mid (\exists n \geq 0)[(x,y) \in R^n\}$
		where $R^0 = I$ and $R^{n+1} = R^n;R$ for $n \geq 0$.

The equational laws satisfied by these operations are described in a famous and seminal paper of Tarski [1941].

A variant of composition is the application of a relation to a set. One such operation for a binary relation R and a set Q is defined by

(e) $R : Q = \{x \mid (\exists y)[(x,y) \in R \text{ and } y \in Q]\}$

Brink ([1978], [1981]) called this operation the *Peirce product* (after CS Peirce, who introduced it in 1870).

The following theorem gives the arithmetical facts concerning ':' that will be used in §4 and §5.

(1) **Theorem** *For any relations $R, T \subseteq S \times S$ and any sets $P, Q \subseteq S$ the following hold:*

(i) $(R;T) : Q = R : (T : Q)$

(ii) $R : (P \cup Q) = R : P \cup R : Q$

(iii) $(R \cup T) : Q = R : Q \cup T : Q$

(iv) $I : Q = Q$

(v) $\emptyset : Q = \emptyset$

(vi) $R^{\smile} : (R : Q)' \subseteq Q'$

(vii) If $P \subseteq Q$ then $R : P \subseteq R : Q$

(viii) $R : (P \bigcap Q) \subseteq R : P \bigcap R : Q$

(ix) If $\bigcup_{i \in I} P_i$ exists, so does $\bigcup_{i \in I} R : P_i$, and $\bigcup_{i \in I} R : P_i = R : (\bigcup_{i \in I} P_i)$

(x) $R : \emptyset = \emptyset$

Proof By way of example, I prove (i) — the others are similar.

$$\begin{aligned}
(R;T) : P &= \{x \mid (\exists y)[(x,y) \in R;T \text{ and } y \in P]\} \\
&= \{x \mid (\exists y)[(\exists z)[(x,z) \in R \text{ and } (z,y) \in T] \text{ and } y \in P]\} \\
&= \{x \mid (\exists y)(\exists z)[[(x,z) \in R \text{ and } (z,y) \in T] \text{ and } y \in P]\} \\
&= \{x \mid (\exists z)(\exists y)[[(x,z) \in R \text{ and } (z,y) \in T] \text{ and } y \in P]\} \\
&= \{x \mid (\exists z)[(x,z) \in R \text{ and } (\exists y)[(z,y) \in T \text{ and } y \in P]]\} \\
&= \{x \mid (\exists z)[(x,z) \in R \text{ and } z \in T : P]\} \\
&= R : (T : P)
\end{aligned}$$

□

(Property (vii) is sometimes described by saying Peirce product is monotone with respect to $\subseteq$.)

There are other *set-forming operations on relations.* For example,

(f) Domain $\quad \text{dom } R = \{x \mid (\exists y)[(x,y) \in R]\} = R : S$

Range $\quad \text{ran } R = \{y \mid (\exists x)[(x,y) \in R]\} = R^{\smile} : Q$

A binary relation $R \subseteq \mathcal{S} \times \mathcal{S}$ is called a *total relation* if dom $R = \mathcal{S}$ and a *partial relation* if dom $R \subseteq \mathcal{S}$. Tarski's [1941] calculus of binary relations provides a formalism for representing programs as binary relations. This is as follows.

Let $\mathcal{S}$ be the set of states of the computer, called the *state space.* A *predicate* P is here (as in Chapters 2, 3) not interpreted as a first-order logic formula but rather thought of as *being* the set of states in which P is true. Thus a predicate is simply a subset of the state space $\mathcal{S}$. Operations on predicates include unions, intersections and complements.

Programs are not assumed to be deterministic and are represented as binary relations over $\mathcal{S}$. In this chapter, as in Blikle [1977], $[\alpha]$ denotes the binary (input-output) relation corresponding to a program α.

The programs *skip* and *abort* are represented respectively by the identity relation, I and the null relation, $\emptyset$. For two programs α and β, sequential composition $\alpha;\beta$ is represented by the composition $[\alpha];[\beta]$ of the corresponding relations $[\alpha]$ and $[\beta]$ (that is, $[\alpha;\beta] = [\alpha];[\beta]$). Since we are dealing with nondeterministic programs, there is for any programs α and β a program $\alpha \cup \beta$, called *nondeterministic choice* (Hoare [1978]) which performs either α or β, without it being known in advance which one. This program is represented by the set-theoretic union of the relations $[\alpha]$ and $[\beta]$ (that is, $[\alpha \cup \beta] = [\alpha] \cup [\beta]$).

In order to define the relation corresponding to the simple assignment statement, a state is viewed as a vector of values of (all) program variables. Then the assignment statement is regarded as a program which relates two states differing in only one position, namely the final state has in that position the value assigned to the corresponding variable by the assignment command. That is, for any state $s \in \mathcal{S}$, $s = (s(1), s(2), s(3), \ldots)$ where for each $j = 1, 2, 3, \ldots$, $s(j)$ denotes the value of the program variable X_j in state s. Then for a program variable X_k and a well-defined expression e, the binary relation corresponding to the simple assignment statement '$X_k := e$' is
$[X_k := e] = \{(s,t) \mid t(k)$ is the value of e evaluated in state s and for $j \neq k$ $s(j) = t(j)\}$.

To represent the remaining two fundamental constructions of programs — the IF and DO statements — in the calculus of binary relations, two ideas are used. First, for a predicate B (used as a guard), a program $B?$ (called *test* B) is used. This program *skips* if the input state s is in B but *aborts* if s is not in B. The corresponding relation is $[B?] = \{(s,s) \mid s \in B\}$. The properties of this relation that will be used in §5 are given in the following Lemma.

(2) **Lemma** *For any program α and any predicates B and Q,*

(a) $[B?] : Q = B \cap Q$.

(b) $[\alpha];[B?] = \{(x,y) \mid (x,y) \in [\alpha]$ and $y \in B\}$.

(c) $[B?];[\alpha] = \{(x,y) \mid x \in B \text{ and } (x,y) \in [\alpha]\}$.

Proof

(a) $[B?] : Q$

$= \{x \mid (\exists y)[(x,y) \in [B?] \text{ and } y \in Q]\} = \{x \mid (\exists y)[x = y \text{ and } y \in Q]\} = B \bigcap Q.$

(b) $[\alpha];[B?]$

$= \{(x,y) \mid (\exists z)[(x,z) \in [\alpha] \text{ and } (z,y) \in [B?]\}$

$= \{(x,y) \mid (\exists z)[(x,z) \in [\alpha] \text{ and } z = y \text{ and } z \in B]\}$

$= \{(x,y) \mid (x,y) \in [\alpha] \text{ and } y \in B\}.$

(c) Similar to (b). □

Second, a program whose execution consists of executing some program α a finite number of times is also used. This program, denoted by α^*, is such that

$$\alpha^* = skip \bigcup \alpha \bigcup \alpha;\alpha \bigcup \ldots \bigcup \alpha^n \bigcup \ldots$$

$$= \bigcup_{n \geq 0} \alpha^n$$

where $\alpha^0 = skip$ and $\alpha^{n+1} = \alpha;\alpha^n$ for $n \geq 0$.

Lemma *For any program* α, $[\alpha^n] = [\alpha]^n$, *for* $n \geq 0$.

Proof By induction on n.

For $n = 0$, $[\alpha]^0 = I$ and $[\alpha^0] = [skip] = I$. Assume as induction hypothesis that $[\alpha]^n = [\alpha^n]$ for some n. Then

$$\begin{aligned} [\alpha]^{n+1} &= [\alpha]^n;[\alpha] && \text{(by definition)} \\ &= [\alpha^n];[\alpha] && \text{(by induction hypothesis)} \\ &= [\alpha^n;\alpha] && \text{(by definition)} \\ &= [\alpha^{n+1}] && \text{(by definition)} \end{aligned}$$

□

The corresponding binary (input-output) relation is $[\alpha^*] = [\alpha]^*$, where $[\alpha]^* = I \bigcup [\alpha] \bigcup [\alpha];[\alpha] \bigcup \ldots \bigcup [\alpha]^n \bigcup \ldots = \bigcup_{n \geq 0} [\alpha]^n$. This is the reflexive transitive closure of a relation. It is called the *Kleene closure* of the relation (Kozen ([1980] p 356)), after SC Kleene, who contributed extensively to the theory of so-called *regular*

expressions (in Kleene [1956]). Then for any programs α_1 and α_2 and any predicates B_1 and B_2, the program '$if B_1 \rightarrow \alpha_1 \,[]\, B_2 \rightarrow \alpha_2 fi$' can be written as '$B_1?;\alpha_1 \cup B_2?;\alpha_2$' with '$[B_1?];[\alpha_1] \cup [B_2?];[\alpha_2]$' as corresponding binary (input-output) relation. It is also easy to define the binary relations corresponding to the two special forms of IF. Namely, $[if\ B\ do\ \alpha] = [B?];[\alpha]$ and $[if\ B\ do\ \alpha\ else\ \beta\ fi] = [B?];[\alpha] \cup [\neg B?];[\beta]$.

Finally, for any program α and any predicate B, the program 'while B do α' can be written as '$(B?;\alpha)^*;(\neg B?)$' (or '$(if\ B\ do\ \alpha)^*;(\neg B?)$') with '$([B?];[\alpha])^*;[\neg B?]$' as corresponding binary (input-output) relation.

To define the semantics of (atomic or compound) programs, a program is treated as a certain operator on predicates, namely the weakest precondition of a program α with respect to a given postcondition $Q \subseteq \mathcal{S}$. That is,

(3) $wp(\alpha, Q) = \{s \mid (\exists t \in \mathcal{S})[(s,t) \in [\alpha]]\} \cap \{s \mid (\forall t \in \mathcal{S})[(s,t) \in [\alpha] \Rightarrow t \in Q]\}$

The reader will have no difficulty in recognising that the first set is interpreted simply as the Peirce product $[\alpha] : \mathcal{S}$ introduced in (e). The other set is equationally definable from Peirce product and complementation (from the Boolean algebra) as $([\alpha] : Q')'$. (This is in fact another one of Peirce's original operations, which he called *involution* (Peirce [1870]).) Thus for any program α and any postcondition $Q \subseteq \mathcal{S}$,

(4) $wp(\alpha, Q) = [\alpha] : \mathcal{S} \cap ([\alpha] : Q')'$

Also a precondition of a program α with respect to a postcondition $Q \subseteq \mathcal{S}$ is

(5) $\{s \mid (\exists t \in \mathcal{S})[(s,t) \in [\alpha] \text{ and } t \in Q]\} = [\alpha] : Q$.

(If $Q = \mathcal{S}$ this is the domain of $[\alpha]$, written dom$[\alpha]$.)

A postcondition of a program α with respect to a precondition $P \subseteq \mathcal{S}$ is

(6) $\{t \mid (\exists s \in \mathcal{S})[(s,t) \in [\alpha] \text{ and } s \in P]\} = [\alpha]^\smile : P$.

(If $P = \mathcal{S}$ this is the range of $[\alpha]$, written ran$[\alpha]$.)

These ideas have been exploited by many. I give a brief overview of instances in the literature where a relational approach is adopted. In the early seventies, binary relations were used to model recursive programs by for example, de Bakker and de Roever [1973], Hitchcock and

Park [1973] and de Bakker ([1978], [1980]). An overview of the calculi of relations is presented in Sanderson's [1981] book, which serves as an excellent introduction to the relational approach to computation. Several methods of relational representations of (nondeterministic) programs have been proposed (Hoare and Lauer [1974], Plotkin [1976], Wand [1977], Smyth [1978]) and investigated (in for example, de Roever [1976], Blikle ([1977], [1981]), de Bakker [1978], Harel ([1979a], [1979b]), Guerreiro ([1980], [1981], [1982]), Jacobs and Gries [1985], as well as Holt [1991]). Jacobs and Gries [1985] provided an overview of many methods of relational representations. A relational approach has been used as a unifying framework in which to view and relate various notions of correctness (by for example, Blikle ([1977], [1981]), Jacobs and Gries [1985] and Holt [1991]). Nelson [1989] presents an excellent discussion of the methods of relational representation corresponding to the notions of partial- and total correctness. The use of binary relations to model nondeterministic programs has also figured in logics of programs, such as dynamic logic (Parikh [1981], Harel [1979a]). (Logics of programs are abstract models of programming languages which reason *about* programs. In this context, a calculus of relations forms part of the mathematical framework to reason about programs — programs are not actually written within the framework of the calculus of relations, as in a relational approach to computation.)

An application of Tarski's [1941] calculus of relations was introduced by Hoare and He Jifeng [1987] in the specification of an abstract data type (that is, a data structure and a set of operations defined on it). They extended the notion of weakest precondition to that of *weakest prespecification* of a program and a postcondition. Unlike the usual relational approaches to program semantics, both programs and predicates are viewed as relations.

One respect in which the relational approach is successful is that proofs are often simple. Gordon [1989a] proved the formulae of Dijkstra's algebra of weakest preconditions in the automated reasoning environment HOL.

4.2 Representation and Execution Methods

In Chapter 1 I introduced twenty-four methods of representing (nondeterministic) programs: $\mathbf{R_j(n)}$ (for $\mathbf{j} = 1, 2, 3$; $\mathbf{n} = 1, 2, \ldots, 8$) as listed in **Table 3**. By analogy with the analysis of the twenty-four representation methods of (nondeterministic) programs in Chapter 1 **Table 3**, I will, in the first part of this section, enumerate twenty-four methods of representing programs in terms of binary (input-output) relations between a set of initial states and a set of outcomes. Recall also the four execution methods described in Chapter 1 **Table 7**, namely **(a)** depth-first; **(b)** depth-first with backtracking; **(c)** breadth-first; and **(d)** breadth-first with backtracking execution methods. I investigate, in the second part of this section, how representation methods correspond to execution methods.

A binary (input-output) relation $[\alpha]$ associated with a program α is a collection of pairs (s, t) of states such that t is a possible outcome of executing α from initial state s. The set of initial states for a program α is represented by the domain of the associated relation $[\alpha]$, written dom$[\alpha]$ and the set of outcomes is represented by the range of $[\alpha]$, written ran$[\alpha]$. Therefore a relational representation of an execution of a program is the abstraction of the initial state and the outcome(s) of the execution. Recall from Chapter 1 that the possible outcomes produced by cleanly terminating executions are states. The range of a binary relation associated with such an execution is a subset of the state space. The 'outcomes' of nonterminating and messily terminating executions are not states. So the range of a binary relation associated with such an execution must include special state(s). I will now show how different relational representation methods arise by putting some constraints on the domain and the range of relations used and the expected relation between the initial and final states to be represented.

First it must be decided which initialisation properties of a program α from a state s are to be represented in a relational model. Within the context of this chapter, the three choices (considered in Chapter 1, p 8) are:

R1 Initialisation Property Representations

either (1) Only states from which a program α always starts are related to outcomes.

or (2) All states from which a program α sometimes or always starts are related to outcomes.

or (3) All states in which a program α is activated are related to outcomes.

Options **R1**(1) and **R1**(2) correspond to modelling programs by partial relations because from some states there may be no *outcome* at all, though the program can be *activated* in the state (that is, the state is a member of the state space). Option **R1**(3) corresponds to restricting relations to total relations because each state is related to at least one outcome in the sense that something must happen when a program is activated in a state. (Note that the relational representation methods in the literature do not distinguish between options **R1**(1) and **R1**(2). Instead they are combined into one option: Some states in which a program α is activated are not related to outcomes.)

Second it must be decided which kind(s) of executions (from the list in Chapter 1, p 9) are to be represented in a relational model. Within the context of this chapter, the eight choices (considered in Chapter 1, p 9, 10) are:

R2 Execution Property Representations

either (1) Only cleanly terminating executions are represented. There are two possibilities for representation, namely,

(a) Only programs α with
$dom[\alpha] = \{s \mid \alpha$ always terminates cleanly from state $s\}$, and
$ran[\alpha] \subseteq \mathcal{S}$ are represented.

or (b) Only programs α with
$dom[\alpha] = \{s \mid \alpha$ sometimes or always terminates cleanly from state $s\}$, and
$ran[\alpha] \subseteq \mathcal{S}$ are represented.

or (2) Only terminating executions are represented (including both cleanly- and messily terminating executions. There are two possibilities for representation, namely,

(a) Only programs α with
$dom[\alpha] = \{s \mid \alpha$ always terminates from state $s\}$, and
$ran[\alpha] \subseteq \mathcal{S} \bigcup \{\perp\}$ (where '$\perp$' denotes the final state of messily terminating executions) are represented.

or (b) Only programs α with
$dom[\alpha] = \{s \mid \alpha$ sometimes or always terminates from state $s\}$, and
$ran[\alpha] \subseteq \mathcal{S} \bigcup \{\perp\}$ (where '$\perp$' denotes the final state of messily terminating executions) are represented.

or (3) Only cleanly terminating- and nonterminating executions are represented. There are two possibilities for representation, namely,

(a) Only programs α with
$dom[\alpha] = \{s \mid \alpha$ always terminates cleanly and/or does not terminate from state $s\}$, and $ran[\alpha] \subseteq \mathcal{S} \bigcup \{\infty\}$ (where '∞' denotes the final state of nonterminating executions) are represented.

or (b) Only programs α with
$dom[\alpha] = \{s \mid \alpha$ sometimes or always terminates cleanly and/or does not terminate from state $s\}$, and $ran[\alpha] \subseteq \mathcal{S} \bigcup \{\infty\}$ (where '∞' denotes the final state of nonterminating executions) are represented.

or (4) All kinds of executions are represented. There are two possibilities for representation, namely,

(a) $ran[\alpha] \subseteq \mathcal{S} \bigcup \{\perp\}$, where '$\perp$' denotes simultaneously the final state of messily terminating executions and nonterminating executions.

or (b) $ran[\alpha] \subseteq \mathcal{S} \bigcup \{\perp\} \bigcup \{\infty\}$, where '$\perp$' and '$\infty$' denote respectively the final states of messily terminating executions and nonterminating executions.

So for any program α, and any initial state s,

- '$(s,t) \in [\alpha]$' means that α sometimes or always terminates cleanly from state s in final state t.

- if '$\perp$' denotes simultaneously the final state of messily terminating and nonterminating executions then '$(s,\perp) \in [\alpha]$' means that α sometimes or always does not terminate cleanly from state s; otherwise '$(s,\perp) \in [\alpha]$' means that α sometimes or always terminates messily from state s.

- '$(s,\infty) \in [\alpha]$' means that α sometimes or always does not terminate from state s.

The three choices (**R1**(1), **R1**(2), **R1**(3) on p 115) for the set of initial states and the eight choices (**R2**(i)(x) on p 115, 116, where i = 1, 2, 3, 4; x = a, b) for executions lead to twenty-four different (relational) representation methods for programs. Namely:

Representation Method $\mathbf{R_j(1)}$: **R1**(j) and **R2**(1)(a)
Representation Method $\mathbf{R_j(2)}$: **R1**(j) and **R2**(1)(b)
Representation Method $\mathbf{R_j(3)}$: **R1**(j) and **R2**(2)(a)
Representation Method $\mathbf{R_j(4)}$: **R1**(j) and **R2**(2)(b)
Representation Method $\mathbf{R_j(5)}$: **R1**(j) and **R2**(3)(a)
Representation Method $\mathbf{R_j(6)}$: **R1**(j) and **R2**(3)(b)
Representation Method $\mathbf{R_j(7)}$: **R1**(j) and **R2**(4)(a)
Representation Method $\mathbf{R_j(8)}$: **R1**(j) and **R2**(4)(b)

where j = 1, 2, 3.

(Note that the notation used here is the same as in Chapter 1 **Table 3**, only now (implicitly) denoting relational representation methods for programs by refering to the options for initialisation property representations on p 115 and the options for execution property representations on p 115, 116.)

In $\mathbf{R_3(1)}$ the conjunct '**R1**(3) and **R2**(1)(a)' means that from every state every execution of the program under consideration is supposed to terminate cleanly. Thus such a model cannot represent either messy termination or nontermination. Likewise, in $\mathbf{R_3(3)}$ the conjunct '**R1**(3) and **R2**(2)(a)' means that there is no way of representing nontermination, and in

$\mathbf{R_3(5)}$ the conjunct '**R1**(3) and **R2**(3)(a)' means that there is no direct representation of messy termination.

In $\mathbf{R_3(2)}$ the conjunct '**R1**(3) and **R2**(1)(b)' means that from every state every execution of the program under consideration is supposed to sometimes or always terminate cleanly. Thus such a model cannot represent an execution which never terminates cleanly. Likewise, in $\mathbf{R_3(4)}$ the conjunct '**R1**(3) and **R2**(2)(b)' means that there is no way of representing an execution which never terminates, and in $\mathbf{R_3(6)}$ the conjunct '**R1**(3) and **R2**(3)(b)' means that there is no direct representation of an execution which always terminates messiy.

Recall from Chapter 1 (p 14) the representation with respect to a representation method $\mathbf{R_j(n)}$ (where $\mathbf{j} = 1, 2, 3$; $\mathbf{n} = 1, 2, \ldots, 8$) of an execution property p (for p = (i) - (xv)) of a program α from a state s is denoted by $exrep_{\mathbf{R_j(n)}}(\alpha, s)$. Here

> for any representation method $\mathbf{R_j(n)}$ (where $\mathbf{j} = 1, 2, 3$; $\mathbf{n} = 1, 2, \ldots, 8$) and any execution property p (for p = (i) - (xv)) of a program α from a state s, $exrep_{\mathbf{R_j(n)}}(\alpha, s) = \{(s, t) \mid \alpha \text{ reaches outcome } t \text{ from state } s\}$. (If there is no representation with respect to $\mathbf{R_j(n)}$ then $exrep_{\mathbf{R_j(n)}}(\alpha, s) = \emptyset$.)

That is, $exrep_{\mathbf{R_j(n)}}(\alpha, s)$ is a set of input-output pair(s) of states with first component s. I will use the notation, '$[\alpha]_{\mathbf{R_j(n)}}$' to denote the binary input-output relation associated with a program α for representation method $\mathbf{R_j(n)}$ (where $\mathbf{j} = 1, 2, 3$, and $\mathbf{n} = 1, 2, \ldots, 8$). That is, $[\alpha]_{\mathbf{R_j(n)}} = \bigcup_{s \in \mathcal{S}} exrep_{\mathbf{R_j(n)}}(\alpha, s)$.

As a reference the symbols used in this chapter to denote the input-output pair(s) associated with a program α from a state s are given in **Table 8**. **Table 8** also indicates the corresponding pictorial representation (in Chapter 1 **Table 4**) and the set for which the symbol is an abbreviation. The general form of the symbols is the cross product of two sets. The first is the singleton $\{s\}$ reflecting the fact that the execution properties p (for p = (i) - (xv)) (listed in Chapter 1 **Table 1**) are relativised to the initial state s. The second is either a singleton or a set of states. A singleton (namely, $\{\infty\}$, or $\{\perp\}$) indicates either there is at most one final state of the program from state s or only one possible outcome is represented; while a set of states (namely, $\mathcal{S}$, $\{\infty, \perp\}$, $\mathcal{S}_\perp$ $(= \mathcal{S} \cup \{\perp\})$, $\mathcal{S}_\infty$ $(= \mathcal{S} \cup \{\infty\})$,

or $\mathcal{S}_U$ $(= \mathcal{S} \cup \{\infty, \perp\}))$ indicates there may be several possible outcomes of the program from s. The meaning of each symbol is the same as explained in Chapter 1 **Table 4** for the corresponding pictorial representation. (Recall an entry in parentheses means blank for j = 1 and the entry for j = 2, 3.)

'$\{s\} \times \mathcal{S}$'	corresponds to '$\bullet \rightarrow$' and
	abbreviates $\{(s,t) \mid \alpha$ terminates cleanly in final state $t\}$
'$\{s\} \times \{\infty\}$'	corresponds to '$\bullet \rightarrow \ldots$' and abbreviates $\{(s,\infty)\}$
'$\{s\} \times \{\perp\}$'	corresponds to '$\bullet$' and abbreviates $\{(s,\perp)\}$
'$\{s\} \times \mathcal{S}_\perp$'	corresponds to '$\bullet$' and
	abbreviates $\{(s,t) \mid \alpha$ terminates cleanly in final state $t\} \cup \{(s,\perp)\}$
'$\{s\} \times \mathcal{S}_\infty$'	corresponds to '$\bullet \ldots$' and
	abbreviates $\{(s,t) \mid \alpha$ terminates cleanly in final state $t\} \cup \{(s,\infty)\}$
'$\{s\} \times \{\infty, \perp\}$'	corresponds to '$\bullet$' and abbreviates $\{(s,\perp)\} \cup \{(s,\infty)\}$
'$\{s\} \times U$'	corresponds to '$\bullet$' and abbreviates
	$\{(s,t) \mid \alpha$ terminates cleanly in final state $t\} \cup \{(s,\perp)\} \cup \{(s,\infty)\}$
'$\{s\} \times \emptyset$'	corresponds to '$\bullet$'
A space	corresponds to a space and abbreviates the empty relation.

Table 8: Table of Relational Representations

Now replacing each entry in Chapter 1 **Table 5** with its corresponding relational representation, we obtain **Table 9**. The information provided in **Table 9** is as explained for Chapter 1 **Table 5**, only using relational representations instead of the pictorial representations, where appropriate. Note that the entries in **Table 9** are in small print. This is done just to shorten the table which otherwise could not be printed on one page.

For example, consider representation method $\mathbf{R_j(4)}$. Then for any program α with property (ix) the entry in the row labelled '(ix)' under the column labelled '$\mathbf{R_j(4)}$' means $[\alpha]_{\mathbf{R_j(4)}}$ (for j = 2, 3) contains input-output pairs (s,t) such that α terminates cleanly from state s in final state t and the input-output pair $(s,\perp)$ but $[\alpha]_{\mathbf{R_1(4)}} = \emptyset$.

	$\mathbf{R_j(1)}$	$\mathbf{R_j(2)}$	$\mathbf{R_j(3)}$	$\mathbf{R_j(4)}$	$\mathbf{R_j(5)}$	$\mathbf{R_j(6)}$	$\mathbf{R_j(7)}$	$\mathbf{R_j(8)}$
(i)	$\{s\}\times\mathcal{S}$	$\{s\}\times\mathcal{S}$	$\{s\}\times\mathcal{S}$	$\{s\}\times\mathcal{S}$	$\{s\}\times\mathcal{S}$	$\{s\}\times\mathcal{S}$	$\{s\}\times\mathcal{S}$	$\{s\}\times\mathcal{S}$
(ii)		$\{s\}\times\mathcal{S}$	$\{s\}\times S_\perp$	$\{s\}\times S_\perp$		$\{s\}\times\mathcal{S}$	$\{s\}\times S_\perp$	$\{s\}\times S_\perp$
(iii)			$\{s\}\times\{\perp\}$	$\{s\}\times\{\perp\}$			$\{s\}\times\{\perp\}$	$\{s\}\times\{\perp\}$
(iv)		$\{s\}\times\mathcal{S}$		$\{s\}\times\mathcal{S}$	$\{s\}\times S_\infty$	$\{s\}\times S_\infty$	$\{s\}\times S_\perp$	$\{s\}\times S_\infty$
(v)		$\{s\}\times\mathcal{S}$		$\{s\}\times S_\perp$		$\{s\}\times S_\infty$	$\{s\}\times S_\perp$	$\{s\}\times U$
(vi)				$\{s\}\times\{\perp\}$		$\{s\}\times\{\infty\}$	$\{s\}\times\{\perp\}$	$\{s\}\times\{\infty,\perp\}$
(vii)					$\{s\}\times\{\infty\}$	$\{s\}\times\{\infty\}$	$\{s\}\times\{\perp\}$	$\{s\}\times\{\infty\}$
(viii)		$(\{s\}\times\mathcal{S})$		$(\{s\}\times\mathcal{S})$		$(\{s\}\times\mathcal{S})$	$(\{s\}\times\mathcal{S})$	$(\{s\}\times\mathcal{S})$
(ix)		$(\{s\}\times\mathcal{S})$		$(\{s\}\times S_\perp)$		$(\{s\}\times\mathcal{S})$	$(\{s\}\times S_\perp)$	$(\{s\}\times S_\perp)$
(x)				$(\{s\}\times\{\perp\})$			$(\{s\}\times\{\perp\})$	$(\{s\}\times\{\perp\})$
(xi)		$(\{s\}\times\mathcal{S})$		$(\{s\}\times\mathcal{S})$		$(\{s\}\times S_\infty)$	$(\{s\}\times S_\perp)$	$(\{s\}\times S_\infty)$
(xii)		$(\{s\}\times\mathcal{S})$		$(\{s\}\times S_\perp)$		$(\{s\}\times S_\infty)$	$(\{s\}\times S_\perp)$	$(\{s\}\times U)$
(xiii)				$(\{s\}\times\{\perp\})$		$(\{s\}\times\{\infty\})$	$(\{s\}\times\{\perp\})$	$(\{s\}\times\{\infty,\perp\})$
(xiv)						$(\{s\}\times\{\infty\})$	$(\{s\}\times\{\perp\})$	$(\{s\}\times\{\infty\})$
(xv)							$(\{s\}\times\{\perp\})$	$(\{s\}\times\emptyset)$

Table 9: Execution Properties and Relational Representation Methods

I now come to the second part of this section. Recall from Chapter 1 (p 25) the claim that the way in which a program is executed corresponds to traversing the execution tree of the program in search of a final state (that is, a leaf of the tree). The four execution methods described in Chapter 1 **Table 6** are:

(a) Depth-first execution method.

(b) Depth-first with backtracking execution method.

(c) Breadth-first execution method.

(d) Breadth-first with backtracking execution method.

I claim here that the way in which a program is represented as a binary relation goes hand in hand with the way in which we think of a program being executed. For a representation method $\mathbf{R_j(n)}$ (where $\mathbf{j} = 1, 2, 3$; $\mathbf{n} = 1, 2, \ldots, 8$), and an execution method E (where E = **(a)** - **(d)**) I need to check whether:

> For any program α the set of outcomes (if any) related by α to a given initial state s under representation method $\mathbf{R_j(n)}$ is the same as the set of results produced by executing α from state s under an execution method E.

Therefore I need a notation for the set of all input-output pairs associated with a program under an execution method.

> For any *execution method* E and any program α, $rel_E(\alpha)$ denotes the set of all pairs (s,t) of states such that α can produce final state t from initial state s under execution method E.

Replacing each entry in Chapter 1 **Table 7** by its corresponding relational representation, we obtain **Table 10** below. Each entry in **Table 10** indicates the set of pairs (s,t) (where $t \in \mathcal{S} \bigcup \{\infty, \perp\}$) such that if program α is activated in state s then state t is a possible result produced using an execution method (**(a)** - **(d)**).

For example, consider execution method **(a)**. Then for any state s and any program α with property (i) the entry in the row labelled '(i)' under the column labelled '**(a)**' means $rel_{(\mathbf{a})}(\alpha)$ contains pairs (s,t) such that α terminates cleanly in final state t for $t \in \mathcal{S}$. Also for any program β with property (ii) the entry in the row labelled '(ii)' under the column labelled '**(a)**' means $rel_{(\mathbf{a})}(\beta)$ contains pairs (s,t) such that α terminates cleanly in final state t for $t \in \mathcal{S}$ and the input-output pair $(s, \perp)$.

From the entries in **Table 10**, I conclude that $rel_{(\mathbf{a})}(\alpha)$ is a total relation over $\mathcal{S} \times (\mathcal{S} \bigcup \{\infty\} \bigcup \{\perp\})$ (because for any state s all execution properties of α from s are represented) and $rel_{(\mathbf{b})}(\alpha)$, $rel_{(\mathbf{c})}(\alpha)$, $rel_{(\mathbf{d})}(\alpha)$ are partial relations over $\mathcal{S} \times (\mathcal{S} \bigcup \{\infty\})$ (because for any state s only some execution properties of α from s are represented).

	(a)	(b)	(c)	(d)
(i)	$\{s\}\times\mathcal{S}$	$\{s\}\times\mathcal{S}$	$\{s\}\times\mathcal{S}$	$\{s\}\times\mathcal{S}$
(ii)	$\{s\}\times S_\perp$	$\{s\}\times\mathcal{S}$		$\{s\}\times\mathcal{S}$
(iii)	$\{s\}\times\{\perp\}$			
(iv)	$\{s\}\times S_\infty$	$\{s\}\times S_\infty$	$\{s\}\times\mathcal{S}$	$\{s\}\times\mathcal{S}$
(v)	$\{s\}\times U$	$\{s\}\times S_\infty$		$\{s\}\times\mathcal{S}$
(vi)	$\{s\}\times\{\infty,\perp\}$	$\{s\}\times\{\infty\}$		$\{s\}\times\mathcal{S}$
(vii)	$\{s\}\times\{\infty\}$	$\{s\}\times\{\infty\}$	$\{s\}\times\{\infty\}$	$\{s\}\times\{\infty\}$
(viii)	$\{s\}\times\mathcal{S}$	$\{s\}\times\mathcal{S}$	$\{s\}\times\mathcal{S}$	$\{s\}\times\mathcal{S}$
(ix)	$\{s\}\times S_\perp$	$\{s\}\times\mathcal{S}$		$\{s\}\times\mathcal{S}$
(x)	$\{s\}\times\{\perp\}$			
(xi)	$\{s\}\times S_\infty$	$\{s\}\times S_\infty$	$\{s\}\times\mathcal{S}$	$\{s\}\times\mathcal{S}$
(xii)	$\{s\}\times U$	$\{s\}\times S_\infty$		$\{s\}\times\mathcal{S}$
(xiii)	$\{s\}\times\{\infty,\perp\}$	$\{s\}\times\{\infty\}$		$\{s\}\times\mathcal{S}$
(xiv)	$\{s\}\times\{\infty\}$	$\{s\}\times\{\infty\}$	$\{s\}\times\{\infty\}$	$\{s\}\times\{\infty\}$
(xv)	$\{s\}\times\emptyset$			

Table 10: Execution Properties and Execution Methods II

My analysis of the correspondence (if any) between a representation method and an execution method is based on the following criterion:

> A *method of representation* $\mathbf{R_j(n)}$ (where $\mathbf{j} = 1, 2, 3$ and $\mathbf{n} = 1, 2, \ldots, 8$) corresponds to an *execution method* E (where $E = \mathbf{(a)} - \mathbf{(d)}$) if and only if $[\alpha]_{\mathbf{R_j(n)}} = rel_E(\alpha)$.

Therefore a method of representation $\mathbf{R_j(n)}$ does not correspond to an execution method E if for some program α:

either (1) $[\alpha]_{\mathbf{R_j(n)}} \not\subseteq rel_E(\alpha)$,

or (2) $[\alpha]_{\mathbf{R_j(n)}} \not\supseteq rel_E(\alpha)$

(1) **Theorem** *The correspondence or otherwise of the various representation methods to the various execution methods is given by the following table.*

	$\mathbf{R_j(1)}$	$\mathbf{R_j(2)}$	$\mathbf{R_j(3)}$	$\mathbf{R_j(4)}$	$\mathbf{R_j(5)}$	$\mathbf{R_j(6)}$	$\mathbf{R_j(7)}$	$\mathbf{R_j(8)}$
(a)	×	×	×	×	×	×	×	(√)
(b)	×	×	×	×	×	(√)	×	×
(c)	×	×	×	×	×	×	×	×
(d)	×	×	×	×	×	×	×	×

where

'×' in a row x (for x = **(a)**, **(b)**, **(c)**, **(d)***) under column* $\mathbf{R_j(n)}$ *(for* $\mathbf{j}$ *= 1, 2, 3;* $\mathbf{n}$ *= 1, 2, ..., 8) means representation method* $\mathbf{R_j(n)}$ *does not correspond to execution method x,*

'√' in a row x (for x = **(a)**, **(b)**, **(c)**, **(d)***) under column* $\mathbf{R_j(n)}$ *(for* $\mathbf{j}$ *= 1, 2, 3;* $\mathbf{n}$ *= 1, 2, ..., 8) means representation method* $\mathbf{R_j(n)}$ *corresponds to execution method x,*

the entries in parentheses mean:

- *'×' for* $\mathbf{j}$ *= 1*
- *'√' for* $\mathbf{j}$ *= 2, 3.*

Proof By way of example I will consider representation method $\mathbf{R_j(8)}$ (for $\mathbf{j}$ = 1, 2, 3). The others are similar.

$\mathbf{R_j(8)(a)}$ A comparison of the column labelled $\mathbf{R_j(8)}$ (for $\mathbf{j}$ = 2, 3) in **Table 9** and the column labelled **(a)** in **Table 10** shows that $[\alpha]_{\mathbf{R_j(8)}}$ and $rel_{(\mathbf{a})}(\alpha)$ contain the same input-output pairs. For the case $\mathbf{j}$ = 1, consider a program α with property (viii). From **Table 9** the entry in row (viii) under column $\mathbf{R_j(8)}$ means $[\alpha]_{\mathbf{R_1(8)}} = \emptyset$. From **Table 10** the entry in row (viii) under column **(a)** means $rel_{(\mathbf{a})}(\alpha)$ contains pairs (s,t) such that α terminates cleanly from state s in final state t. Hence $[\alpha]_{\mathbf{R_1(8)}} \not\supseteq rel_{(\mathbf{a})}(\alpha)$.

(b) Consider a program α with property (iii). From **Table 9** the entry in row (iii) under column $\mathbf{R_j(8)}$ means $[\alpha]_{\mathbf{R_j(8)}}$ (for $\mathbf{j}$ = 1, 2, 3) contains the input-output pair

$(s, \perp)$. From **Table 10** the entry in row (iii) under column **(b)** means α does not relate s to any outcome (under execution method **(b)**), so $rel_{(\mathbf{b})}(\alpha) = \emptyset$. Hence $rel_{(\mathbf{b})}(\alpha) \not\supseteq [\alpha]_{\mathbf{R_j(8)}}$.

(c) Same as for (b) only using '**(c)**' instead of '**(b)**'.

(d) Same as for (b) only using '**(d)**' instead of '**(b)**'. □

My conclusion is that of the twenty-four relational representation methods only four, namely $\mathbf{R_j(n)}$ (for $\mathbf{j} = 2, 3$; $\mathbf{n} = 7, 8$) can be used to describe the behaviour of programs executed using execution methods **(a)** and **(b)** (that is, sequential execution methods p 26). In particular, if messy termination is equated with nontermination $\mathbf{R_j(n)}$ (for $\mathbf{j} = 2, 3$; $\mathbf{n} = 7, 8$) correspond to execution method **(a)**; while if messy termination is not equated with nontermination, $\mathbf{R_2(7)}$ and $\mathbf{R_3(7)}$ correspond to execution method **(b)**, and $\mathbf{R_2(8)}$ and $\mathbf{R_3(8)}$ correspond to execution method **(a)**.

4.3 Representation and Correctness

In §2 I considered 24 methods of representing programs in terms of binary relations. These correspond to the 24 representation methods $\mathbf{R_j(n)}$ (where $\mathbf{j} = 1, 2, 3$; $\mathbf{n} = 1, 2, \ldots, 8$) listed in Chapter 1 **Table 3**. Now recall from Chapter 3 the discussion concerning three kinds of correctness: General correctness, Total correctness, and Partial correctness. In this section I investigate whether the relational model of programs allows a satisfactory formulation of (each of) these notion(s) of correctness. My exposition follows (and occassionally adapts) that of Majster-Cederbaum [1980]. (However, I attempt a more complete analysis.) In particular, I check how the relational model fits with the three kinds of semantics: Dijkstra's weakest precondition semantics, Dijkstra's weakest liberal precondition semantics as well as the general weakest precondition semantics of Jacobs and Gries [1985]. Here, as in Chapter 3, I will assume every state is an initial state of every program. So I need only consider programs with execution properties (i) - (vii).

My analysis is based on the following criterion:

A representation method will be called *suitable* for a notion of correctness iff the representation method distinguishes among all (but not necessarily only) the programs (semantically) distinguishable by the correctness weakest precondition predicate transformers.

Therefore a representation method $\mathbf{R_j(n)}$ (where $\mathbf{j} = 1, 2, 3$; $\mathbf{n} = 1, 2, \ldots 8$) is *not suitable* for a notion of correctness if:

either (1) there is at least one program α, which is not representable (Chapter 1, p 17) with respect to $\mathbf{R_j(n)}$ (that is, $[\alpha]_{\mathbf{R_j(n)}} = \emptyset$),

or (2) there are at least two programs with the same relational representation but different weakest preconditions for the notion of correctness.

As counterexamples, in Theorem 1 I consider the programs:

$\alpha_{(i)}$: *skip*

$\alpha_{(ii)}$: if *true* $\longrightarrow$ *skip* [] *true* $\longrightarrow$ *abort* fi

$\alpha_{(iii)}$: *abort*

$\alpha_{(iv)}$: if *true* $\longrightarrow$ *skip* [] *true* $\longrightarrow$ *loop* fi

$\alpha_{(vii)}$: do *true* $\longrightarrow$ *skip* od.

The subscript p (for $p =$ (i), ..., (iv), (vii)) indicates that the program has execution property p from every state $s \in \mathcal{S}$.

The partial-, total- and general correctness weakest preconditions of each program α_p (for $p =$ (i), (ii), (iii), (iv), (vii)) with respect to a postcondition are given in **Table 11**. In particular, for program α_p (for $p =$ (i), (ii), (iii), (iv), (vii)),

- $wlp(\alpha_p, Q)$ for any $Q \subseteq \mathcal{S} \cup \{\infty\} \cup \{\perp\}$ is given by the entry in row 'α_p' under column 'Partial',
- $wp(\alpha_p, R)$ for any $R \subseteq \mathcal{S}$ is given by the entry in row 'α_p' under column 'Total', and

- $gwp(\alpha_p, T)$ for any $T \subseteq \mathcal{S} \cup \{\infty\}$ is given by the entry in row 'α_p' under column 'General'.

	Partial	Total	General
$\alpha_{(i)}$	Q	R	T
$\alpha_{(ii)}$	Q	$\emptyset$	$\emptyset$
$\alpha_{(iii)}$	$\mathcal{S}$	$\emptyset$	$\emptyset$
$\alpha_{(iv)}$	Q	$\emptyset$	T
$\alpha_{(vii)}$	$\mathcal{S}$	$\emptyset$	$\mathcal{S}$

Table 11: Weakest Preconditions

The relational representations with respect to the various representation methods $\mathbf{R_j(n)}$ (for $\mathbf{j} = 1, 2, 3$; $\mathbf{n} = 1, 2, \ldots, 8$) for each program α_p (for p = (i), ..., (iv), (vii)) are given in **Table 12**. In particular, for a program α_p (for p = (i), ..., (iv), (vii)) and representation method $\mathbf{R_j(n)}$ (where $\mathbf{j} = 1, 2, 3$; $\mathbf{n} = 1, 2, \ldots, 8$), $exrep_{\mathbf{R_j(n)}}(\alpha_p, s)$ is given by the entry in row 'α_p' under column '$\mathbf{R_j(n)}$'. Then $[\alpha_x]_{\mathbf{R_j(n)}} = \bigcup_{s \in \mathcal{S}} exrep_{\mathbf{R_j(n)}}(\alpha_p, s)$

	$\mathbf{R_j(1)}$	$\mathbf{R_j(2)}$	$\mathbf{R_j(3)}$	$\mathbf{R_j(4)}$	$\mathbf{R_j(5)}$	$\mathbf{R_j(6)}$	$\mathbf{R_j(7)}$	$\mathbf{R_j(8)}$
$\alpha_{(i)}$	$\{s\}\times\{s\}$	$\{s\}\times\{s\}$	$\{s\}\times\{s\}$	$\{s\}\times\{s\}$	$\{s\}\times\{s\}$	$\{s\}\times\{s\}$	$\{s\}\times\{s\}$	$\{s\}\times\{s\}$
$\alpha_{(ii)}$		$\{s\}\times\{s\}$	$\{s\}\times\{s,\perp\}$	$\{s\}\times\{s,\perp\}$		$\{s\}\times\{s\}$	$\{s\}\times\{s,\perp\}$	$\{s\}\times\{s,\perp\}$
$\alpha_{(iii)}$			$\{s\}\times\{\perp\}$	$\{s\}\times\{\perp\}$			$\{s\}\times\{\perp\}$	$\{s\}\times\{\perp\}$
$\alpha_{(iv)}$		$\{s\}\times\{s\}$		$\{s\}\times\{s\}$	$\{s\}\times\{s,\infty\}$	$\{s\}\times\{s,\infty\}$	$\{s\}\times\{s,\perp\}$	$\{s\}\times\{s,\infty\}$
$\alpha_{(vii)}$					$\{s\}\times\{\infty\}$	$\{s\}\times\{\infty\}$	$\{s\}\times\{\perp\}$	$\{s\}\times\{\infty,\perp\}$

Table 12: Relational Representations

Each nonblank entry in a row α_p (for p = (i), (ii), (iii), (iv), (vii)) in **Table 12** indicates the set of pairs (s, x) where $x = s, \perp$, or ∞ such that if α_p is activated in a state s then state x is a possible outcome. (Note that here (unlike in **Tables 9** and **10**) we are considering

particular programs namely, those for which there is only one possible outcome for cleanly terminating executions from a state s, namely s itself.)

Now I can prove the main result of this section.

(1) **Theorem** *The suitability or otherwise of the various representation methods for the various notions of correctness is given by the following table.*

	$\mathbf{R_j(1)}$	$\mathbf{R_j(2)}$	$\mathbf{R_j(3)}$	$\mathbf{R_j(4)}$	$\mathbf{R_j(5)}$	$\mathbf{R_j(6)}$	$\mathbf{R_j(7)}$	$\mathbf{R_j(8)}$
General	×	×	×	×	√	×	√	√
Total	√	×	√	×	√	×	√	√
Partial	×	√	×	√	×	√	√	√

where $\mathbf{j}$ = *1, 2, 3, and*

'√' in row 'c' under column '$\mathbf{R_j(n)}$' means $\mathbf{R_j(n)}$ is suitable for the notion of correctness c, and

'×' in row 'c' under column '$\mathbf{R_j(n)}$' means $\mathbf{R_j(n)}$ is not suitable for the notion of correctness c

where 'c' denotes General-, Total- or Partial correctness.

Proof The subsumption relationships between the representation methods in Chapter 1 **Figure 4** reduce the number of cases to be considered. First I consider the representation methods suitable for notions of correctness (that is, each '√' entry in the above table). Since the core topic of this thesis is Dijkstra's weakest precondition semantics I will only show the suitability of the various representation methods for the notion of total correctness. The other cases are similar.

$\mathbf{R_j(1)}$ For any two programs α_1 and α_2 and any predicate $Q \subseteq \mathcal{S}$, I show $[\alpha_1]_{\mathbf{R_j(1)}} = [\alpha_2]_{\mathbf{R_j(1)}}$ iff $wp(\alpha_1, Q) = wp(\alpha_2, Q)$.

Take any two programs α_1 and α_2.Assume there is a predicate $Q \subseteq \mathcal{S}$ such that $[\alpha_1]_{\mathbf{R_j(1)}} = [\alpha_2]_{\mathbf{R_j(1)}}$ but $wp(\alpha_1, Q) \neq wp(\alpha_2, Q)$. Without loss of generality suppose there is some state s such that $s \in wp(\alpha_1, Q)$ but $s \notin wp(\alpha_2, Q)$. Hence α_1 always terminates from s and Q is true in every final state of α_1 from s. So

$(s,t) \in [\alpha_1]_{\mathbf{R_j(1)}}$ implies $t \in Q$. Since $s \notin wp(\alpha_2, Q)$ there is some computation of α_2 from s which does not terminate cleanly in a state in Q. So there is some $t \in \mathcal{S}$ such that $(s,t) \in [\alpha_2]_{\mathbf{R_j(1)}}$ with $t \notin Q$. So $[\alpha_1]_{\mathbf{R_j(1)}} \neq [\alpha_2]_{\mathbf{R_j(1)}}$ yielding a contradiction.

Conversely, assume there are two programs α_1 and α_2 such that $wp(\alpha_1, Q) = wp(\alpha_2, Q)$ for every $Q \subseteq \mathcal{S}$ but $[\alpha_1]_{\mathbf{R_j(1)}} \neq [\alpha_2]_{\mathbf{R_j(1)}}$ Without loss of generality suppose there are states $s,t \in \mathcal{S}$ such that $(s,t) \in [\alpha_1]_{\mathbf{R_j(1)}}$ but $(s,t) \notin [\alpha_2]_{\mathbf{R_j(1)}}$. Hence α_1 always terminates cleanly from s, so $s \in wp(\alpha, \mathcal{S})$. Since $(s,t) \notin [\alpha_2]_{\mathbf{R_j(1)}}$, there is an execution of α_2 from s which does not terminate cleanly. So $s \notin wp(\alpha, \mathcal{S})$. Thus $wp(\alpha_1, \mathcal{S}) \neq wp(\alpha_2, \mathcal{S})$ yielding a contradiction.

$\mathbf{R_j(3)}$ For any two programs α_1 and α_2 and any predicate $Q \subseteq \mathcal{S}$, I show if $[\alpha_1]_{\mathbf{R_j(3)}} = [\alpha_2]_{\mathbf{R_j(3)}}$ then $wp(\alpha_1, Q) = wp(\alpha_2, Q)$, but the converse does not hold.

Take any two programs α_1 and α_2 and any postcondition $Q \subseteq \mathcal{S}$. Assume there is a predicate $Q \subseteq \mathcal{S}$ such that $[\alpha_1]_{\mathbf{R_j(3)}} = [\alpha_2]_{\mathbf{R_j(3)}}$. Recall $\mathbf{R_j(1)}$ is subsumed under $\mathbf{R_j(3)}$ (for $\mathbf{j} = 1, 2, 3$) (Theorem 1.1(a)). So $[\alpha_1]_{\mathbf{R_j(1)}} = [\alpha_2]_{\mathbf{R_j(1)}}$ and hence $wp(\alpha_1, Q) = wp(\alpha_2, Q)$ (by the $\mathbf{R_j(1)}$ case above).

As a counterexample to the converse, consider the programs $\alpha_{(ii)}$ and $\alpha_{(iii)}$. From **Table 11** it follows that $wp(\alpha_{(ii)}, Q) = \emptyset = wp(\alpha_{(iii)}, Q)$ for any $Q \subseteq \mathcal{S}$. But from **Table 12** it follows that $[\alpha_{(ii)}]_{\mathbf{R_j(3)}} \neq [\alpha_{(iii)}]_{\mathbf{R_j(3)}}$. So the converse does not hold.

$\mathbf{R_j(5)}$ For any two programs α_1 and α_2 and any predicate $Q \subseteq \mathcal{S}$, I show if $[\alpha_1]_{\mathbf{R_j(5)}} = [\alpha_2]_{\mathbf{R_j(5)}}$ then $wp(\alpha_1, Q) = wp(\alpha_2, Q)$, but the converse does not hold.

Take any two programs α_1 and α_2 and any postcondition $Q \subseteq \mathcal{S}$. Assume there is a predicate $Q \subseteq \mathcal{S}$ such that $[\alpha_1]_{\mathbf{R_j(5)}} = [\alpha_2]_{\mathbf{R_j(5)}}$. Recall $\mathbf{R_j(1)}$ is subsumed under $\mathbf{R_j(5)}$ (for $\mathbf{j} = 1, 2, 3$) (Theorem 1.1(c)). So $[\alpha_1]_{\mathbf{R_j(1)}} = [\alpha_2]_{\mathbf{R_j(1)}}$ and hence $wp(\alpha_1, Q) = wp(\alpha_2, Q)$ (by the $\mathbf{R_j(1)}$ case above).

As a counterexample to the converse, consider the programs $\alpha_{(iii)}$ and $\alpha_{(iv)}$. From **Table 11** it follows that $wp(\alpha_{(iii)}, Q) = \emptyset = wp(\alpha_{(iv)}, Q)$ for any $Q \subseteq \mathcal{S}$. But from **Table 12** it follows that $[\alpha_{(iii)}]_{\mathbf{R_j(5)}} \neq [\alpha_{(iv)}]_{\mathbf{R_j(5)}}$. So the converse does not hold.

$\mathbf{R_j(7)}$ I show that for any two programs α_1 and α_2 and any predicate $Q \subseteq \mathcal{S}$ if $[\alpha_1]_{\mathbf{R_j(7)}} = [\alpha_2]_{\mathbf{R_j(7)}}$ then $wp(\alpha_1, Q) = wp(\alpha_2, Q)$, but the converse does not hold.

Take any two programs α_1 and α_2. Assume there is a predicate $Q \subseteq \mathcal{S}$ such that $[\alpha_1]_{\mathbf{R_j(7)}} = [\alpha_2]_{\mathbf{R_j(7)}}$ but $wp(\alpha_1, Q) \neq wp(\alpha_2, Q)$. Without loss of generality suppose there is some state s such that $s \in wp(\alpha_1, Q)$ but $s \notin wp(\alpha_2, Q)$. Hence α_1 always terminates from s and Q is true in every final state of α_1 from s. So $(s,t) \in [\alpha_1]_{\mathbf{R_j(7)}}$ implies $t \neq \perp$ and $t \in Q$. Since $s \notin wp(\alpha_2, Q)$ there is some computation of α_2 from s which does not terminate cleanly in a state in Q. Hence either $(s, \perp) \in [\alpha_2]_{\mathbf{R_j(7)}}$ or $(s,t) \in [\alpha_2]_{\mathbf{R_j(7)}}$ with $t \notin Q$. Hence $[\alpha_2]_{\mathbf{R_j(7)}} \neq [\alpha_1]_{\mathbf{R_j(7)}}$ which establishes a contradiction.

As a counterexample to the converse, consider the programs $\alpha_{(ii)}$ and $\alpha_{(iii)}$. From **Table 11** it follows that $wp(\alpha_{(ii)}, Q) = \emptyset = wp(\alpha_{(iii)}, Q)$ for any $Q \subseteq \mathcal{S}$. But from **Table 12** it follows that $[\alpha_{(ii)}]_{\mathbf{R_j(7)}} \neq [\alpha_{(iii)}]_{\mathbf{R_j(7)}}$. So the converse does not hold.

$\mathbf{R_j(8)}$ For any two programs α_1 and α_2 and any predicate $Q \subseteq \mathcal{S}$, I show if $[\alpha_1]_{\mathbf{R_j(8)}} = [\alpha_2]_{\mathbf{R_j(8)}}$ then $wp(\alpha_1, Q) = wp(\alpha_2, Q)$, but the converse does not hold.

Take any two programs α_1 and α_2 and any postcondition $Q \subseteq \mathcal{S}$. Assume there is a predicate $Q \subseteq \mathcal{S}$ such that $[\alpha_1]_{\mathbf{R_j(8)}} = [\alpha_2]_{\mathbf{R_j(8)}}$. Recall $\mathbf{R_j(7)}$ is subsumed under $\mathbf{R_j(8)}$ (for $\mathbf{j} = 1, 2, 3$) (Theorem 1.1(e)). So $[\alpha_1]_{\mathbf{R_j(7)}} = [\alpha_2]_{\mathbf{R_j(7)}}$ and hence $wp(\alpha_1, Q) = wp(\alpha_2, Q)$ (by the $\mathbf{R_j(7)}$ case above).

As a counterexample to the converse, consider the programs $\alpha_{(iii)}$ and $\alpha_{(iv)}$. From **Table 11** it follows that $wp(\alpha_{(iii)}, Q) = \emptyset = wp(\alpha_{(iv)}, Q)$ for any $Q \subseteq \mathcal{S}$. But from **Table 12** it follows that $[\alpha_{(iii)}]_{\mathbf{R_j(8)}} \neq [\alpha_{(iv)}]_{\mathbf{R_j(8)}}$. (Note this also follows from the fact that $[\alpha_{(iii)}]_{\mathbf{R_j(7)}} \neq [\alpha_{(iv)}]_{\mathbf{R_j(7)}}$ because $\mathbf{R_j(7)}$ is subsumed under $\mathbf{R_j(8)}$ (for $\mathbf{j} = 1, 2, 3$).) So the converse does not hold.

Second I consider the representation methods not suitable for notions of correctness. I give counterexamples for each '×' entry in the table. (Note in order to use the subsumption relationships in Chapter 1 **Figure 4** the order in which the representation methods are considered is reversed.)

Take any postconditions $Q \subseteq \mathcal{S} \cup \{\infty, \perp\}$, $R \subseteq \mathcal{S}$ and $T \subseteq \mathcal{S} \cup \{\infty\}$. Then from the

entries in **Tables 11** and **12** it can be deduced that:

$\mathbf{R_j(7)}$ $gwp(\alpha_{(ii)}, T) \neq gwp(\alpha_{(iv)}, T)$ but $[\alpha_{(ii)}]_{\mathbf{R_j(7)}} = [\alpha_{(iv)}]_{\mathbf{R_j(7)}}$.

$\mathbf{R_j(6)}$ $gwp(\alpha_{(i)}, T) \neq gwp(\alpha_{(ii)}, T)$ but $[\alpha_{(i)}]_{\mathbf{R_j(6)}} = [\alpha_{(ii)}]_{\mathbf{R_j(6)}}$.
$wp(\alpha_{(i)}, R) \neq wp(\alpha_{(ii)}, R)$ but $[\alpha_{(i)}]_{\mathbf{R_j(6)}} = [\alpha_{(ii)}]_{\mathbf{R_j(6)}}$.

$\mathbf{R_j(5)}$ $wlp(\alpha_{(ii)}, Q) \neq wlp(\alpha_{(iii)}, Q)$ but $[\alpha_{(ii)}]_{\mathbf{R_j(5)}} = [\alpha_{(iii)}]_{\mathbf{R_j(5)}}$.

$\mathbf{R_j(4)}$ Let $\alpha_{(xv)}$ is a program which never starts. $gwp(\alpha_{(xv)}, T) = \emptyset$, and
$gwp(\alpha_{(vii)}, T) = \mathcal{S}$ but $[\alpha_{(xv)}]_{\mathbf{R_j(4)}} = \emptyset = [\alpha_{(vii)}]_{\mathbf{R_j(4)}}$.
$wp(\alpha_{(i)}, R) \neq wp(\alpha_{(iv)}, R)$ but $[\alpha_{(i)}]_{\mathbf{R_j(4)}} = [\alpha_{(iv)}]_{\mathbf{R_j(4)}}$.

$\mathbf{R_j(3)}$ $gwp(\alpha_{(iv)}, T) \neq gwp(\alpha_{(vii)}, T)$ but $[\alpha_{(iv)}]_{\mathbf{R_j(3)}} = [\alpha_{(vii)}]_{\mathbf{R_j(3)}}$.
$wlp(\alpha_{(iv)}, Q) \neq wlp(\alpha_{(vii)}, Q)$ but $[\alpha_{(iv)}]_{\mathbf{R_j(3)}} = [\alpha_{(vii)}]_{\mathbf{R_j(3)}}$.

$\mathbf{R_j(2)}$ $gwp(\alpha_{(i)}, T) \neq gwp(\alpha_{(ii)}, T)$ but $[\alpha_{(i)}]_{\mathbf{R_j(2)}} = [\alpha_{(ii)}]_{\mathbf{R_j(2)}}$.
$wp(\alpha_{(i)}, R) \neq wp(\alpha_{(ii)}, R)$ but $[\alpha_{(i)}]_{\mathbf{R_j(2)}} = [\alpha_{(ii)}]_{\mathbf{R_j(2)}}$.
(Note these counterexamples also follow from the $\mathbf{R_j(6)}$ case since the programs indistinguishable with respect to $\mathbf{R_j(6)}$ are indistinguishable with respect to $\mathbf{R_j(2)}$ (by Theorem 1.1 (d)).)

$\mathbf{R_j(1)}$ $wlp(\alpha_{(iv)}, Q) \neq wlp(\alpha_{(vii)}, Q)$ but $[\alpha_{(iv)}]_{\mathbf{R_j(1)}} = [\alpha_{(vii)}]_{\mathbf{R_j(1)}}$.
$gwp(\alpha_{(iv)}, T) \neq gwp(\alpha_{(vii)}, T)$ but $[\alpha_{(iv)}]_{\mathbf{R_j(1)}} = [\alpha_{(vii)}]_{\mathbf{R_j(1)}}$.
(Note these counterexamples also follow from the $\mathbf{R_j(3)}$ case since the programs indistinguishable with respect to $\mathbf{R_j(3)}$ are indistinguishable with respect to $\mathbf{R_j(1)}$ (by Theorem 1.1 (a)).) □

To conclude this section I point out some representation methods investigated in the literature. Hoare and Lauer [1974] used partial relations and option **R2**(1)(b) to represent deterministic programs. (Their method corresponds to $\mathbf{R_1(2)}$ or $\mathbf{R_2(2)}$.) In fact, as pointed out in Jacobs and Gries [1985], it is the simplest method but it does not provide sufficient information for studying general-and total correctness of programs.

Guerreiro [1982] claimed using partial relations and option **R2**(1)(a) to represent nondeterministic programs is useful if we are only interested in studying total correctness of programs.

This method (which corresponds to $\mathbf{R_j(1)}$ (for $\mathbf{j} = 1, 2$)) was used by Wand [1977] to give the first relational characterisation of Dijkstra's weakest precondition predicate transformer. Recall (§2, p 117) the problem with representation method $\mathbf{R_3(1)}$ is it has no obvious representation for an execution of a program which *always* terminates messily from some initial state. However, as pointed out in Nelson [1989] the problem can be finessed by a convention: If the set of final states for a program is the state space, then messy termination is allowed; if the set of final states is a subset of the state space, then messy termination is forbidden. This convention as introduced by Hehner[1984] and Hoare [1985] has aroused some controversy (Parnas [1985])

It is noted in Nelson [1989] that Harel and Pratt were the first to use partial relations and option **R2**(4)(a) to represent nondeterministic programs (that is, representation methods $\mathbf{R_j(7)}$ (for $\mathbf{j} = 1, 2$)). (Harel and Pratt [1978] and Harel [1979b].) If this method is chosen the question is: What (if any) program is represented by a relation which relates an initial state to no outcome at all? Nelson ([1989] p 526, 527) discusses some possibilities. As pointed out in Guerreiro ([1982] p 165) representation method $\mathbf{R_1(7)}$ is independent of any notion of correctness. In fact, this approach was proposed by Plotkin [1976] for the study of total correctness and has been investigated by de Roever [1976], de Bakker [1978], Guerreiro ([1980], [1981], [1982]) and many others. However, it is noted in Jacobs and Gries [1985] that 'this model gives extraneous information for studying (only) total correctness: a pair (s, s') is irrelevant if there is also a pair $(s, \perp)$'. Jacobs and Gries [1985] used this representation method to study general-, total- and partial correctness.

The representation methods $\mathbf{R_j(3)}$, $\mathbf{R_j(4)}$, $\mathbf{R_j(5)}$, $\mathbf{R_j(6)}$, and $\mathbf{R_j(8)}$ (for $\mathbf{j} = 1, 2, 3$) have not been investigated in the literature.

4.4 Modelling Weakest Precondition Semantics

My aim in this section is to find at least one method of relational representation for nondeterministic programs which is consistent Dijkstra's weakest precondition semantics (in the sense that all the formulae in the algebra of weakest preconditions can be verified). This

method will be used in §5 to prove the formulae of Dijkstra's algebra of weakest preconditions. Recall from §2 the methods of relational representation (namely, $\mathbf{R_j(n)}$ (for $\mathbf{j} = 1, 2, 3$; $\mathbf{n} = 1,\ldots 8$)) for nondeterministic programs. First I show $\mathbf{R_j(n)}$, (for $\mathbf{j} = 1, 2, 3$; $\mathbf{n} = 1, 2, \ldots, 6$) and $\mathbf{R_j(n)}$ (for $\mathbf{j} = 1, 2$; $\mathbf{n} = 7, 8$) do not satisfactorily model Dijkstra's weakest precondition semantics. Second I investigate sufficient (but not necessary) conditions for the Dijkstra/Gries laws to hold and then specialise to a certain set of binary relations.

Recall definition (1.4) of the weakest precondition of a program α (represented as a binary (input-output) relation $[\alpha] \subseteq \mathcal{S} \times \mathcal{S}$) with respect to a postcondition Q. Namely: $wp(\alpha, Q)$ $= [\alpha] : \mathcal{S} \cap ([\alpha] : Q')' = \text{dom } [\alpha] \cap ([\alpha] : Q')'$

(1) **Theorem** *Using representation method* $\mathbf{R_j(2)}$ ($\mathbf{R_j(4)}$ or $\mathbf{R_j(6)}$) (for $\mathbf{j} = 1, 2, 3$), *Gries [1981] (8.3) is not satisfied (that is, there are at least two programs* α *and* β *and a postcondition* Q *such that* $wp(\alpha;\beta, Q) \neq wp(\alpha, wp(\beta, Q))$.

Proof By way of example I consider $\mathbf{R_j(2)}$ (for $\mathbf{j} = 1, 2, 3$). A similar argument can be used for the other six representation methods. Take, for example, the programs α, β and γ:

α: if $x := 1 \longrightarrow$ *skip* $[]$ $x := 1 \longrightarrow x := 2$ fi

β: if $x = 1 \longrightarrow x := x$ $[]$ $x = 1 \longrightarrow$ *abort* fi

γ: if $x = 1 \longrightarrow x := x$ $[]$ $x = 1 \longrightarrow$ *loop* fi

and the postcondition $Q = true$. The first step is to represent these programs as relations. That is,

$[\alpha]_{\mathbf{R_j(2)}} = \{(s, s) \mid x = 1 \text{ in state } s\} \cup \{(s, t) \mid x = 1 \text{ in state } s \text{ and } x = 2 \text{ in state } t\}$, and

$[\beta]_{\mathbf{R_j(2)}} = \{(t, t) \mid x = 1 \text{ in state } t\}$, and

$[\gamma]_{\mathbf{R_j(2)}} = \{(t, t) \mid x = 1 \text{ in state } t\}$.

Also $[\alpha;\beta]_{\mathbf{R_j(2)}} = [\alpha]_{\mathbf{R_j(2)}};[\beta]_{\mathbf{R_j(2)}} = \{(s, s) \mid x = 1 \text{ in state } s\}$, and

$[\alpha;\gamma]_{\mathbf{R_j(2)}} = \{(t, t) \mid x = 1 \text{ in state } t\}$.

Then by definition (1.4) $wp(\beta, Q) = \{t \mid x = 1 \text{ in state } t\}$ and $wp(\alpha, wp(\beta, Q)) = \emptyset$ but $wp(\alpha;\beta, Q) = \mathcal{S}$. Hence $wp(\alpha;\beta, Q) \not\subseteq wp(\alpha, wp(\beta, Q))$. Also $wp(\gamma, Q) = \{t \mid$

$x = 1$ in state $t\}$ and $wp(\alpha, wp(\gamma, Q)) = \emptyset$ but $wp(\alpha; \gamma, Q) = \mathcal{S}$. Hence $wp(\alpha; \gamma, Q) \not\subseteq wp(\alpha, wp(\gamma, Q))$. □

The problem lies in the fact that if representation method $\mathbf{R_j(2)}$ (or $\mathbf{R_j(6)}$) (for $\mathbf{j} = 1, 2, 3$) is chosen the binary (input-output) relation associated with β does not record that β sometimes terminates messily when begun in a state in which x has the value 1 For program γ, if representation method $\mathbf{R_j(2)}$ (or $\mathbf{R_j(4)}$) (for $\mathbf{j} = 1, 2, 3$) is chosen, the problem is due to the fact that the binary (input-output) relation associated with γ does not record that γ sometimes does not terminate when begun in a state in which x has the value 1.

In other words, using representation method $\mathbf{R_j(2)}$ ($\mathbf{R_j(4)}$ or $\mathbf{R_j(6)}$) (for $\mathbf{j} = 1, 2, 3$) there is no distinction between a program which 'always terminates cleanly' and one which 'sometimes terminates cleanly'. (Milner first noted this problem for $\mathbf{R_2(2)}$ (Plotkin ([1976] p 454)).) This is also exactly the problem found in de Roever ([1976] p 473), Guerreiro ([1982] p 165) and Gordon ([1989a] p 434). It is therefore desirable to have a representation method which caters specifically for the state of messy termination and the state of nontermination. My conclusion is that not one of the representation methods $\mathbf{R_j(n)}$ (for $\mathbf{j} = 1, 2, 3$ and $\mathbf{n} = 2, 4, 6$) can be used to model Dijkstra's weakest precondition semantics. (Recall in §3 these representation methods were found to be not suitable for total correctness.) It therefore remains to consider representation method $\mathbf{R_j(n)}$ (for $\mathbf{j} = 1, 2, 3$; $\mathbf{n} = 1, 3, 5, 7, 8$).

In Chapter 3.5 (p 95–99) I showed that execution method **(a)** gives rise to a definition of weakest precondition consistent with Dijkstra's ([1975], [1976]) definition of weakest precondition. The result of Theorem (2.1) shows that if messy termination is equivalent to nontermination only representation methods $\mathbf{R_j(n)}$ (for $\mathbf{j} = 2, 3$; $\mathbf{n} = 7, 8$) correspond to execution method **(a)**. These results eliminate representation methods $\mathbf{R_j(n)}$ (for $\mathbf{j} = 1, 2, 3$; $\mathbf{n} = 1, 3, 5$) and $\mathbf{R_1(7)}$ from further consideration. It therefore remains to consider $\mathbf{R_j(n)}$ (for $\mathbf{j} = 2, 3$; $\mathbf{n} = 7, 8$). Since $\mathbf{R_j(7)}$ is subsumed under $\mathbf{R_j(8)}$ (for $\mathbf{j} = 1, 2, 3$) (Theorem 1.1 (e), p 21) the programs indistinguishable with respect to $\mathbf{R_j(8)}$ are indistinguishable with respect to $\mathbf{R_j(7)}$. Hence it suffices to consider $\mathbf{R_j(8)}$ (for $\mathbf{j} = 2, 3$).

Then the set of outcomes is $U = \mathcal{S} \cup \{\infty\} \cup \{\perp\}$. Postconditions are subsets of U. For any program α there corresponds a binary (input-output) relation written $[\alpha]_{\mathbf{R_j(8)}} \subseteq \mathcal{S} \times U$. For ease of exposition I will drop the subscript (so $[\alpha]$ denotes the binary (input-output) relation corresponding to program α).

Now it seems reasonable to ask: 'Under what necessary and sufficient conditions does (8.3) hold?' By analogy with the result that functional composition of two functions is defined when the range of the first is contained in the domain of the second one would expect, for any programs α and β and any predicate Q that

$$wp(\alpha;\beta,Q) = wp(\alpha, wp(\beta,Q)) \text{ iff } \text{ran}[\alpha] \subseteq \text{dom}[\beta].$$

But in fact $\text{ran}[\alpha] \subseteq \text{dom}[\beta]$ is sufficient but not necessary for (8.3) to hold.

(2) **Theorem** *For any programs α and β and postcondition Q if $ran[\alpha] \subseteq dom[\beta]$ then $wp(\alpha;\beta,Q) = wp(\alpha, wp(\beta,Q))$. The converse does not hold.*

Proof I first show that $\text{ran}[\alpha] \subseteq \text{dom}[\beta]$ is a sufficient condition. For suppose it holds. Then

$wp(\alpha;\beta,Q) = \text{dom}[\alpha;\beta] \cap ([\alpha;\beta] : Q')'$.

$$\begin{aligned} wp(\alpha, wp(\beta,Q)) &= \text{dom}[\alpha] \cap ([\alpha] : wp(\beta,Q)')'. \\ &= \text{dom}[\alpha] \cap ([\alpha]\text{: } (\text{dom}[\beta] \cap ([\beta] : Q')')')' \\ &= \text{dom}[\alpha] \cap ([\alpha]\text{: } (\text{dom}[\beta])' \cup ([\beta] : Q')))' \text{ (by Theorem (1.1)(ii))} \\ &= \text{dom}[\alpha] \cap ([\alpha]\text{: } (\text{dom}[\beta])') \cup ([\alpha] : ([\beta] : Q')))' \\ &= \text{dom}[\alpha] \cap ([\alpha]\text{: } (\text{dom}[\beta])')' \cap ([\alpha] : ([\beta] : Q'))' \\ &= \text{dom}[\alpha] \cap ([\alpha]\text{: } (\text{dom}[\beta])')' \cap ([\alpha;\beta] : Q')' \end{aligned}$$

So $wp(\alpha;\beta,Q) = wp(\alpha, wp(\beta,Q))$ iff

$\text{dom}[\alpha;\beta] \cap ([\alpha;\beta], Q')' = \text{dom}[\alpha] \cap ([\alpha]\text{: } (\text{dom}[\beta])')' \cap ([\alpha;\beta] : Q')'$.

Hence a necessary and sufficient condition for (8.3) to hold is that

$(*)$ $\text{dom}([\alpha;\beta]) = \text{dom}[\alpha] \cap ([\alpha]\text{: } (\text{dom}[\beta])')'$

Now for $\text{ran}[\alpha] \subseteq \text{dom}[\beta]$ to be a sufficient condition for (8.3) to hold it must be a sufficient condition for $(*)$ to hold. I now consider under what conditions (if any) $(*)$ holds. Consider the right hand side of the equality in $(*)$.

$\text{dom}[\alpha] \cap ([\alpha]\text{:}(\text{dom } [\beta])')'$

$= \text{dom}[\alpha] \cap \{s \mid (\forall t \in \mathcal{S})[(s,t) \in [\alpha] \Rightarrow t \in \text{dom}[\beta]]\}$

$= \text{dom}[\alpha] \cap \{s \mid (\forall t \in \mathcal{S})[(s,t) \in [\alpha] \Rightarrow (\exists u \in \mathcal{S})((t,u) \in [\beta])]\}$

Thus

$s \in \text{dom}[\alpha] \cap ([\alpha]\text{: } (\text{dom}[\beta])')'$

$\Leftrightarrow (\exists t \in \mathcal{S})[(s,t) \in [\alpha]]$ and $(\forall t \in \mathcal{S})[(s,t) \in [\alpha] \Rightarrow (\exists u \in \mathcal{S})((t,u) \in [\beta])]$

$\Rightarrow (\exists t \in \mathcal{S})(\exists u \in \mathcal{S})\ [(s,t) \in [\alpha]$ and $(t,u) \in [\beta]]$

$\Leftrightarrow (\exists u \in \mathcal{S})(\exists t \in \mathcal{S})\ [(s,t) \in [\alpha])$ and $((t,u) \in [\beta]]$

$\Leftrightarrow (\exists u \in \mathcal{S})[(s,u) \in [\alpha];[\beta]]$

$\Leftrightarrow s \in \text{dom}[\alpha;\beta]$.

But the implication cannot be reversed without assuming that

$(\exists t \in \mathcal{S})(\exists u \in \mathcal{S})\ [(s,t) \in [\alpha]$ and $(t,u) \in [\beta]]$

$\Rightarrow (\forall t \in \mathcal{S})[(s,t) \in [\alpha] \Rightarrow (\exists u \in \mathcal{S})((t,u) \in [\beta])]$

that is, that $\text{ran}[\alpha] \subseteq \text{dom}[\beta]$.

Therefore $\text{ran}[\alpha] \subseteq \text{dom}[\beta]$ guarantees that $\text{dom}[\alpha] \cap ([\alpha]\text{: } (\text{dom}[\beta])')' = \text{dom}[\alpha;\beta]$. In order for (8.3) to hold, it is thus sufficient to assume that any state to which α can lead from s is a possible initial state of β.

I now show that $\text{ran}[\alpha] \subseteq \text{dom}[\beta]$ is *not* a necessary condition. In order to have $\text{ran}[\alpha] \subseteq \text{dom}[\beta]$ as a necessary condition we must have for any programs α and β and any predicate Q, $wp(\alpha, wp(\beta, Q)) = wp(\alpha;\beta, Q) \Rightarrow \text{ran}[\alpha] \subseteq \text{dom}[\beta]$. As a counterexample to this implication, consider the programs α, β and γ in Theorem (1) and the postcondition $Q = \mathit{true}$. Now the relations associated with these programs are:

$[\alpha] = \{(s,s) \mid x = 1$ in state $s\} \cup \{(s,t) \mid x = 1$ in state s and $x = 2$ in state $t\} \cup$

$\{(s,\perp) \mid x \neq 1$ in state $s\}$

$[\beta] = \{(t,t) \mid x = 1$ in state $t\} \cup \{(t,\perp) \mid t \in \mathcal{S}\}$

$[\gamma] = \{(t,t) \mid x = 1$ in state $t\} \cup \{(t,\infty) \mid x = 1$ in state $t\} \cup$

$\{(t,\perp) \mid x \neq 1$ in state $s\}$

Also

$[\alpha;\beta] = \{(s,t) \mid x = 1$ in state $t\} \cup \{(s,\perp) \mid s \in \mathcal{S}\}$

$[\alpha;\gamma] = \{(s,t) \mid x = 1 \text{ in state } t\} \cup \{(s,\infty) \mid s \in \mathcal{S}\} \cup \{(s,\perp) \mid s \in \mathcal{S}\}$

It then follows that $wp(\beta,Q) = \emptyset$, $wp(\alpha,wp(\beta,Q)) = \emptyset$ and $wp(\alpha;\beta,Q) = \emptyset$. So (8.3) is satisfied. But $\text{ran}[\alpha] = \mathcal{S} \cup \{\perp\}$ and $\text{dom}[\beta] = \mathcal{S}$, so $\text{ran}[\alpha] \not\subseteq \text{dom}[\beta]$. On the other hand, for program γ, $wp(\gamma,Q) = \emptyset$, so $wp(\alpha,wp(\gamma,Q)) = \emptyset$ and $wp(\alpha;\gamma,Q) = \emptyset$. Hence $wp(\alpha;\gamma,Q) = wp(\alpha,wp(\gamma,Q))$. But $\text{ran}[\alpha] \not\subseteq \text{dom}[\gamma] = \mathcal{S}$. □

If total relations $R \subseteq U \times U$ are used to model programs then for any program α $\text{dom}[\alpha] = U$ and hence for any programs α and β, $\text{ran}[\alpha] \subseteq \text{dom}[\beta]$ always holds. Then for the counterexamples in the proof of Theorem (1), $wp(\alpha,wp(\beta,Q)) = \emptyset = wp(\alpha;\beta,Q)$ and $\text{ran}[\alpha] \subseteq \text{dom}[\beta]$ since $\text{dom}[\beta] = U$. Also $wp(\alpha,wp(\gamma,Q)) = \emptyset = wp(\alpha;\gamma,Q)$ and $\text{ran}[\alpha] \subseteq \text{dom}[\gamma]$ since $\text{dom}[\beta] = U$.

Therefore a necessary and sufficient condition for (8.3) to hold is that the relations used for modelling programs must be total. This condition eliminates representation methods $\mathbf{R_2(7)}$ and $\mathbf{R_2(8)}$ from further consideration. It remains to consider representation methods $\mathbf{R_3(7)}$ and $\mathbf{R_3(8)}$. Also, recall $[\alpha]_{\mathbf{R_3(7)}} \subseteq \mathcal{S} \times (\mathcal{S} \cup \{\perp\})$ and $[\alpha]_{\mathbf{R_3(8)}} \subseteq \mathcal{S} \times U$. My conclusion is that a variant of representation method $\mathbf{R_3(8)}$ which uses total relations over $U \times U$ or a variant of representation method $\mathbf{R_3(7)}$) which uses total relations $(\mathcal{S} \cup \{\perp\}) \times (\mathcal{S} \cup \{\perp\})$ is required.

A variant of representation method $\mathbf{R_3(7)}$ was proposed by Plotkin [1976]. Although it has been investigated by de Roever [1976], de Bakker [1978], Guerreiro ([1980], [1981], [1982]), Jacobs and Gries [1985] and many others, I select a further extension thereof for presentation. Namely, I consider a variant of representation method $\mathbf{R_3(8)}$ (in a context where abortion is not equivalent to nontermination). This seems to me a culmination of the research that lead first to Plotkin's [1976] method which caters specifically for nontermination and then to the suggestions of (for example) Blikle [1981] and Nelson [1989] to have a model which caters for the other ways in which a program can go wrong.

It therefore seems that *total* relations of the form $R \subseteq U \times U$ could represent programs. Unfortunately, this is not the case. In the first instance, the input-output pairs $(\perp,s)$, $(\infty,s) \subseteq$

$U \times U$ do not represent meaningful computations for any program. Secondly, a total relation R such that for some $s \in U$, $\{t \mid (s,t) \in R\}$ is infinite does not correspond to any boundedly nondeterministic program. In other words, under an assumption of bounded nondeterminism such a relation would not represent any program. One way to eliminate these problems is to restrict our attention to total relations $R \subseteq U \times U$ satisfying three conditions: (i) if $(\perp, s) \in R$ then $s = \perp$, (ii) if $(\infty, s) \in R$ then $s = \infty$, and (iii) $\{t \mid (s,t) \in R\}$ is finite.

Following the suggestion in Blikle (([1977] p 31), ([1981], p 213)), Guerreiro (([1981] p 139)([1982] p 167)) as well as in Jacobs and Gries ([1985] p 71), I therefore consider a restricted set of binary relations over $U = S \bigcup \{\infty, \perp\}$ which I call *execution relations*. Namely:

(3) A binary relation $R \subseteq U \times U$ is called an *execution relation* if it satisfies

(i) $\forall s \in U$, $\{t \mid (s,t) \in R\} \neq \emptyset$ (that is, dom $R = U$),

and (ii) $\forall s \in U$, if $(\perp, s) \in R$ then $s = \perp$

and (iii) $\forall s \in U$, if $(\infty, s) \in R$ then $s = \infty$,

and (iv) $\forall s \in U$ $\{t \mid (s,t) \in R\}$ is finite.

The idea with restriction (i) is that every state in which a program is activated is a possible initial state of the program. Therefore since nontermination is not equated with messy termination, an execution of a program α, begun in some state s may terminate cleanly in some final state t, it may terminate messily or it may not terminate. Restriction (ii) means that execution 'begun' in a 'state of messy termination' terminates messily. Analogously, restriction (iii) means that execution 'begun' in a 'state of nontermination' never terminates. It then follows from (i) and (ii) (or (iiii)) that the only possible outcome of a program begun in $\perp$ (or ∞) is $\perp$ (or respectively, ∞). Finally, the idea with restriction (iii) is that all programs must be boundedly nondeterministic.

It is interesting to note that in the context where abortion is equivalent to nontermination, a relation satisfying (3) (i), (ii), (iv) corresponds exactly to the notion of a *resulting relation* in Blikle [1977], a *programmable relation* in Guerreiro ([1980], [1982]) and a *program relation* in Jacobs and Gries [1985].

In conclusion I point out one limitation of this method of representing a program as a binary relation. The representation of a program α which is not meant to terminate as an execution relation $[\alpha] = \{(s, \infty) \mid s \in U\}$ is useless for studying the properties of such a program. For this we need to analyse the intermediate states of an execution of α from a state s. A model in which programs are modelled as *next-state relations* instead of as input-output relations is presented in Chapter 6.

4.5 Verifying the Dijkstra/Gries Conditions

My aim in this section is to provide a relational model based on execution relations (defined in (4.3)), to give the semantics for Dijkstra's guarded command language and to try verifying the formulae of Dijkstra's algebra of weakest preconditions. I will do so equationally as far as possible.

To maintain consistency with the Dijkstra/Gries type exposition in this section I do not distinguish between messily terminating- and nonterminating executions. Let $U = \mathcal{S} \bigcup \{\perp\}$ be the state space where '$\perp$' denotes simultaneously the final state of messily terminating executions and nonterminating executions. Then a binary relation $R \subseteq U \times U$ is called an *execution relation* if it satisfies

(1) (i) dom $R = U$,

and (ii) $\forall x \in U$, if $(\perp, x) \in R$ then $x = \perp$,

and (iii) $\forall x \in U$, $\{(y \mid (x, y) \in R\}$ is finite.

Suppose the behaviour of a program α is described by an execution relation $[\alpha] \subseteq U \times U$. Then

(2) **Theorem** $\forall Q \subseteq U$, $\perp \in Q$ *iff* $\perp \in ([\alpha] : Q')'$.

Proof Left to right: Suppose $\perp \notin ([\alpha] : Q')'$. Then for some $y \in U$, $(\perp, y) \in [\alpha]$ and $y \in Q'$. It follows from (1)(ii) that $y = \perp$; hence $\perp \in Q'$. Thus if $\perp \in Q$ then $\perp \in ([\alpha] : Q')'$.

Right to left: Suppose $\perp \in ([\alpha] : Q')'$. Then for every $y \in U$, if $(\perp, y) \in [\alpha]$ then $y \in Q$. So by (1)(i) since $(\perp, \perp) \in [\alpha]$ we get $\perp \in Q$. □

(3) **Corollary**

(a) *For any predicate* $Q \subseteq U$, $\perp \in Q$ iff $\perp \in [\alpha] : Q$.

(b) $[\alpha] : \mathcal{S} = \mathcal{S}$.

Proof

(a) Take any $Q \subseteq U$. Then by Theorem (2), $\perp \in Q'$ iff $\perp \in ([\alpha] : Q')$. Replacing Q' by P, it follows that for any $P \subseteq U$, $\perp \in P$ iff $\perp \in [\alpha] : P$.

(b) By (1)(i) $\text{dom}[\alpha] = [\alpha] : U = U$. But by Theorem (2) $\perp \notin [\alpha] : \mathcal{S}$ (since $\perp \notin \mathcal{S}$). So $[\alpha] : \mathcal{S} = \mathcal{S}$. □

I now define a total correctness predicate transformer for execution relations. Recall the definition (1.4) (p 112) of the weakest precondition of a program α with respect to a post-condition Q, that is, $wp(\alpha, Q) = [\alpha] : \mathcal{S} \bigcap ([\alpha] : Q')'$.

Since Dijkstra's weakest precondition for a program α with respect to a postcondition Q represents the set of initial states from which α is guaranteed to terminate cleanly (and in Q), it is reasonable to consider only subsets that do not contain '$\perp$' as postconditions for programs. Then for any postcondition Q, $\perp \notin Q$ and hence by Theorem (2), $\perp \notin ([\alpha] : Q')'$. Thus the definition of a weakest precondition for a program α (represented as an execution relation $[\alpha] \subseteq U \times U$) with respect to a postcondition $Q \subseteq \mathcal{S}$ is:

(4) $wp(\alpha, Q) = ([\alpha] : Q')'$.

So $wp(\alpha, -) : \mathcal{S} \longrightarrow \mathcal{S}$ such that $\forall Q \subseteq \mathcal{S}$, $wp(\alpha, Q) = ([\alpha] : Q')'$ is a total correctness predicate transformer for execution relations.

To check that $wp(\alpha, -)$ is a useful total correctness predicate transformer we must verify that it satisfies the five healthiness criteria (in Chapter 2.2, p 38, 39).

(5) **Theorem** *For any program* α *and predicates* Q *and* R*:*

(a) (Gries [1981] (7.3)) **Law of the Excluded Miracle:** $wp(\alpha, \emptyset) = \emptyset$.

(b) (Gries [1981] (7.4)) **Distributivity of Conjunction:**

$$wp(\alpha, Q) \bigcap wp(\alpha, R) = wp(\alpha, Q \bigcap R).$$

(c) (Gries [1981] (7.5)) **Law of Monotonicity:**

$$\textit{If } Q \subseteq R \textit{ then } wp(\alpha, Q) \subseteq wp(\alpha, R).$$

(d) (Gries [1981] (7.6)) **Distributivity of Disjunction:**

$$wp(\alpha, Q) \bigcup wp(\alpha, R) \subseteq wp(\alpha, Q \bigcup R).$$

(e) (Dijkstra ([1976] p 76)) **Law of Continuity:**

For any increasing chain under $\subseteq$ *of predicates* $\{P_i\}_{i \in I}$ $wp(\alpha, \bigcup_i P_i) = \bigcup_i wp(\alpha, P_i)$.

Proof (a) $wp(\alpha, \emptyset) = ([\alpha] : \emptyset')' \; \mathcal{S}' = \emptyset$.

(b) $wp(\alpha, Q \bigcap R) = ([\alpha] : (Q \bigcap R)')' = ([\alpha] : (Q' \bigcup R'))' = ([\alpha] : Q' \bigcup [\alpha] : R')' = ([\alpha] : Q')' \bigcap ([\alpha] : R')' = wp(\alpha, Q) \bigcap wp(\alpha, R)$

(c) Suppose $Q \subseteq R$. Then $R' \subseteq Q'$. By the monotonicity of Peirce product with respect to $\subseteq$ (Theorem (1.1) (vii)), $[\alpha] : R' \subseteq [\alpha] : Q'$, from which it follows that $([\alpha] : Q')' \subseteq ([\alpha] : R')'$; so $wp(\alpha, Q) \subseteq wp(\alpha, R)$.

(d) $Q \subseteq Q \bigcup R$ and $R \subseteq Q \bigcup R$ hence by (c) $wp(\alpha, Q) \subseteq wp(\alpha, Q \bigcup R)$ and $wp(\alpha, R) \subseteq wp(\alpha, Q \bigcup R)$; so $wp(\alpha, Q) \bigcup wp(\alpha, R) \subseteq wp(\alpha, Q \bigcup R)$

(e) Choose any increasing chain $\{P_i\}_{i \in I}$ of subsets of $\mathcal{S}$ such that $P = \bigcup_i P_i$. Now

$$\begin{aligned} wp(\alpha, \bigcup_i P_i) &= ([\alpha] : (\bigcup_i P_i)')' \\ &= ([\alpha] : (\bigcap_i (P_i)')' \\ &\supseteq (\bigcap_i [\alpha] : (P_i)')' \text{ (by Theorem (1.1)(vii) since } \forall i, \bigcap_i P_i \subseteq P_i) \\ &= \bigcup_i ([\alpha] : (P_i)')' \\ &= \bigcup_i wp(\alpha, P_i). \end{aligned}$$

For the reverse inclusion, take $x \in wp(\alpha, \bigcup_i P_i)$ arbitrarily. Then $\forall y \in U$, if $(x, y) \in [\alpha]$ then $y \in \bigcup_i P_i$. But by (1.1)(iii), there are only finitely many y such that $(x, y) \in [\alpha]$. So since $P_i \subseteq P_{i+1}$ for each $i \in I$, there is some $i \in I$ such that if $(x, y) \in [\alpha]$ then $y \in P_i$. Hence for some $i \in I$, $x \in ([\alpha] : (P_i)')' = wp(\alpha, P_i)$, so $wp(\alpha, \bigcup_i P_i) \subseteq \bigcup_i wp(\alpha, P_i)$. Thus $wp(\alpha, \bigcup_i P_i) = \bigcup_i wp(\alpha, P_i)$. Therefore $wp(\alpha, -)$ is continuous at any $P \subseteq \mathcal{S}$. □

In general, the converse to Theorem (5)(d) does not hold. As a counterexample within this context the relational representation of the example given in Gries ([1981] p 111) suffices.

But the converse *does* hold for deterministic programs (that is, programs which from any initial state s can proceed in exactly one way).

(6) A program α is said to be *deterministic* iff $[\alpha]$ is a function.

The weakest precondition for a deterministic program α with respect to a postcondition Q can be reformulated as:

(7) $wp(\alpha, Q) = [\alpha] : Q$ (α deterministic)

(Blikle ([1977] p 30) provides an analogous definition.) Then we have:

(8) **Theorem** *For any deterministic program* α

(Gries [1981] (7.7)) $wp(\alpha, Q) \bigcup wp(\alpha, R) = wp(\alpha, Q \bigcup R)$.

Proof Using the distributivity of the Peirce product over unions (Theorem (1.1)(ii)),

$wp(\alpha, Q \bigcup R) = [\alpha] : (Q \bigcup R) = [\alpha] : Q \bigcup [\alpha] : R = wp(\alpha, Q) \bigcup wp(\alpha, R)$. □

The question now is: does the weakest precondition predicate transformer defined in (4) correspond to Dijkstra's weakest precondition predicate transformer? Note that $wp(\alpha, -)$ in (4) is defined in terms of an execution relation $[\alpha]$, while $wp(\alpha, -)$ in Dijkstra is postulated. To answer this question I define an execution relation for each command in Dijkstra's guarded command language (Chapter 2.2) and use the properties in (1) of execution relations to try to derive the formulae in Dijkstra's algebra of weakest preconditions. If this can be done, then we will have shown the consistency of Dijkstra's algebra of weakest preconditions in our relational model for nondeterministic programs.

Recall from §1 (p 110–112) the binary relations associated with the programs in Dijkstra's guarded command language. Namely:

(a) $[skip] = \{(x, y) \mid x = y\} = I$.

(b) $[abort] = \emptyset$.

(c) For any program variable X_k and any well-defined expression e,

$[X_k := e] = \{(x, y) \mid y(k)$ is the value of e in state x and for $j \neq k$, $x(j) = y(j)\}$

(d) $[if\ B_1 \to \alpha_1 \ [] \ B_2 \to \alpha_2\ fi] = [B_1?]\,;[\alpha_1] \cup [B_2?]\,;[\alpha_2]$
where $[B_i?] = \{(x,x) \mid x \in B_i\}$ for $i = 1,2$.
Also $[if\ B\ do\ \alpha] = [B?]\,;[\alpha]$, and $[if\ B\ do\ \alpha\ else\ \beta\ fi] = [B?];[\alpha] \cup [\neg B?];[\beta]$.

(e) $[while\ B\ do\ \alpha] = [if\ B\ do\ \alpha]^*\,;[\neg B?] = (\bigcup_{n\geq 0}[if\ B\ do\ \alpha]^n);[\neg B?]$
where $[if\ B\ do\ \alpha]^0 = I$, and $[if\ B\ do\ \alpha]^{n+1} = [if\ B\ do\ \alpha]^n\,;[if\ B\ do\ \alpha]$, for $n \geq 0$.

Of these only $[skip]$, $[X_k := e]$ and $[if\ B\ do\ \alpha\ else\ \beta\ fi]$ are execution relations. The others are not total relations and hence do not satisfy condition (1)(i) for execution relations. In particular,

(b) dom $[abort] = \emptyset$,

(d) $\forall x \in \neg B_1 \cap \neg B_2$, $\{y \mid (x,y) \in [if\ B_1 \to \alpha_1 \ [] \ B_2 \to \alpha_2\ fi]\} = \emptyset$.
$\forall x \in \neg B$, $\{y \mid (x,y) \in [if\ B\ do\ \alpha]\} = \emptyset$, and

(e) since $[if\ B_1 \to \alpha_1 \ [] \ B_2 \to \alpha_2\ fi]$ is not an execution relation, $[while\ B\ do\ \alpha]$ captures only the (cleanly) terminating executions of $while\ B\ do\ \alpha$.

To rectify this situation the execution relations are taken to be:

(b)′ $[abort] = \{(x,\perp) \mid x \in U\}$

(d)′ $[if\ B_1 \to \alpha_1 \ [] \ B_2 \to \alpha_2\ fi]$
$=[B_1?]\,;[\alpha_1] \cup [B_2?]\,;[\alpha_2] \cup\{(x,\perp) \mid x \in \neg B_1 \cap \neg B_2\}$
$= [B_1?]\,;[\alpha_1] \cup [B_2?]\,;[\alpha_2] \cup[(\neg B_1 \cap \neg B_2)?]\,;[abort]$ (by Lemma (1.2)(c))
where $[B_i?] = \{(x,x) \mid x \in B_i\}$ for $i = 1,2$.
(Note that $[if\ B_1 \to \alpha_1 \ [] \ B_2 \to \alpha_2\ fi]$ is an execution relation only if $[\alpha_1]$ and $[\alpha_2]$ are.)
'$if\ B\ do\ \alpha$' can be taken to mean '$if B \to \alpha \ [] \ \neg B \to abort fi$' with execution relation $[if\ B\ do\ \alpha] = [B?];[\alpha] \cup[\neg B?]\,;[abort]$.
Alternatively to be consistent with the Dijkstra/Gries type exposition we could take '$if\ B\ do\ \alpha$' to mean '$if B \to \alpha \ [] \ \neg B \to skip\ fi$' with execution relation $[if\ B\ do\ \alpha]$ $= [B?]\,;[\alpha] \cup[\neg B?];[skip]$. (I will use the latter.)

(e)′ $[while\ B\ do\ \alpha] = [if\ B\ do\ \alpha]^* = (\bigcup_{n\geq 0}[if\ B\ do\ \alpha]^n)$

where $[if\ B\ do\ \alpha]^0 = I$, and

$[if\ B\ do\ \alpha]^{n+1} = [if\ B\ do\ \alpha]^n; [if\ B\ do\ \alpha]$, for $n \geq 0$.

(This simplification is possible due to the definition of $[if\ B\ do\ \alpha]$.)

Now using these definitions we can prove the formulae in Dijkstra's algebra of weakest preconditions.

(9) **Theorem** *For any predicate* Q

(a) (Gries [1981] (8.1)) $wp(skip, Q) = Q$.

(b) (Gries [1981] (8.2)) $wp(abort, Q) = \emptyset$.

Proof

(a) $wp(skip, Q) = (I : Q')' = (Q')' = Q$ (by Theorem (1.1)(iv)).

(b) $wp(abort, Q) = ([abort] : Q')'$

$= \{x \mid (\forall y \in \mathcal{S})[(x, y) \in [abort] \Rightarrow y \in Q\}$

$= \{x \mid (\forall y \in \mathcal{S})[y = \perp \Rightarrow y \in Q]\}$

$= \{x \mid \perp \in Q\}$

$= \emptyset$. (since predicates do not contain $\perp$) □

We now come to composition: the point where most relational models fail.

(10) **Theorem** *For any programs* α *and* β*, and any predicate* Q*,*

$$wp(\alpha; \beta, Q) = wp(\alpha, wp(\beta, Q)).$$

Proof Since $\text{ran}[\alpha] \subseteq \text{dom}[\beta] = \mathcal{S} \bigcup \{\perp\}$, this proof follows from Theorem (4.2). □

For the assignment statement, I assume well-definedness and simply show that

(11) **Theorem** *For any program variable* X_k*, expression* e *and predicate* Q*:*

$$wp(\text{`}X_k := e\text{'}, Q) = Q[X_k/e].$$

Proof Let α denote the statement '$X_k := e$' then $wp(\alpha, Q) = ([\alpha] : Q')'$ since $[\alpha] : \mathcal{S} = \mathcal{S}$. Now by definition (in §1, p 110)

$([\alpha] : Q')' = \{s \mid (\forall t \in \mathcal{S})[(t(k) = e \text{ and } s(j) = t(j) \text{ for } j \neq k) \Rightarrow t \in Q]\}$

where '$t \in Q$' means 'Q is true in state t' that is '$Q[X_1 \mid t(1), \ldots, X_n \mid t(n)]$ is true'.

Let $s \in ([\alpha] : Q')'$ and let t be any state (actually the unique state) in which $s(j) = t(j)$ whenever $j \neq k$ and $t(k) = e$. Then $t \in Q$. Thus $Q[X_1 \mid t(1), \ldots, X_k \mid t(k), \ldots, X_n \mid t(n)]$ is true. That is,

$Q[X_1 \mid s(1), \ldots, X_{k-1} \mid s(k-1), X_k \mid e, X_{k+1} \mid s(k+1), \ldots, X_n \mid s(n)]$ is true.

Hence $s \in Q[X_k \mid e]$, so $wp(\alpha, Q) = ([\alpha] : Q')' \subseteq Q[X_k \mid e]$.

For the reverse direction, let $s \in Q[X_k \mid e]$, then $Q[X_k \mid e]$ is true in state s. That is, $Q[X_1 \mid s(1), \ldots, X_{k-1} \mid s(k-1), X_k \mid e, X_{k+1} \mid s(k+1), \ldots, X_n \mid s(n)]$ is true. Take any state t in which $t(j) = s(j)$ whenever $j \neq k$ and $t(k) = e$. Then

$Q[X_1 \mid t(1), \ldots, X_{k-1} \mid t(k-1), X_k \mid t(k), X_{k+1} \mid t(k+1), \ldots, X_n \mid t(n)]$ is true.

Thus Q is true in state t, that is $t \in Q$. Hence $s \in ([\alpha] : Q')' = wp(`X_k := e', Q)$ so $Q[X_k \mid e] \subseteq wp(`X_k := e', Q)$. □

It is not surprising that this proof is *not equational* since to model the assignment statement (in §1, p 110) the view of a state as an individual was inadequate — we had to descend to a level where a state is viewed as a vector (or string) over some other individuals (values of program variables).

We now come to the IF construct. Since a special relation *test* is used to model IF, I need to determine its weakest precondition.

(12) **Lemma** *For any predicates B and Q, $wp(B?, Q) = B \cap Q$*

Proof Recall $[B?] : Q = B \cap Q$ (Lemma (1.2)(a)). Now $wp(B?, Q) = ([B?] : \mathcal{S}) \cap ([B?] : Q')' = (B \cap \mathcal{S}) \cap (B \cap Q')' = B \cap (B' \cup Q) = B \cap Q$. □

I also need to determine the weakest precondition of the nondeterministic choice operator $\cup$.

(13) **Lemma** *For any programs α and β, and any predicate Q,*

$$wp(\alpha \cup \beta, Q) = wp(\alpha, Q) \cap wp(\beta, Q).$$

Proof

$wp(\alpha \cup \beta, Q) = ([\alpha \cup \beta] : Q')' = (([\alpha] \cup [\beta] : Q')' = ([\alpha] : Q' \cup [\beta] : Q')'$

$= ([\alpha] : Q')' \cap ([\beta] : Q')' = wp(\alpha, Q) \cap wp(\beta, Q)$ □

I also used a relation $[(\neg B_1 \cap \neg B_2)?]; [abort] = \{(x, y) \mid x \in \neg B_1 \cap \neg B_2 \text{ and } y = \perp\}$ (by

Lemma 1.2(c)). Let $[noguard]$ denote $[(\neg B_1 \bigcap \neg B_2)?];[abort]$. Then

(14) **Lemma** For any predicates B_1 and B_2, $wp(noguard, Q) = B_1 \bigcup B_2$.

Proof

$$\begin{aligned} wp(noguard, Q) &= ([noguard] : Q')' \\ &= \{x \mid (\forall y)[(x,y) \in [noguard] \Rightarrow y \in Q]\} \\ &= \{x \mid (\forall y)[(x \in \neg B_1 \bigcap \neg B_2 \text{ and } y = \perp) \Rightarrow y \in Q]\} \\ &= \{x \mid (\forall y)[x \in B_1 \bigcup B_2 \text{ or } y \neq \perp \text{ or } y \in Q]\} \\ &= \{x \mid x \in B_1 \bigcup B_2\} \\ &= B_1 \bigcup B_2 \end{aligned}$$ □

For the IF operator we need to verify Gries [1981] (10.3).

(15) **Theorem** *For any programs α_1 and α_2, and any predicates B_1, B_2, and Q,*

$wp(if\ B_1 \rightarrow \alpha_1 \ [] \ B_2 \rightarrow \alpha_2\ fi, Q)$

$= (B_1 \bigcup B_2) \bigcap (\neg B_1 \bigcup wp(\alpha_1, Q)) \bigcap (\neg B_2 \bigcup wp(\alpha_2, Q))$

Proof

$wp(if\ B_1 \rightarrow \alpha_1 \ [] \ B_2 \rightarrow \alpha_2\ fi, Q)$

$= (([B_1?];[\alpha_1] \bigcup [B_2?];[\alpha_2] \bigcup [noguard]) : Q')'$

$= ([B_1?];[\alpha_1] : Q' \bigcup [B_2?];[\alpha_2] : Q' \bigcup [noguard] : Q')'$ (by Theorem (1.1) (iii))

$= ([B_1?];[\alpha_1] : Q')' \bigcap ([B_2?];[\alpha_2] : Q')' \bigcap ([noguard] : Q')'$

$= ([B_1?] : ([\alpha_1] : Q'))' \bigcap ([B_2?] : ([\alpha_2] : Q'))' \bigcap (B_1 \bigcup B_2)$ (by Theorem (1.1)(i))

$= (B_1 \bigcap ([\alpha_1] : Q'))' \bigcap (B_2 \bigcap ([\alpha_2] : Q')' \bigcap (B_1 \bigcup B_2)$ (by Lemma (1.2)(a)

$= (\neg B_1 \bigcup ([\alpha_1] : Q')') \bigcap (\neg B_2 \bigcap ([\alpha_2] : Q')') \bigcap (B_1 \bigcup B_2)$

$= (B_1 \bigcup B_2) \bigcap (\neg B_1 \bigcup wp(\alpha_1, Q)) \bigcap (\neg B_2 \bigcup wp(\alpha_2, Q))$ □

For '$if\ B\ do\ \alpha$' and '$if\ B\ do\ \alpha\ else\ \beta\ fi$' (defined above) we get as special cases from (15):

(16) **Corollary** *For any programs α and β, and any predicates B and Q,*

$wp(if\ B\ do\ \alpha, Q) = (B \bigcap wp(\alpha, Q)) \bigcup (\neg B \bigcap Q)$ and

$wp(if\ B\ do\ \alpha\ else\ \beta\ fi, Q) = (B \bigcap wp(\alpha, Q)) \bigcup (\neg B \bigcap wp(\beta, Q))$ □

I can also prove Gries ([1981] (10.5))

(17) **Theorem** *For any predicates . and Q,*

$X \subseteq wp(if\ B_1 \rightarrow \alpha_1\ []\ B_2 \rightarrow \alpha_2\ fi, Q)$ iff (i) $X \subseteq B_1 \bigcup B_2$

(ii) $X \bigcap B_1 \subseteq wp(\alpha_1, Q)$, and

(iii) $X \bigcap B_2 \subseteq wp(\alpha_2, Q)$, and

Proof By mutual implication as follows:

Left to right: If $s \in X$ then $s \in (B_1 \bigcup B_2) \bigcap (\neg B_1 \bigcup wp(\alpha, Q)) \bigcap (\neg B_2 \bigcup wp(\beta, Q))$, so $s \in (B_1 \bigcup B_2)$. If $s \in X \bigcap B_1$ then $s \in wp(if\ B_1 \rightarrow \alpha_1\ []\ B_2 \rightarrow \alpha_2\ fi, Q)$; so by Theorem (15) $s \in \neg B_1 \bigcup wp(\alpha_1, Q)$. But $s \in B_1$; hence $s \in wp(\alpha_1, Q)$. Similarly, if $s \in X \bigcap B_2$ then $s \in wp(\alpha_2, Q)$.

Right to left: Suppose $s \in X$. Then by (i) $s \in B_1 \bigcup B_2$. So either $s \in B_1$ or $s \notin B_1$. If $s \in B_1$ then $s \in X \bigcap B_1 \subseteq wp(\alpha_1, Q)$; so $s \in wp(\alpha_1, Q) \subseteq \neg B_1 \bigcup wp(\alpha_1, Q)$. If $s \notin B_1$ then $s \in \neg B_1 \subseteq \neg B_1 \bigcup wp(\alpha_1, Q)$. Similarly, $s \in \neg B_2 \bigcup wp(\alpha_2, Q)$. Hence $s \in [B_1 \bigcup B_2] \bigcap [\neg B_1 \bigcup wp(\alpha_1, Q)] \bigcap [\neg B_2 \bigcup wp(\alpha_2, Q)]$

So $s \in wp(if\ B_1 \rightarrow \alpha_1\ []\ B_2 \rightarrow \alpha_2\ fi, Q)$. □

I conclude this section by reporting that I was unable to verify Gries's [1981] law (11.2) for the iterative command using some seemingly obvious execution relations to characterise the semantics of '*while B do* α'. However, I will draw the attention of the reader to Guerreiro's ([1980], [1981], [1982]) work on characterising the semantics of the iterative command. This is as follows.

The execution relation for *while B do* α is defined as the union of two execution relations: one capturing the terminating executions of *while B do* α and the other capturing the non-terminating ones. That is,

$$[while\ B\ do\ \alpha] = [if\ B\ do\ \alpha]; [\neg B?] \bigcup \{(x, \perp) \mid x \in W\}$$

where W is a predicate characterising the set of initial states from which *while B do* α does not terminate. This predicate W is then the set of all initial states for which there is no integer k so that *while B do* α will terminate in k or fewer iterations or from which α does not terminate. Hence W is a subset of B. This means that an iteration starting in a state in W must either terminate in W or not terminate at all. That is,

$$\forall x \in U,\quad x \in W \Rightarrow (\exists y)[(x,y) \in [if\ B\ do\ \alpha] \text{ and } (y \in W \text{ or } y = \perp)]$$

So $\forall x \in U,\quad x \in W \;\Rightarrow\; x \in [if\ B\ do\ \alpha] : (W \bigcup \{\perp\})$.

That is, $W \subseteq [if\ B\ do\ \alpha] : (W \bigcup \{\perp\})$.

The problem then is to find a nonrecursive definition of this set W. Guerreiro's [1980] proposal is to characterise the termination of the iterative command by solving the equation

$$W = W \bigcap [if\ B\ do\ \alpha] : (W \bigcup \{\perp\}).$$

I have not been able to prove or disprove the claims made in this paper. Towards the end of my study I became aware of the paper Guerreiro [1981] (referenced in Guerreiro [1982]) which contains a detailed study of the semantics of the iterative command. Unfortunately this paper is written in Portuguese and time constraints did not allow me to obtain an English translation thereof or to contact the author.

In this section I have verified all the formulae, except that for the iterative command, in the algebra of weakest preconditions using the relational model. All the reasoning was done equationally except that for the assignment command. My conclusion is that relational model chosen in §4 provides useful tools for reasoning about most nondeterministic programs but it is not convenient for dealing with the iterative command.

Chapter 5

Predicate Transformers

The previous chapter showed how an operation on or a relation between the *elements* of a state space can be used to formalise the behaviour of a program. Another approach is to use operations on or relations between *predicates*. This chapter deals with the use of operations on predicates. A unary operation from predicates to predicates is called a *predicate transformer* . The idea of formalising a program as a predicate transfomer was introduced in Dijkstra [1975] and in his classic monograph [1976], in which he launched the whole enterprise of weakest precondition semantics of programs for total correctness. (In Chapter 2, I discussed this approach to the semantic characterisation of Dijkstra's guarded command language.) There are two ways of associating a predicate transformer with a program: simply to postulate one or constructively to define one.

The first technique, due to Dijkstra ([1975], [1976]), involves postulating for a notion of correctness a set of constraints which must be satisfied by a predicate transformer. These constraints are called *healthiness properties* (Hoare [1978]). Then the idea is to postulate for each program in a programming language a predicate transformer which satisfies the healthiness properties. For example, Dijkstra's ([1975], [1976]) healthiness properties (presented in Chapter 2.2) are constraints for a useful total correctness predicate transformer. He used these properties to postulate for each program in his guarded command language a total correctness weakest precondition predicate transformer. The discussion in Chapter 2 is restricted to the total- and partial correctness weakest precondition predicate transform-

ers, '$wp(\alpha, -)$' and '$wlp(\alpha, -)$', and to programs written in Dijkstra's guarded command language. My aim in §1 is to address (but not answer) the question of how to postulate predicate transformers for all possible notions of correctness and for programs in general. I also identify the predicate transformers which have been investigated in the literature.

The second approach involves presenting an operational model for programs and defining the semantics of a program in terms of the operational representations. Here the idea is to derive for a program a predicate transformer based on the operational representation of the program. For example, in Chapter 4 a relational model based on *execution relations* was given. The semantics of each program in Dijkstra's guarded command language was defined in terms of execution relations. Then for each program a total correctness weakest precondition predicate transformer was derived from the associated execution relation.

Since a predicate transformer arising from a program is constructively defined (based on the program) while a predicate transformer satisfying a set of healthiness properties is simply postulated it is reasonable to ask: How, if at all, are the two definitions related? For this we need some sort of *representation result*, to the effect that: *Any predicate transformer* $F : \mathcal{P}(U) \to \mathcal{P}(U)$ *which satisfies a set of healthiness properties arises from some program.*

A primary concern in the study of predicate transformers is to develop a uniform framework. The fundamental question is: What, if anything, is the relationship between the operational and predicate transformer representations of a program? Knowing that they are related through a program, we enquire whether there is some way of translating between the two. This question has been addressed in a relational context by, for example, Guerreiro ([1980], [1982]), Majster-Cederbaum [1980], Jacobs and Gries [1985] as well as Holt [1991], but as yet there is no coherent account of how or when these two descriptions may be obtained from each other.

My aim in this chapter is to show that there is an elegant means of both answering this question and solving the representation problem. The formal apparatus required is that of a *power construction*. This concept is introduced and investigated in §2. In §3 I investigate

within this unifying framework how representations of nondeterministic programs as *execution relations* (introduced in Chapter 4.5) allow us to construct nine different predicate transformers that capture different aspects of our intuition of the behaviour of programs. I show how the healthiness properties of these predicate transformers can be obtained from the properties of the execution relations involved. The main result, a representation theorem, is proved in §4 by an application of the power construction.

5.1 Postulating Predicate Transformers

My aim in this section is to point out how predicate transformers for programs may be postulated.

In a practical situation it is often difficult to predict the final state of a program given just the program and an initial state. Instead we ask: given a set of final states, what corresponding set of initial states would produce those final states via an execution of the program? This means that a program determines a relationship between predicates. From this viewpoint, we can describe *what* a program must do without saying *how* it is to accomplish a specific task by simply associating with a specific postcondition for the program a corresponding precondition. To capture the semantics of a program, α, under a notion of correctness, we want for any given postcondition Q the set of all initial states s from which α has some execution property and some final state property. This predicate is called the *weakest precondition of α with respect to Q*, for that notion of correctness. It is a 'precondition' in the sense that it characterises a set of initial states; it is the 'weakest' precondition in the sense that it contains every set of initial states from which α has a chosen execution property and a chosen final state property. (In Chapter 3, I discussed two different weakest preconditions for each notion of correctness. Namely, for partial correctness: $wlp(\alpha, Q)$ and $wlpa(\alpha, Q)$; for total correctness: $wp(\alpha, Q)$ and $wpa(\alpha, Q)$; for general correctness: $gwp(\alpha, Q)$, $gwpa(\alpha, Q)$ (Jacobs and Gries [1985]).)

A program α may have many possible postconditions, so we want some sort of rule for deriving the weakest precondition for a notion of correctness of α with respect to any given

postcondition of α. The current proposal is to define for every program α, a predicate transformer mapping any given predicate Q (viewed as a postcondition) onto the weakest precondition (for a notion of correctness) of α with respect to Q. Such a mapping is called the *weakest precondition predicate transformer* of α for that notion of correctness. (Examples of mappings defining sets of initial states of a program α given desired sets of final states interpreted under demonic nondeterminism are $wlp(\alpha, -)$, $wp(\alpha, -)$, $gwp(\alpha, -)$; while those interpreted under angelic nondeterminism are $wlpa(\alpha, -)$, $wpa(\alpha, -)$, $gwpa(\alpha, -)$ (Jacobs and Gries [1985]).)

Following Dijkstra [1975], most authors denote a weakest precondition (for a notion of correctness) of a program α with respect to a postcondition Q by a notation of the form '$F(\alpha, Q)$'. But this notation is somewhat misleading: a predicate transformer appears to be an operation '$F : programs \times predicates \rightarrow predicates$' mapping a program and a predicate onto a predicate. In fact, predicate transformers are *indexed* by programs, so the mapping '$F(\alpha, -) : predicates \rightarrow predicates$' is a predicate transformer for a program α. To avoid such confusion, I will use 'F_α' to denote a predicate transformer for a program α.

An alternative approach to capturing the semantics of a program α under a notion of correctness is to introduce a predicate which characterises for a program α and any given precondition P the smallest set of possible final states such that α has some execution property from every state in P which will lead to one of them. By analogy with the previous proposal this predicate is called the *strongest postcondition of α with respect to P* for that notion of correctness. (For example, under a demonic interpretation of nondeterminism this predicate could denote the set of all final states such that *every* execution of α from some state in P is guaranteed to reach one of them; under an angelic interpretation of nondeterminism it could denote the set of all final states such that *at least one* execution of α from some state in P is guaranteed to reach one of them.) It is a 'postcondition' in the sense that it characterises a set of final states; it is the 'strongest' postcondition in the sense that it is contained in every set of final states which α can reach from a state in P. (Note that Chapter 3 was devoted to formulating notions of correctness in terms of weakest preconditions. Another approach is to use stongest postconditions.) The idea, here, is that the meaning

of any program α is entirely characterised by defining a predicate transformer which maps any given predicate P (viewed as a precondition) onto the strongest postcondition for a notion of correctness of α with respect to P. This predicate transformer is then called the *strongest postcondition predicate transformer* of the program for that notion of correctness. (Examples of mappings defining sets of initial states of a program α given desired sets of final states interpreted under demonic nondeterminism are $slp(-,\alpha)$, $sp(-,\alpha)$, $gsp(-,\alpha)$; while those interpreted under angelic nondeterminism are $slpa(-,\alpha)$, $spa(-,\alpha)$, $gspa(,\alpha)$ (Jacobs and Gries [1985]).)

If a predicate transformer is to be useful in the definition of programming languages it should allow us to describe computationally meaningful constructs rather than arbitrary ones. That is, a predicate transformer should be useful for showing that a program has a particular execution property or for answering questions of program correctness. Recall from Chapter 3.2 that a notion of correctness arises from constraints on all or some states s with property P in which the program is activated. Since a predicate transformer should capture a notion of correctness, it must satisfy some constraints. The contraints on predicate transformers for a notion of correctness are commonly called *healthiness properties*. In particular, the healthiness properties on a predicate transformer F_α of a program α could include properties describing:

(i) the outcome of applying F_α to the empty predicate

(ii) the range of F_α

(iii) the monotonicity of F_α

(iv) how F_α distributes over (at-most-countably-infinite) conjunctions and disjunctions

(v) how the use of $\perp$ is controlled

(vi) the continuity of F_α.

(Note that the original five healthiness properties (Chapter 2.2) postulated in Dijkstra [1975] do not include a property for (v) since nontermination was not explicitly formalised there.)

I will call property (v) a *strictness property* of F_α. Properties (iii) and (iv) are inherent from the definition of F_α; while propeties (i), (ii), (v) and (vi) refer to program execution.

The healthiness properties of a predicate transformer limit the extent to which we can reason about programs. In this connection recall **Table 11** in Chapter 4.3 which tabulates the weakest preconditions of some example programs for notions of general-, total- and partial correctness. From **Table 11**, the program $\alpha_{(i)}$: *skip* (which always terminates cleanly) has the same partial correctness weakest precondition as the program $\alpha_{(ii)}$: if *true* $\longrightarrow$ *skip* [] *true* $\longrightarrow$ *abort* fi (which sometimes terminates cleanly). So partial correctness predicate transformers cannot be used to express guaranteed termination.

From **Table 11**, the program $\alpha_{(iii)}$: *abort* (which always terminates messily) has the same total correctness weakest precondition as the program $\alpha_{(ii)}$: if *true* $\longrightarrow$ *skip* [] *true* $\longrightarrow$ *abort* fi (which sometimes terminate cleanly). So total correctness predicate transformers cannot be used to distinguish possible clean termination from guaranteed messy termination.

If we wish to reason carefully about programs that may not terminate, only the general correctness predicate transformers, (namely, '$gwp(\alpha, -)$', '$gsp(-, \alpha)$', '$gwpa(\alpha, -)$' and $gspa(-, \alpha)$') introduced in Jacobs and Gries [1985], are of interest. As motivated in Holt [1991], for any program α, '$gsp(-, \alpha)$' and '$gwpa(\alpha, -)$' are of interest. '$gsp(-, \alpha)$' is intuitively appealing in that for any given set of initial states it describes what can be observed as a final state of α. Often a program debugger observes the final states from various executions of a program and then tries to find the set of states from which execution of the program must have begun. In such a situation, '$gwpa(\alpha, -)$' is useful.

In an earlier investigation, Blikle [1981] introduced a notion of global correctness to capture the error of messy termination. It is interesting that Blikle's ([1981] p 204) global correctness predicate transformer is exactly what Jacobs and Gries [1985] independently introduced as '$wpa(\alpha, -)$'.

Dijkstra's predicate transformers '$wlp(\alpha, -)$' and '$wp(\alpha, -)$' are shown to be best suited to practical programming in, for example, Hoare [1978] and Guerreiro [1980]. Hoare [1978]

uses *trace semantics* of programs to postulate predicate transformers and their healthiness properties; while Guerreiro [1980] uses relational semantics. Other characterisations of Dijkstra's weakest precondition predicate transformer are given in De Roever [1976] and Wand [1977].

Several authors, including Hoare [1978] and Holt [1991], have pointed out that Dijkstra's [1976] healthiness properties for '$wp(\alpha, -)$' *do not* exclude programs which, though computable, are impractical to implement and *do* exclude some programs with reasonably realistic implementations, namely those which allow *unbounded nondeterminism.* In the first instance, Hoare's [1978] Bowdlerize program is not excluded. This program has a predicate transformer which scans a predicate and replaces every occurrence of a particular obnoxious sequence of letters with a zero. Although this predicate transformer satisfies the healthiness conditions for '$wp(\alpha, -)$' it does not map different predicates characterising the same sequence of letters (which by definition is finite) to the same predicate simply because there are infinitely many possible series of assignments which can implement the program. Secondly, as originally recognised in Dijkstra ([1976] p 76,77), the weakest precondition predicate transformer for the program α (in Chapter 2.2, p 41) which assigns to variable, say x, an arbitrary positive integer, fails to be continuous. In particular, for a chain $\{P_i\}_{i\geq 0}$ of predicates P_i such that for each $i \geq 0$, P_i characterises the set of states in which $0 < x < i$, it is possible for $wp(\alpha, \bigcup_i P_i)$ to hold but no $wp(\alpha, P_i)$ to hold since 'no a priori upper bound for the final value of x can be given'. Recall (from Chapter 2.2, p 40, 41) this program α terminates *weakly* (that is, there is no integer k such that α is guaranteed to terminate in less than k iterations).

The conclusion drawn in Chapter 2.2 (p 44) was that Dijkstra's weakest precondition semantics do not provide sufficient information to distinguish between guaranteed nontermination and weak termination. It therefore seems reasonable to ask: Could the abovementioned distinction be made if a special state $\perp$ representing the 'state of nontermination' is added to $\mathcal{S}$? Let $wp_\perp(\alpha, -)$ be the weakest precondition predicate transformer for α with respect to $\mathcal{S}_\perp = \mathcal{S} \bigcup \{\perp\}$, and let $wp(\alpha, -)$ have their usual meanings with respect to $\mathcal{S}$. That is, for $Q \subseteq \mathcal{S}$, $wp_\perp(\alpha, Q) = wp(\alpha, Q)$. Then what we want is an interpretation for $wp_\perp(\alpha, \{\perp\})$.

The fundamental question is: What set of healthiness conditions will capture exactly the predicate transformer representations of programs, *including* those which allow unbounded nondeterminism? Guerreiro [1982] suggested an explicit formalisation of the states of nontermination and two extra healthiness properties: one expressing that a program is guaranteed to terminate if and only if it is guaranteed not to go on forever and the other expressing that a program is guaranteed to terminate from a state only if it is guaranteed to start in the state. I show, in the §5, that Dijkstra and Scholten's [1990] new set of healthiness conditions (presented in Chapter 2.4, p 52) can be used. Their theory does not have an explicit formalisation of the state of nontermination.

Nelson [1989] found the normality of '$wp(\alpha, -)$' (commonly called the *law of excluded miracle*, $wp(\alpha, \emptyset) = \emptyset$) excluded some programs whose implementation would involve much backtracking but would nevertheless be reasonable. In particular, programs which may not start from a given state are excluded. However, including such programs into a programming language raises a rather subtle question concerning nondeterminism: Does the notion of nondeterminism allow a program to sometimes proceed from an initial state and sometimes not?

5.2 A Power Construction

The main algebraic notion in this chapter is the notion of the *power algebra* of a relational structure, as initiated by Jònsson and Tarski [1951]. In the context of this thesis I only need to consider the case of binary relations over a set U. All the apparatus needed is introduced in this section.

Given a binary relation $R \subseteq U^2$, define a unary operation $R^\dagger : \mathcal{P}(U) \longrightarrow \mathcal{P}(U)$ by

(1) $R^\dagger(X) = \{x | (\exists y)[(x, y) \in R \text{ and } y \in X]\}$

for any $X \subseteq U$.

This unary operation $R^\dagger$ is called the *power operation* of R. Note that for every binary

relation $R \subseteq U^2$, $R^{\uparrow}(\emptyset) = \emptyset$ and $R^{\uparrow}(\bigcup_i X_i) = \bigcup_i R^{\uparrow}(X_i)$. Any mapping $F : \mathcal{P}(U) \longrightarrow \mathcal{P}(U)$ with these properties is called respectively *normal* and *completely additive*. For any relational structure $\mathcal{U} = (U, \mathcal{R})$ where $\mathcal{R}$ is a set of binary relations $R \subseteq U^2$, we call the structure $(\mathcal{P}(U), \mathcal{F})$ obtained by powering every relation $R \in \mathcal{R}$, the *power algebra* of $\mathcal{U}$. (Recalling definition 4.1 (e) we see that the set in (1) captures exactly what Brink [1981] calls the *Peirce product* (introduced by Peirce [1870]) of the relation R and the set X.)

The operation of powering a relation has an inverse. That is,

For any normal, completely additive unary operation $F : \mathcal{P}(U) \longrightarrow \mathcal{P}(U)$ there is a relation $F^{\downarrow} \subseteq U^2$ defined by

(2) $(x, y) \in F^{\downarrow}$ iff $x \in F(\{y\})$

for any $x, y \in U$.

I will call this *lowering* a unary operation F to get its *underlying relation* $F^{\downarrow}$. These two operations allow us to translate back and forth between binary relations and normal completely additive operations. More precisely,

(3) **Theorem** (Jònsson and Tarski [1952])

(i) *For any relation* $R \subseteq U^2$, $R^{\uparrow\downarrow} = R$.

(ii) *For any normal, completely additive unary operation* $F : \mathcal{P}(U) \longrightarrow \mathcal{P}(U)$, $F^{\downarrow\uparrow} = F$.

For any unary operation $F : \mathcal{P}(U) \longrightarrow \mathcal{P}(U)$ we can define a unary operation $F^* : \mathcal{P}(U) \longrightarrow \mathcal{P}(U)$ by

(4) $F^*(X) = [F(X')]'$

for any $X \subseteq U$.

I will call F^* the *dual operation* of F. (Note that the dual of the dual of a unary operation is the original operation and the properties of F^* are dual to those of F.) If in addition F is normal and completely additive then $F^*(U) = U$ and $F^*(\bigcap_i X_i) = \bigcap_i F^*(X_i)$. Any unary

operation with these properties is called respectively *full* and *completely multiplicative.* (Note that for a relation R, $R^{\uparrow *}$ coincides with Peirce's [1870] involution operation (mentioned in Chapter 4.1, p112).)

Let $F, G : \mathcal{P}(U) \longrightarrow \mathcal{P}(U)$ be any unary operations. Then if

(5) $\forall X, Y \subseteq U, \quad F(X) \subseteq Y'$ iff $G(Y) \subseteq X'$

I call G a *converse* of F (as in Dijkstra and Scholten ([1990] p 201). In general, the converse G of a function F is not a function. If it is we call G the *inverse* of F and denote it by F^{-1}.

Note that for any binary relation $R \subseteq U^2$ we can define the power operation $R^{\smile\uparrow}$ of its converse $R^{\smile}$. We have to be unusually careful about the terminology used for this power operation. Holt ([1991] p 12) calls $R^{\smile\uparrow}$ the 'converse' of $R^{\uparrow}$. Jacobs and Gries ([1985] p 69, 70) introduce the power operation as a boolean operator and call the set representing the converse of a relation R, the 'inverse' of R (denoted by R^{-1}) and hence refer to $R^{\smile\uparrow}$ and $R^{\uparrow}$ as inverses.

The following lemma deals with the translation between properties of binary relations and the corresponding unary operations. Let $U = \mathcal{S} \bigcup \{\perp\}$. I deal with those relations over U called *execution relations* in Chapter 4.5 (p 138) (that is, total relations $R \subseteq U^2$ such that (i) for every $x \in U$, if $(\perp, x) \in R$ then $x = \perp$, and (ii) for every $x \in U$, $\{y \mid (x, y) \in R\}$ is finite). For later convenience some terminology is introduced here.

(6) **Lemma** *Let $F : \mathcal{P}(U) \longrightarrow \mathcal{P}(U)$ be normal and completely additive and let $R \subseteq U^2$ be $F^{\downarrow}$, then the properties of R correspond to F as (ii) to (i) below. Conversely, let $R \subseteq U^2$ be any binary relation and let $F = R^{\uparrow}$, then $F : \mathcal{P}(U) \longrightarrow \mathcal{P}(U)$ is normal and competely additive, and properties of F correspond to properties of R as (i) to (ii) below.*

(a) (i) *F is full.* $[F(U) = U]$

(ii) *R is a total relation.* [dom $R = U$]

(b) (i) *F is strict.* $[\forall X \subseteq U, \perp \in X$ iff $\perp \in F(X)]$

(ii) *R is strict.* $(\forall x \in U)[(\bot, x) \in R \Rightarrow x = \bot]$

(c) (i) $F(\{\bot\}) = \{\bot\}$.

(ii) *R is terminating, that is,* $(\bot, \bot) \in R$.

$[(\forall x \in U)[(x, \bot) \in R \Rightarrow x = \bot]$ *and* $(\forall x \in U)[(\bot, x) \in R \Rightarrow x = \bot]$

(d) (i) $\forall Y \subseteq U, F(Y) = F^*(Y)$.

(ii) *R is deterministic.* $[\forall x, y, z \in U, (x, y) \in R$ and $(x, z) \in R \Rightarrow y = z]$

Proof

(a) Suppose (i) holds and take $x \in U$ arbitrarily. Then (by (i)) $x \in F(U)$. But $F(U) = F(\bigcup_{y \in U}\{y\}) = \bigcup_{y \in U} F(\{y\})$ by complete additivity of F. Hence for some $y \in U$, $x \in F(\{y\})$, so by (2) for some $y \in U$, $(x, y) \in F^{\downarrow}$. Thus $x \in \text{dom}(F^{\downarrow})$, so $U \subseteq \text{dom}(F^{\downarrow})$. Trivially dom $F^{\downarrow} \subseteq U$; hence dom $R =$ dom $F^{\downarrow} = U$.

For the converse, suppose (ii) holds. Take any $x \in U$. Then by (ii) $x \in$ dom R. Thus for some $y \in U$, $(x, y) \in R$ and hence by (2) for some $y \in U$, $x \in R^{\uparrow}(\{y\}) \subseteq R^{\uparrow}(U) = F(U)$ (by monotonicity of $R^{\uparrow}$, $R^{\uparrow}(\{y\}) \subseteq R^{\uparrow}(U)$ since $\{y\} \subseteq U$). So $U \subseteq F(U)$. Trivially $F(U) \subseteq U$ hence $F(U) = U$.

(b) Suppose (i) holds and take any $x \in U$ such that $(\bot, x) \in R$. Then $\bot \in R^{\uparrow}(\{x\})$ and hence (by (i)) $\bot \in \{x\}$ which means $x = \bot$. So if $(\bot, x) \in R$ then $x = \bot$.

For the converse, suppose (ii) holds and take $X \subseteq U$ arbitrarily. Suppose $\bot \in F(X)$. Since $F(X) = F(\bigcup_{x \in X}\{x\}) = \bigcup_{x \in X} F(\{x\})$ by complete additivity of F there is some $x \in X$ such that $\bot \in F(\{x\})$, so for some $x \in X$, $(\bot, x) \in F^{\downarrow}$ and hence (by (ii)) $x \in \{\bot\}$, so $\bot \in \{x\} \subseteq X$. Thus if $\bot \in F(X)$ then $\bot \in X$. On the other hand, suppose $\bot \notin F(X) = \bigcup_{x \in X}(F(\{x\})$ then for every $x \in X$, $\bot \notin F(\{x\})$, so for every $x \in X$, $(\bot, x) \notin F^{\downarrow}$. Hence for every $x \in U$, $(\bot, x) \in F^{\downarrow} \Rightarrow x \notin X$. Then by (ii) $\bot \notin X$. Hence if $\bot \in X$ then $\bot \in F(X)$.

(c) Suppose (i) holds. Then $\bot \in F(\{\bot\})$, so for some $y \in \{\bot\}$, $(\bot, y) \in F^{\downarrow}$; hence $(\bot, \bot) \in F^{\downarrow}$.

For the converse, suppose (ii) holds, that is, $(\bot, \bot) \in R$. Then $\bot \in R^{\uparrow}(\{\bot\})$, so $\{\bot\} \subseteq R^{\uparrow}(\{\bot\})$. On the other hand, let $x \in R^{\uparrow}(\{\bot\})$. Then by (2) $(x, \bot) \in R$ so

by (ii) $x = \perp$ (that is, $x \in \{\perp\}$); hence $R^{\uparrow}(\{\perp\}) \subseteq \{\perp\}$. So $F(\{\perp\}) = R^{\uparrow}(\{\perp\}) = \{\perp\}$.

(d) Suppose (i) holds. Take any $x, y, z \in U$ such that $(x, y) \in R$ and $(x, z) \in R$. Assume $y \neq z$. Then $x \in F(\{y\}) = F^*(\{y\})$ and $x \in F(\{z\}) = F^*(\{z\})$ and hence $x \in F^*(\{y\}) \bigcap F^*(\{z\})$. Then by complete multiplicativity of F^*, $x \in F^*(\{y\} \bigcap \{z\})$, so by (i) $x \in F(\{y\} \bigcap \{z\})$. But $y \neq z$, so $x \in F(\emptyset) = \emptyset$ since F is normal. This contradicts the assumption that $x \in U$. So R is deterministic.

For the converse suppose (ii) holds. Then $\forall x \in U$, $\{y \mid (x, y) \in R\}$ is empty or a singleton, say $\{t\}$. Then $\forall Y \subseteq U$, $R^{\uparrow *}(\mathrm{Y}) = \{x \mid (\exists t)[(x, t) \in R \text{ and } t \in Y]\} = R^{\uparrow}(\mathrm{Y})$. □

A unary operation $F : \mathcal{P}(U) \longrightarrow \mathcal{P}(U)$ is *continuous* if for every increasing chain (under $\subseteq$) $\{X_i\}_{i \in I}$ of subsets of U, $F(\bigcup_i X_i) = \bigcup_i F(X_i)$. Note that for any $R \subseteq U^2$, the complete additivity of $R^{\uparrow}$ implies the continuity of $R^{\uparrow}$. For $R^{\uparrow *}$ we have the following result:

(7) **Lemma** *If* $\forall x \in U$, $\{y \mid (x, y) \in R\}$ *is finite then* $R^{\uparrow *}$ *is continuous.*

Proof Choose any increasing chain $\{X_i\}_{i \in I}$ of subsets of U with $X = \bigcup_i X_i$. Take $x \in R^{\uparrow *}(\bigcup_i X_i)$ arbitrarily. Then for every $y \in U$, if $(x, y) \in R$ then $y \in \bigcup_i X_i$. But by (ii) there are only finitely many y such that $(x, y) \in R$, so since $X_i \subseteq X_{i+1}$ for each $i \in I$, there is some $i \in I$ such that $(\forall y)[(x, y) \in R \Rightarrow y \in X_i]$. Hence for some $i \in I$, $x \in R^{\uparrow *}(X_i)$, so $x \in \bigcup_i R^{\uparrow *}(X_i)$. Thus $R^{\uparrow *}(\bigcup_i X_i) \subseteq \bigcup_i R^{\uparrow *}(x_i)$. On the other hand, $\forall i \in I$, $X_i \subseteq \bigcup_i X_i$, so $\forall i \in I$, $X_i = X_i \bigcap (\bigcup_i X_i)$. Then by complete multiplicativity of $R^{\uparrow *}$, $\forall i \in I$, $R^{\uparrow *}(X_i) = R^{\uparrow *}(X_i \bigcap (\bigcup_i X_i)) = R^{\uparrow *}(X_i) \bigcap R^{\uparrow *}(\bigcup_i X_i)$. So $\forall i \in I$, $R^{\uparrow *}(X_i) \subseteq R^{\uparrow *}(\bigcup_i X_i)$ and hence $\bigcup_i R^{\uparrow *}(X_i) \subseteq R^{\uparrow *}(\bigcup_i X_i)$ (by definition of the least upper bound for sets). So F is continuous at any $X \subseteq U$. □

5.3 Predicate Transformers as Power Operations

In Chapter 4.3 (p 117) I introduced various relational representation methods $\mathbf{R_j(n)}$ (where $\mathbf{j} = 1, 2, 3$; $\mathbf{n} = 1, 2, \ldots, 8$). Recall also from Chapter 3 the notions of general-, total- and

partial correctness. The connections between these representation methods and these notions of correctness are given in Theorem (4.3.1). Namely, representation methods $\mathbf{R_j(5)}$, $\mathbf{R_j(7)}$ and $\mathbf{R_j(8)}$ (for $\mathbf{j} = 1, 2, 3$) are suitable for general correctness; representation methods $\mathbf{R_j(1)}$, $\mathbf{R_j(3)}$, $\mathbf{R_j(5)}$, $\mathbf{R_j(7)}$ and $\mathbf{R_j(8)}$ are suitable for total correctness; representation methods $\mathbf{R_j(2)}$, $\mathbf{R_j(4)}$, $\mathbf{R_j(6)}$, $\mathbf{R_j(7)}$ and $\mathbf{R_j(8)}$ are suitable for partial correctness. There are at least two ways to introduce for a program α predicate transformers for a notion of correctness based on the relational representation of α.

(i) One approach is to use the results of Theorem (4.3.1) to choose one representation method suitable for the notion of correctness and define the corresponding set of binary relations that models the behaviour of nondeterministic programs. Then applying the power construction described in §2, I could derive the power versions of these relations and their properties and use the notions of a dual and a converse to construct three more unary operations for the notion of correctness. This reasoning could by applied *mutatis mutandis* for each notion of correctness. Holt [1991] adopted a similar approach.

(ii) The alternative is to choose one representation method suitable for all the notions of correctness and then define a set of binary relations. (One of $\mathbf{R_j(7)}$, $\mathbf{R_j(8)}$ for $\mathbf{j} = 1, 2, 3$ can be chosen.) Applying the reasoning above I could derive predicate transformers for general correctness. Recall from Chapter 3.5 that the notion of partial correctness arises from general correctness by restricting predicates to those which always contain $\perp$ (because nontermination is always allowed); while the notion of total correctness arises by restricting predicates to those which never contain $\perp$ (because nontermination is never allowed). Using this observation the total and partial correctness predicate transformers can then be obtained by viewing those for general correctness respectively as functions of predicates that do not contain $\perp$ and as functions of predicates that must contain $\perp$. This is the approach adopted by Jacobs and Gries [1985].

I will adopt the second approach because it shows that partial– and total correctness predicate transformers can be constructed in a consistent fashion based on the general correctness execution relations. This is important for technical reasons: it will allow us to define (in

§3) a total correctness predicate transformer in terms of a partial correctness one. We will then have answered the open question addressed in Holt ([1991] p 30). However, it should be noted that much can be gained from the first approach since it less contrived (being a simple application of the power construction).

I will choose for presentation representation method $\mathbf{R_3(7)}$: all kinds of executions are represented and a special symbol $\perp$ is used to denote the state of nontermination. By Theorem (4.3.1) it is suitable for general-, total- and partial correctness in a context where nontermination is equated with messy termination. (Recall from Chapter 4.4 (p 137) $\mathbf{R_3(7)}$ was also found to be well-suited to modelling Dijkstra's weakest precondtion semantics.) Let $U = \mathcal{S} \bigcup \{\perp\}$ be the state space. Let $\mathrm{Rel}(U)$ denote the set of execution relations $R \subseteq U^2$ (based on $\mathbf{R_3(7)}$) defined in §2.

I General Correctness

Consider any program α viewed as a relation $[\alpha] \in \mathrm{Rel}(U)$. Powering $[\alpha]$ we derive a unary operation $[\alpha]^{\uparrow} : \mathcal{P}(U) \longrightarrow \mathcal{P}(U)$. Then for any $Y \subseteq U$, viewed as a postcondition of α, $[\alpha]^{\uparrow}(Y) = \{x \mid (\exists y)[(x,y) \in [\alpha] \text{ and } y \in Y]\}$ represents the set of all initial states x such that *at least one* execution of α from x is guaranteed to reach a state in Y. The constraint characterising this set of states is an angelic statement (Chapter 1, p 29). Therefore this predicate $[\alpha]^{\uparrow}(Y)$ captures our intuition of a weakest precondition under angelic nondeterminism. $[\alpha]^{\uparrow}$ is normal and completely additive (and hence continuous) and since $[\alpha]$ is an execution relation, by Lemma (2.6) (a) and (b), $[\alpha]^{\uparrow}$ is full and strict. Hence $[\alpha]^{\uparrow}$ satisfies the healthiness properties of the *(angelic) general weakest precondition predicate transformer*, '$gwpa(\alpha, -)$', defined in Jacobs and Gries [1985].

Knowing that the dual $[\alpha]^{\uparrow^*}$ of $[\alpha]^{\uparrow}$ is another unary operation with properties that are dual to those of $[\alpha]^{\uparrow}$ we next enquire whether it corresponds to one defined in Jacobs and Gries [1985]. By definition of a dual, for any $Y \subseteq U$ viewed as a postcondition of α, $[\alpha]^{\uparrow^*}(Y) = \{x \mid (\forall y)[(x,y) \in [\alpha] \Rightarrow y \in Y]\}$ represents the set of all initial states x such that *every* execution of α from x is guaranteed to reach a state in Y. Thus $[\alpha]^{\uparrow^*}$ computes weakest

preconditions under demonic nondeterminism (Chapter 1, p 28). By duality, $[\alpha]^{\uparrow^*}$ is full and completely multiplicative and since $[\alpha]$ is an execution relation $[\alpha]^{\uparrow^*}$ is normal and strict (by Lemma (2.6) (a) and (b)) and continuous (by Lemma (2.7)). Hence $[\alpha]^{\uparrow^*}$ satisfies the healthiness properties of Jacobs and Gries's [1985] *(demonic) general weakest precondition predicate transformer*, '$gwp(\alpha, -)$'.

From this we can conclude that there is a *duality relationship* between weakest preconditions under the two interpretations of nondeterminism. To formulate strongest postcondition predicate transformers first note that the only difference between $[\alpha]^{\uparrow}$ and $[\alpha]^{\smile\uparrow}$ is that the former transforms final states to initial states of α and the latter transforms initial states into final states of α. This means that for any $X \subseteq U$, viewed as a precondition, $[\alpha]^{\smile\uparrow}(X) = \{y \mid (\exists x)[(x,y) \in [\alpha] \text{ and } x \in X]\}$ characterises the smallest set of final states such that some execution of α from a state in X is guaranteed to reach one of them. Thus $[\alpha]^{\smile\uparrow}$ computes strongest postconditions under demonic nondeterminism. $[\alpha]^{\smile\uparrow}$ is normal and completely additive (and hence continuous) and since $[\alpha]$ is an execution relation (by Lemma (2.6) (a)) $[\alpha]^{\smile\uparrow}$ is full and such that $[\alpha]^{\smile\uparrow}(\{\bot\}) = \{\bot\}$ (since $[\alpha]^{\smile\uparrow}(\{\bot\}) = \{y \mid (\exists x)\ [(x,y) \in [\alpha]$ and $x \in \{\bot\}\} = \{y \mid (\exists x)\ [(x,y) \in [\alpha]$ and $x = \bot\} = \{\bot\}$ (by property (ii) of execution relations)). Hence $[\alpha]^{\smile\uparrow}$ is consistent with Jacobs and Gries's [1985] *(demonic) generalised strongest postcondition predicate transformer*, '$gsp(-, \alpha)$'.

By analogy with the dual of $[\alpha]^{\uparrow}$ one might expect for any $X \subseteq U$, viewed as a precondition, $([\alpha]^{\smile\uparrow})^*(X) = \{y \mid (\forall x)[(x,y) \in [\alpha] \Rightarrow x \in X]\}$ to be the strongest postcondition predicate transformer of α under angelic nondeterminism. However, there seems to be some doubt in the literature as to whether or not a program has a *unique* strongest postcondition with respect to a given precondition under angelic nondeterminism. Jacobs and Gries ([1985] p 73) claim that 'under angelic nondeterminism there is no unique strongest (smallest) postcondition for a given precondition, because two "candidates" for the strongest postcondition (in the sense that they cannot be made any stronger) can be completely disjoint.' Unfortunately Jacobs and Gries provide no example of two distinct postconditions. I have not been able to provide such an example either, nor have I been able to disprove their claim. If this claim in Jacobs and Gries [1985] is indeed true then $([\alpha]^{\smile\uparrow})^*$, being a function, is not

the only strongest postcondition predicate transformer of α under angelic nondeterminism. (Note that $([\alpha]^{\smile\uparrow})^*$ corresponds to '$gspa(-,\alpha)$' used in Jacobs and Gries [1985].)

II Total Correctness

In a total correctness model non-termination is never permitted. So only predicates which do not contain $\perp$ can be viewed as postconditions. Hence by the strictness property of $[\alpha]^\uparrow : \mathcal{P}(U) \longrightarrow \mathcal{P}(U)$ for any postcondition Y, $\perp \notin [\alpha]^\uparrow(Y)$. Thus it is reasonable to consider only predicates that do not contain $\perp$. Then for any $Y \subseteq U - \{\perp\}$, viewed as a postcondition of α, $[\alpha]^\uparrow(Y)$ is a predicate which characterises the set of all initial states x such that at least one execution of α from x is guaranteed to *terminate* in a state in Y. Therefore $[\alpha]^\uparrow$, viewed as a function of predicates over $U - \{\perp\}$, computes the weakest preconditions for the total correctness of α under angelic nondeterminism. This predicate transformer satisfies all the properties of $[\alpha]^\uparrow$ which do not refer only to predicates containing $\perp$ and consequently is consistent with the *(angelic) total correctness weakest precondition predicate transformer*, '$wpa(\alpha,-)$' (Jacobs and Gries [1985]). By analogy with the dual of $[\alpha]^\uparrow : \mathcal{P}(U) \longrightarrow \mathcal{P}(U)$, $[\alpha]^{\uparrow^*} : \mathcal{P}(U - \perp) \longrightarrow \mathcal{P}(U - \perp)$ is consistent with the conventional interpretation of the weakest precondition predicate transformer for total correctness, '$wp(\alpha,-)$' (Jacobs and Gries [1985]).

A simple restriction of $[\alpha]^{\smile\uparrow}$ does not lead to the *(demonic) total correctness strongest postcondition predicate transformer.* In the first instance note that the strictness property of $[\alpha]^{\smile\uparrow}$ does not exclude the possibility of $\perp \notin X$ and $\perp \in [\alpha]^{\smile\uparrow}(X)$ for every $X \subseteq U$. However, for any $X \subseteq U - \{\perp\}$, viewed as a precondition of α, such that execution of α from every state in X is guaranteed to terminate, $[\alpha]^{\smile\uparrow}(X)$ is a predicate which characterises the smallest set of final states x such that at least one execution of α from a state in X is guaranteed to terminate in x. Under this condition, $[\alpha]^{\smile\uparrow}(X)$ describes the strongest postcondition of α for total correctness under demonic nondeterminism; otherwise it is undefined.

Secondly, $[\alpha]^{\smile\uparrow}$, viewed as a function on predicates over $U-\{\perp\}$, fails to be monotonic. Take

any $X \subseteq U - \{\bot\}$ such that α is guaranteed to terminate from every state in X. Suppose $x \notin \{\bot\}$ is a state which leads to nontermination. Then $X \subseteq X \bigcup \{x\}$ and $[\alpha]^{\smile\uparrow}(X)$ is defined but $[\alpha]^{\smile\uparrow}(X \bigcup \{x\})$ is undefined. Consequently, this predicate transformer is only normal and in general, fails to be a useful total correctness predicate transformer. However, for any terminating executable relation $[\alpha] \in \mathrm{Rel}(U)$ (that is, $(\bot, \bot) \in [\alpha]$), it follows from Lemma (2.6)(c) that $[\alpha]^{\smile\uparrow}$ is strict. In this special case, $[\alpha]^{\smile\uparrow}$ viewed as a function of predicates over $U - \{\bot\}$ is always defined and satisfies all the properties of $[\alpha]^{\smile\uparrow}$ which do not refer to predicates containing $\bot$. Hence this predicate is useful for the semantic characterisation of terminating programs and is consistent with Jacobs and Gries's [1985] *(demonic) total correctness strongest postcondition predicate transformer*, '$sp(-, \alpha)$'. By duality, for any terminating $[\alpha] \in \mathrm{Rel}(U)$, $([\alpha]^{\smile\uparrow})^*$ computes strongest postconditions of α for total correctness under angelic nondeterminism. But as for general correctness the question of whether or not $([\alpha]^{\smile\uparrow})^* : \mathcal{P}(U - \{\bot\}) \longrightarrow \mathcal{P}(U - \{\bot\})$ is the unique such predicate transformer is left unanswered in this thesis.

III Partial Correctness

Based on the semantic distinctions permitted in a partial correctness model it seems that preconditions for programs always allow nontermination. By the strictness property of $[\alpha]^{\uparrow} : \mathcal{P}(U) \longrightarrow \mathcal{P}(U)$, $\bot \in [\alpha]^{\uparrow}(X \bigcup \{\bot\})$, $\forall X \bigcup \{\bot\} \subseteq U$. So our attention is henceforth restricted to predicates that contain $\bot$. Then for any $X \bigcup \{\bot\} \subseteq U$, $[\alpha]^{\uparrow}(X \bigcup \{\bot\})$ is a predicate characterising the set of all states x such that at least one execution of α from x will either not terminate or terminate in a state in X. This predicate captures our interpretation of the *weakest liberal precondition* under angelic nondeterminism. As can be expected any property of $[\alpha]^{\uparrow} : \mathcal{P}(U) \longrightarrow \mathcal{P}(U)$ which refers to predicates not containing $\bot$ is not satisfied by $[\alpha]^{\uparrow} : \mathcal{P}(U) \bigcup \{\bot\} \longrightarrow \mathcal{P}(U) \bigcup \{\bot\}$. In particular, the normality property (that is, $[\alpha](\emptyset) = \emptyset$) does not hold and the strictness property becomes $\bot \in [\alpha]^{\uparrow}(\{\bot\})$. Hence $[\alpha]^{\uparrow}$, viewed as a function over $\mathcal{P}(U) \bigcup \{\bot\}$, is consistent with Jacobs and Gries's [1985] *(angelic) partial correctness predicate transformer* '$wlpa(\alpha, -)$'. By duality, $\alpha^{\uparrow *} : (\mathcal{P}(U) \bigcup \{\bot\}) \longrightarrow (\mathcal{P}(U) \bigcup \{\bot\})$ is consistent with the *(demonic) partial correctness predicate transformer* '$wlp(\alpha, -)$'.

Unlike the total correctness case, the partial correctness strongest postcondition predicate transformer is a simple restriction of that for general correctness to predicates that contain $\perp$. The strictness property and monotonicity of $[\alpha]^{\smile\uparrow}$ ensures that these restrictions are well-defined (in the sense that for any $X \bigcup \{\perp\} \subseteq U$, $[\alpha]^{\smile\uparrow}(X \bigcup \{\perp\})$ is a predicate containing $\perp$). Therefore $[\alpha]^{\smile\uparrow} : (\mathcal{P}(U) \bigcup \{\perp\}) \longrightarrow (\mathcal{P}(U) \bigcup \{\perp\})$ computes strongest postconditions for partial correctness under demonic nondeterminism and satisfies all the properties of $[\alpha]^{\smile\uparrow} : \mathcal{P}(U) \longrightarrow \mathcal{P}(U)$ except that of normality. This predicate transformer is therefore consistent with Jacobs and Gries's [1985] *(demonic) partial correctness strongest postcondition predicate transformer*, '$slp(-, \alpha)$'. As before the question of whether or not there is a unique strongest postcondition predicate transformer for partial correctness under angelic nondeterminism is left unanswered in this thesis.

Note that we have constructively defined twelve predicate transformers (as power operations) and shown that each satisfies a set of healthiness properties for some notion of correctness. Namely, for any program α,

I General Correctness: For any predicate $X, Y \subseteq U$,

$gwp(\alpha, Y) = [\alpha]^{\uparrow^*}(Y)$ $\quad$ $gspa(X, \alpha) = [\alpha]^{\smile\uparrow^*}(X)$

$gwpa(\alpha, Y) = [\alpha]^{\uparrow}(Y)$ $\quad$ $gsp(X, \alpha) = [\alpha]^{\smile\uparrow}(X)$

II Total Correctness: For any predicates $X, Y \subseteq \mathcal{S}$,

$wp(\alpha, Y) = [\alpha]^{\uparrow^*}(Y)$ $\quad$ $spa(X, \alpha) = [\alpha]^{\smile\uparrow^*}(X)$

$wpa(\alpha, Y) = [\alpha]^{\uparrow}(Y)$ $\quad$ $sp(X, \alpha) = [\alpha]^{\smile\uparrow}(X)$

III Partial Correctness: For any predicates $X, Y \subseteq U$ which contain $\perp$,

$wlp(\alpha, Y) = [\alpha]^{\uparrow^*}(Y)$ $\quad$ $slpa(X, \alpha) = [\alpha]^{\smile\uparrow^*}(X)$

$wlpa(\alpha, Y) = [\alpha]^{\uparrow}(Y)$ $\quad$ $slp(X, \alpha) = [\alpha]^{\smile\uparrow}(X)$

These converse-duality relationships between the general correctness predicate transformers together allow us to define weakest preconditions in terms of strongest postconditions and vice versa. For example, by the duality relationships,

$$gwp(\alpha, Y) = [\alpha]^{\uparrow^*}(Y) = \neg[\alpha]^{\uparrow}(\neg Y) = \neg gwpa(\alpha, \neg Y).$$

$$gspa(X, \alpha) = [\alpha]^{\smile\uparrow^*}(X) = \neg[\alpha]^{\smile\uparrow}(\neg X) = \neg gspa(\neg X, \alpha).$$

$gwp(\alpha, Y) = [\alpha]^{\dagger^*}(Y) = ([\alpha]^\smile)^{\dagger^*}(Y) = \neg gsp(\neg Y, \beta)$ (also $gsp(X, \alpha) = \neg gwp(\beta, \neg X)$).

$gwpa(\alpha, Y) = [\alpha]^\dagger(Y) = ([\alpha]^\smile)^{\smile\dagger}(Y) = \neg gspa(\neg Y, \beta)$

(also $gspa(X, \alpha) = \neg gwpa(\beta, \neg X)$), where the execution relation of β is $[\alpha]^\smile$.

Unfortunately the converse-duality relationships are not so simple under partial- and total correctness because the properties of the corresponding predicate transformers are not as uniform and complete as those for general correctness (due to the restictions made on predicates). In §4 I will show that every predicate transformer associated with a program arises in this way.

The inverse of the power operation allows us to lower each of these predicate transformers to a corresponding relation. By virtue of Lemma (2.6) we can isolate the corresponding set of execution relations. In §4 I will show for a program α how to translate between its relational and predicate transformer representations.

In conclusion I mention how restrictions on the execution relations effect the properties of the predicate transformers.

(i) In our discussion on total correctness we saw that '$sp(-, \alpha)$' can be used only for the semantic characterisation of *terminating* execution relations.

(ii) *Determinism* restricts the final states but not the initial states of programs and hence affects the properties of the weakest preconditions. In particular, for any deterministic relation $[\alpha] \in \mathrm{Rel}(U)$, by Lemma (2.6)(d), for any $Y \subseteq U$, $[\alpha]^\dagger(Y) = [\alpha]^{\dagger^*}(Y)$, which means that under any notion of correctness the two interpretations of nondeterminism are equivalent for deterministic programs.

5.4 Representability

The question of representability (for predicate transformers) arises in several different forms. Guerreiro [1982] translated back and forth between properties of executable relations and properties of predicate transformer representations of programs and provided as solution

to the representation problem the following theorem (formulated using the notation of this chapter).

Theorem (Guerreiro ([1982] p 174))
For any given predicate transformer F there exists an execution relation R such that F is a predicate transformer representation of R iff F satisfies the set of healthiness properties for Dijkstra's weakest precondition predicate transformer and such that $F(U) = F(U - \{\bot\})$ and $F(U) \subseteq U - \{\bot\}$.

Some of the ideas introduced there also appear in Jacobs and Gries [1985]. Jacobs and Gries derived healthiness properties of predicate transformers from those of execution relations for their different notions of correctness and argued (but did not prove) that under a notion of correctness *a binary relation is consistent with the healthiness properties iff it is an execution relation.* The fundamental question (concerning representability) addressed by Holt ([1991] p 9, 19) is : How do relations $R \subseteq U^2$ correspond to unary operations $F : \mathcal{P}(U) \longrightarrow \mathcal{P}(U)$?

Although each of these studies is based on a theory of relations, as yet it has not been recognised that this question is implicitly answered by an application of Jònsson and Tarski's ([1951], [1952]) power construction.

Within the algebraic context of this chapter, the question of representability is:

(i) Can every relational representation of a program α, say $[\alpha]$, be transformed to a mapping, say $[\alpha]^{\uparrow} : \mathcal{P}(U) \longrightarrow \mathcal{P}(U)$ which is a predicate transformer representation of α?

(ii) If (i) holds, does every predicate transformer arise in this way — that is, for every predicate transformer, say $F : \mathcal{P}(U) \longrightarrow \mathcal{P}(U)$, which satisfies the set of healthiness properties for some notion of correctness under an interpretation of nondeterminism is there a relation, say $F^{\downarrow} \subseteq U^2$ which is a relational representation of some program?

One approach to solving the representation problem for predicate transformers arising from programs is to select some properties of predicate transformers associated with programs.

Then one could try to show that all predicate transformer representations of programs arise in this way. I show here, this can indeed be done within our unifying framework. For this we need to adopt a more formal approach.

I will call a Boolean algebra $\mathcal{B}$ endowed with a set $\mathcal{F}$ of normal , completely additive unary operations a *predicate transformer algebra.* (Note that this is simply a Boolean algebra with unary operators, a concept introduced and investigated in Jònsson and Tarski ([1951], [1952]).) A predicate transformer algebra will be called *executable* if each of its operations is a predicate transformer representation of some program, *continuous* and *strict* if each of its operations is respectively continuous and strict under some notion of correctness.

From §1 it is clear that a relational structure $\mathcal{U} = (U, \mathcal{R})$ where $\mathcal{R}$ is a set of binary relations $R \subseteq U^2$, gives rise to a predicate transformer algebra, namely $(\mathcal{P}(U), \mathcal{F})$, where $\mathcal{F} = \{R^\dagger \mid R \in \mathcal{R}\}$. Conversely, let $\mathcal{A} = (\mathcal{B}, \mathcal{F})$ be *any* predicate transformer algebra. Then $\mathcal{A}$ is a Boolean algebra with unary operators. As a Boolean algebra $\mathcal{B}$ is isomorphic to a subalgebra, say $\mathcal{P}(X)$, of some Boolean set algebra. By the isomorphism each normal, completely additive unary operator over $\mathcal{B}$ induces another such operation over $\mathcal{P}(X)$. Theorem (1.1) allows us to regard each such operator as the power operation of some binary relation over X. Formally, this is the representation theorem for Boolean algebras with unary operators.

(1) **Theorem** (Jònsson and Tarski ([1951], [1952]))

Any Boolean algebra with unary operators is isomorphic to a subalgebra of the power algebra of some relational structure.

Hence any strict predicate transformer algebra, being a Boolean algebra with unary operators, is isomorphic to a subalgebra of the power algebra of some relational structure. By virtue of Lemma (2.6) we can isolate this relational structure which I will call an *executable relational structure.* More precisely, an *executable relational structure* is a relational structure $\mathcal{U} = (U, \mathcal{R})$ where $\mathcal{R}$ is a set of strict binary relations describing the behaviour patterns of programs including those with unbounded nondeterminism (that is, binary relations $R \subseteq U^2$ such that $(\forall x \in U)[(\perp, x) \in R \Rightarrow x = \perp]$).

From here the representation result follows easily. It must be shown that this notion of

an executable relational structure translates back and forth into the notion of a predicate transformer algebra via the power construction.

(2) **Theorem**

The power algebra of an executable relational structure is a(n) (executable) predicate transformer algebra. Conversely, any strict predicate transformer algebra is isomorphic to a subalgebra of the power algebra of some executable relational structure.

Proof Let $\mathcal{U} = (U, \mathcal{R})$ be any executable relational structure. Let $\mathcal{P}(\mathcal{U}) = (\mathcal{P}(U), \mathcal{F})$ be its power algebra where $\mathcal{F} = \{R^{\uparrow} \mid R \in \mathcal{R}\} \bigcup \{R^{\smile\uparrow} \mid R \in \mathcal{R}\}$. By definition of powering a relation, each operation in $\mathcal{F}$ is normal and completely additive. Since each $R \in \mathcal{R}$ is a relational representation of a program, each operation in $\mathcal{F}$ arises from a program. Also since each $R \in \mathcal{R}$ is strict, each operation in $\mathcal{F}$ is strict (by Lemma (2.6) (b)). Hence $\mathcal{P}(\mathcal{U})$ is a strict predicate transformer algebra, in fact, an executable predicate transformer algebra.

For the converse, let $\mathcal{A} = (\mathcal{B}, \mathcal{F})$ be any strict predicate transformer algebra. Then $\mathcal{A}$ is isomorphic to a subalgebra, say $\mathcal{P}(X)$, of some Boolean algebra under some isomorphism $h : \mathcal{B} \longrightarrow \mathcal{P}(X)$ and operations of $\mathcal{F}$ are induced over $\mathcal{P}(X)$ by h. Let $\mathcal{R} = \{f^{\downarrow} \subseteq X^2 \mid f \in \mathcal{F}\}$. The strictness property of operations over $\mathcal{P}(X)$ translates (by Lemma (2.6) (b)) into the strictness property of relations over X. Hence $(X, \mathcal{R})$ is an executable relational structure. By construction, $\mathcal{A}$ is isomorphic to the subalgebra of the power algebra of some executable relational structure. □

(It is worth noting (from Hansoul [1983]) that *Stone duality* can be used to establish a full topological duality for the notion of a predicate transformer algebra. However, I will not present this result here.)

This theorem suggests that in the present context, where we have an explicit representation of non-termination, we can capture exactly the predicate transformer representations of programs including those with unbounded nondeterminism if we drop the continuity property from the sets of healthiness properties for each notion of correctness (in Jacobs and Gries [1985]). This is not surprising since by Lemma (2.7) the continuity properties allow unbounded nondeterminism only if infinitely many final states can be produced by a

program.

Without the use of an explicit formalisation of nontermination, Dijkstra and Scholten [1990] proposed a new set of healthiness properties for total correctness predicate transformer representations of programs, including those with unbounded nondeterminism. Recall from Chapter 2.4 their new approach is to postulate for every program α a partial correctness predicate transformer $wlp.\alpha$ and a predicate $wp.\alpha.true$ and then define a weakest precondition predicate transformer $wp.\alpha$ by $wp.\alpha.X = wlp.\alpha.X \bigcap wp.\alpha.true$ for any predicate $X \subseteq U$. The revised set of healthiness conditions are:

(R0) $wlp(\alpha, \bigcap_i X_i) = \bigcap_i wlp(\alpha, X_i)$ for any predicates $X_i \subseteq U$

(R1) $wp(\alpha, \emptyset) = \emptyset$

So the representation problem for total correctness predicate transformers would be completely solved if we could show that the representation result in Theoem (2) is consistent with Dijkstra and Scholten's [1990] new postulation (as given in Chapter 2.4).

In order to do this it must be shown that the power construction covers their new method of postulation. Recall from Chapter 2.4 that (R1) excludes partial programs (that is, programs which may not always start).

Let $\mathrm{Rel}_T(U)$ denote the set of *total executable relations* over U (that is, binary relations $R \subseteq U^2$ satisfying:

(i) dom $R = U$

(ii) $\forall x \in U$, if $(\perp, x) \in R$ then $x = \perp$.

Now the approach in §3 really pays off: for any program α, we can construct both '$wp(\alpha, -)$' and '$wlp(\alpha, -)$' from the total executable relational representations of α and hence define '$wp(\alpha, -)$' in terms of '$wlp(\alpha, -)$' as follows: Take any program α with relational representation $[\alpha] \subseteq \mathrm{Rel}_T(U)$. Recall from §2.III, $[\alpha]^{\uparrow^*} : \mathcal{P}(U) \bigcup \{\perp\} \longrightarrow \mathcal{P}(U) \bigcup \{\perp\}$ computes weakest liberal preconditions of α under demonic nondeterminism and is completely multiplicative. Since α is a total executable relation, by the duals to Lemma (2.6) (a) and

(b), $[\alpha]^{\uparrow^*}$ is normal and strict. Thus $[\alpha]^{\uparrow^*}$, viewed as a function over $\mathcal{P}(U)\bigcup\{\bot\}$, is consistent with '$wlp.\alpha$' as postulated in Dijkstra and Scholten [1990]. From the strictness of $[\alpha]^{\uparrow^*}$, $\bot \notin [\alpha]^{\uparrow^*}(U-\{\bot\})$, so $[\alpha]^{\uparrow^*}(U-\{\bot\})$ is consistent with '$wp.\alpha.true$' in Dijkstra and Scholten [1990]. (It is worth noting that, as Dijkstra and Scholten ([1990] p 132) explain, the complete multiplicativity of F_α is not explicitly required since it is implied by that of $[\alpha]^{\uparrow^*}$.)

Before proving the main result of this chapter I need some more terminology: I will call an operation $F : \mathcal{P}(U)\bigcup\{\bot\} \longrightarrow \mathcal{P}(U-\{\bot\})$ *healthy* predicate transformer if F is completely multiplicative (that is, for any predicates $X_i \subseteq U\bigcup\{\bot\}$, $F(\bigcap_i X_i) = \bigcap_i F(X_i)$) and F is normal (that is, $F(\{\bot\}) = \emptyset$).

(The reason for considering mappings with domain $\mathcal{P}(U)\bigcup\{\bot\}$ and range $\mathcal{P}(U-\{\bot\})$ is that $wlp.\alpha$ deals with predicates which contain $\bot$; while $wp.\alpha$ deals with predicates which do not contain $\bot$. As will become clear in Theorem (3) these restrictions allow for simpler definitions of the unary operations and relations needed in the proof. Without these restrictions more cases would have to be handled.)

(3) **Theorem** *For any total executable relation $R \subseteq U^2$ there is a strict healthy predicate transformer $F : \mathcal{P}(U)\bigcup\{\bot\} \longrightarrow \mathcal{P}(U-\{\bot\})$ such that $R = F^{*\downarrow}$. Conversely, for any strict healthy predicate transformer $F : \mathcal{P}(U)\bigcup\{\bot\} \longrightarrow \mathcal{P}(U-\{\bot\})$ there is a relation $R \subseteq U^2$ such that $F = R^{\uparrow^*}$.*

Proof Let $[\alpha] \subseteq U^2$ be total executable relational representation of some program α. Since $[\alpha] \subseteq \mathrm{Rel}_T(U)$, $[\alpha]^\uparrow$ is a strict, normal, completely additive operation over $\mathcal{P}(U)$ (by Theorem (2)). Since $[\alpha]$ is total, $[\alpha]^\uparrow$ is full (by Lemma (2.6)(a)). Then by duality, $[\alpha]^{\uparrow^*}$ is a strict, full, normal, completely multiplicative operation over $\mathcal{P}(U)$. Now define a unary operation $F_\alpha : \mathcal{P}(U)\bigcup\{\bot\} \longrightarrow \mathcal{P}(U-\{\bot\})$ by

$$\begin{aligned} F_\alpha(X) &= [\alpha]^{\uparrow^*}(X) \bigcap [\alpha]^{\uparrow^*}(U-\{\bot\}) \\ &= [\alpha]^{\uparrow^*}(X-\{\bot\}) \end{aligned}$$

for any $X \subseteq \mathcal{P}(U)\bigcup\{\bot\}$. Then

$F_\alpha(\{\bot\}) = [\alpha]^{\uparrow^*}(\{\bot\}-\{\bot\}) = [\alpha]^{\uparrow^*}(\emptyset) = \emptyset$ (since $[\alpha]^{\uparrow^*}$ is normal over $\mathcal{P}(U)$)

This shows that F_α is normal with respect to $\mathcal{P}(U) - \{\bot\}$. The complete multiplicative F_α over $\mathcal{P}(U) \bigcup \{\bot\}$ follows from that of $[\alpha]^{\uparrow^*}$. Therefore F_α is a strict healthy predicate transformer. Also ${F_\alpha}^{*\downarrow} = ([\alpha]^{\uparrow^*})^{*\downarrow} = [\alpha]^{\uparrow\downarrow} = [\alpha]$ (by Theorem (2.3) and properties of duals). Hence $F_\alpha : \mathcal{P}(U) \bigcup \{\bot\} \longrightarrow \mathcal{P}(U - \{\bot\})$ is consistent with '$wp.\alpha$' in Dijkstra and Scholten [1990].

For the converse, let $F : \mathcal{P}(U) \bigcup \{\bot\} \longrightarrow \mathcal{P}(U - \{\bot\})$ be any strict, normal, completely multiplicative operation. Then by duality F^* is a strict, full and completely additive operation. Now define a relation $R \subseteq U^2$ by

$$(x, y) \in R \text{ iff } x \in F^*(\{y\}) \text{ iff } (x, y) \in F^{*\downarrow}$$

Firstly R is strict because F^* is strict (In particular, $\forall y \in U$ if $(\bot, y) \in R$ then $\bot \in F^*(\{y\})$; hence by strictness of F^*, $\bot \in \{y\}$, so $y = \bot$. Secondly, R is total since F^* is full. (Suppose R is not total. Then there is some $x \in U$ such that $\{y \mid (x, y) \in R\} = \emptyset$ so for some $x \in U$, $\{y \mid x \in F^*(\{y\})\} = \emptyset$. So there is some $x \in U$ such that $\forall y$, $(x, y) \notin R$; hence there is some $x \in U$ such that $\forall y$, $x \notin F^*(\{y\})$. But $\{y\} \subseteq U$ hence $F^*(\{y\}) \subseteq F^*(U)$ (by monotonicity of F^*). Thus for there is some $x \in U$, $x \notin F^*(U) = U$ (since F^* is full) which establishes the required contradiction.) Therefore $R \subseteq U^2$ is a total executable relation. Also $R^{\uparrow^*} = (F^{*\downarrow})^{\uparrow^*} = F$ (by Theorem (2.3) and properties of duals). □.

As we had hoped, Dijkstra and Scholten [1990] new set of healthiness conditions for total correctness are exactly our properties for a strict healthy predicate transformer. This shows that we correctly predicted the constraints on this predicate transformer which was independently postulated and hence solved the open problem set in Holt ([1990] p 30).

In conclusion: my aim in this chapter has been to stimulate interest in the algebraic treatment of predicate transformers, not to attempt a detailed exposition. Therefore there are several unresolved questions which bear further investigation. I briefly mention two.

As I mentioned in the introduction to this chapter relations between predicates can also be

used to formalise the behaviour of programs. This idea is due to Hoare [1969] (although not previously recognised as such). By analogy with predicate transformers relations between predicates could be called *predicate relators*.

First, can a uniform framework for predicate relators (analogous to that for predicate transformers) be developed to study notions of correctness? The aim would be to find some way of lifting a relation between states to a *relation* between predicates, instead of to an *operation* over predicates. Such a construction (called *power structures*) has been investigated in Brink [19?] This involves defining for any relation R over a set U, a corresponding power relation R^+ over its power set.

Second, is there a way of translating for a program α between its predicate relator and its predicate transformer representations? In this connection, if a predicate relator can be modelled as a power relation, then a characterisation, such as given in Brink [19?], of the power relation in term of power operations may be required. That is, for any relation $R \subseteq U^2$ and any sets $X, Y \subseteq U$,

$$XR^+Y \text{ iff } X \subseteq R^{\uparrow}(Y) \text{ and } Y \subseteq R^{\smile\uparrow}.$$

Chapter 6

The Flowsets Model

In earlier chapters the idea of viewing a program as a sequence of atomic steps has occurred several times. In Chapter 1 it was used to describe execution methods, in Chapter 2 it was used to categorise the state space and in Chapter 3 it was used to discuss notions of correctness. My aim in this chapter is to use this approach to model the algebra of weakest preconditions. For completeness I include the definitions of all the relevant notation and terminology used in the earlier chapters and a page reference to their first use.

Recall from Chapter 1 (p 24) that an execution of a program yields an *execution sequence* (or *exseq*) of states; because programs are nondeterministic any initial state s gives rise to an *execution tree* (or *extree*), and the meaning of a program is given by the set of all its possible execution sequences. Such a set will in this chapter be called a *flowset* (by analogy with *flow diagram*). In §1 I develop a calculus of flowsets. This is used in §2 to verify all the Dijkstra/Gries conditions (given in Chapter 2.2) or in some cases small variations thereof. Thus the calculus of flowsets models the algebra of weakest preconditions. In §3 I briefly consider invariants (in an algebraic setting) in order to prove what Dijkstra [1976] calls 'The Fundamental Invariance Theorem for Loops'.

6.1 The Calculus of Flowsets

Let $\mathcal{S}$ be the state space. In this chapter I distinguish between cleanly terminating-, messily terminating and nonterminating executions of a program from a state and represent all executions of a program. (That is, I choose representation method $\mathbf{R_3(8)}$ defined in Chapter 1 **Table 3** (p 13).)

Recall from Chapter 3.5 that when these distinctions were made we defined (p 93) the set $\text{Seq}(\mathcal{S})$ of exseqs as follows:

(1) $\mathcal{S}^+$ denotes the set of all finite non-empty sequences of elements of $\mathcal{S}$.

$\mathcal{S}^\perp$ denotes the set of all finite nonempty sequences with $\perp$ as the last component and elements of $\mathcal{S}$ as the first and (if any) intermediate components.

$\mathcal{S}^\infty$ denotes the set of all infinite sequences of elements of $\mathcal{S}$.

$\text{Seq}(\mathcal{S}) = \mathcal{S}^+ \bigcup \mathcal{S}^\perp \bigcup \mathcal{S}^\infty$.

The sets $\mathcal{S}^+, \mathcal{S}^\perp$ and $\mathcal{S}^\infty$ are disjoint.

Recall the notation for exseq introduced in Chapter 2.5 (p 58): exseqs are denoted by $\mathbf{x}$, $\mathbf{y}$, $\mathbf{z}$, ... etc, and '$\mathbf{x} = (x_1, x_2, x_3, \ldots)$' denotes either a finite or an infinite exseq; while '$\mathbf{x} = (x_1, x_2, \ldots, x_n)$' denotes only a finite exseq with x_n as last element, where either $x_n \in \mathcal{S}$ (if $\mathbf{x} \in \mathcal{S}^+$) or $x_n = \perp$ (if $\mathbf{x} \in \mathcal{S}^\perp$).

I now introduce a little *calculus of exseqs.* Recall from Chapter 2.5 (p 58) two operations *first* and *last* were defined on exseqs. I now include a third operation *length* on exseqs. For any $\mathbf{x} \in \text{Seq}(\mathcal{S})$, say $\mathbf{x} = (x_1, x_2, x_3, \ldots)$:

(2) $first(\mathbf{x}) = x_1$

$$length(\mathbf{x}) = \begin{cases} n & \text{if } \mathbf{x} \in \mathcal{S}^+ \bigcup \mathcal{S}^\perp \text{ and } \mathbf{x} = (x_1, x_2, \ldots, x_n) \\ \infty & \text{if } \mathbf{x} \in \mathcal{S}^\infty \end{cases}$$

$$last(\mathbf{x}) = \begin{cases} x_n & \text{if } \mathbf{x} \in \mathcal{S}^+ \bigcup \mathcal{S}^\perp \text{ and } \mathbf{x} = (x_1, x_2, \ldots, x_n) \\ \text{undefined} & \text{if } \mathbf{x} \in \mathcal{S}^\infty \ . \end{cases}$$

I also need to be able to *compose* two exseqs $\mathbf{x}$ and $\mathbf{y}$. In the paradigm case this takes

place when $\mathbf{x}$ terminates in exactly that state in which $\mathbf{y}$ begins: then $\mathbf{x} \circ \mathbf{y}$ is obtained by identifying $\mathbf{x}$'s last component with $\mathbf{y}$'s first component, thus joining the two exseqs together. For other cases special provision must be made. As follows:

(3) For any $\mathbf{x}, \mathbf{y} \in \mathrm{Seq}(\mathcal{S})$,

$$\mathbf{x} \circ \mathbf{y} = \begin{cases} \mathbf{x} & \text{if } \mathbf{x} \in \mathcal{S}^{\perp} \bigcup \mathcal{S}^{\infty} \\ (x_1, x_2, \ldots, x_n, y_2, y_3, \ldots) & \text{if } \mathbf{x} \in \mathcal{S}^{+}, \text{ say } \mathbf{x} = (x_1, x_2, \ldots, x_n) \text{ and} \\ & \mathbf{y} = (y_1, y_2, y_3, \ldots), \text{ and } last(\mathbf{x}) = first(\mathbf{y}) \\ \text{undefined} & \text{otherwise .} \end{cases}$$

The idea is that if an execution terminates messily, or does not terminate, then formally sequencing another execution after it has no effect. Sequencing is only effective when the second execution picks up where the first left off.

As a final operation on exseqs I make explicit the usual *prefix ordering* of sequences — also called 'ordering by initial subsequences'. For any $\mathbf{x}, \mathbf{y} \in \mathrm{Seq}(\mathcal{S})$, we have (with $n < \infty$ for every $n \in \mathcal{N}$):

(4) $\mathbf{x} \leq \mathbf{y}$ iff (i) $length(\mathbf{x}) \leq length(\mathbf{y})$, and
(ii) $x_i = y_i$ for every i such that $0 \leq i \leq length(\mathbf{x})$.

I can now explore some mathematical properties of this little calculus of exseqs. To begin with, $\circ$ is associative and $\leq$ is a partial order. Thus:

(5) **Lemma** $(Seq(\mathcal{S}), \circ, \leq)$ *is a partially ordered semigroup.*

Proof We need to show that $\circ$ is associative and that $\leq$ is a partial order (ie. reflexive, anti-symmetric and transitive).

First, to show that $\circ$ is associative (i.e. for any $\mathbf{x}, \mathbf{y}, \mathbf{z} \in Seq(\mathcal{S})$ $\mathbf{x} \circ (\mathbf{y} \circ \mathbf{z}) = (\mathbf{x} \circ \mathbf{y}) \circ \mathbf{z}$) consider the following possible cases:

(i) Let $\mathbf{x} \in \mathcal{S}^{\perp} \bigcup \mathcal{S}^{\infty}$ arbitrarily, then for any exseqs $\mathbf{y}$, $\mathbf{z}$, $\mathbf{x} \circ (\mathbf{y} \circ \mathbf{z}) = \mathbf{x}$ and $(\mathbf{x} \circ \mathbf{y}) \circ \mathbf{z} = \mathbf{x} \circ \mathbf{z} = \mathbf{x}$.

(ii) Let $\mathbf{x} \in \mathcal{S}^{+}$ arbitrarily, then for any exseq $\mathbf{y}$ with $last(\mathbf{x}) = first(\mathbf{y})$, there are three possible cases:

(a) if $\mathbf{y} \in \mathcal{S}^{\perp} \bigcup \mathcal{S}^{\infty}$ then for any exseq $\mathbf{z}$, $\mathbf{x} \circ (\mathbf{y} \circ \mathbf{z}) = \mathbf{x} \circ \mathbf{y}$ and

$(\mathbf{x} \circ \mathbf{y}) \circ \mathbf{z} = \mathbf{x} \circ \mathbf{y}$ since $\mathbf{x} \circ \mathbf{y} \in \mathcal{S}^{\perp} \bigcup \mathcal{S}^{\infty}$

(b) if $\mathbf{y} \in \mathcal{S}^{+}$ with $last(\mathbf{y}) = first(\mathbf{z})$ for any exseq $\mathbf{z}$, say

$\mathbf{x} = (x_1, x_2, \ldots, x_n)$, $\mathbf{y} = (x_n, y_2, \ldots, y_m)$, $\mathbf{z} = (y_m, z_2, z_3 \ldots)$ then

$\mathbf{x} \circ (\mathbf{y} \circ \mathbf{z}) = \mathbf{x} \circ (x_n, y_2, \ldots, y_m, z_2, z_3, \ldots) = (x_1, \ldots, x_n, y_2, \ldots, y_m, z_2, z_3, \ldots)$

and $(\mathbf{x} \circ \mathbf{y}) \circ \mathbf{z} = (x_1, \ldots, x_n, y_2, \ldots, y_m) \circ \mathbf{z} = (x_1, \ldots, x_n, y_2, \ldots, y_m, z_2, z_3, \ldots)$

(c) if $\mathbf{y} \in \mathcal{S}^{+}$ with $last(\mathbf{y}) \neq first(\mathbf{z})$ for any $\mathbf{z} \in Seq(\mathcal{S})$, then $\mathbf{x} \circ (\mathbf{y} \circ \mathbf{z})$ is undefined since $\mathbf{y} \circ \mathbf{z}$ is undefined and $(\mathbf{x} \circ \mathbf{y}) \circ \mathbf{z}$ is undefined since $\mathbf{x} \circ \mathbf{y} \in \mathcal{S}^{+}$ but $last(\mathbf{x} \circ \mathbf{y}) = last(\mathbf{y}) \neq first(\mathbf{z})$.

(iii) If $\mathbf{x} \in \mathcal{S}^{+}$ with $last(\mathbf{x}) \neq first(\mathbf{y})$ for every exseq $\mathbf{y}$ then for any exseq $\mathbf{z}$, $\mathbf{x} \circ (\mathbf{y} \circ \mathbf{z})$ is undefined since $\mathbf{y} \circ \mathbf{z}$ is either undefined or an exseq with $first(\mathbf{y} \circ \mathbf{z}) = first(\mathbf{y}) \neq last(\mathbf{x})$ and $(\mathbf{x} \circ \mathbf{y}) \circ \mathbf{z}$ is undefined since $\mathbf{x} \circ \mathbf{y}$ is undefined.

Second, $\leq$ is a partial order. For any $\mathbf{x}$, $\mathbf{y}$, $\mathbf{z} \in Seq(\mathcal{S})$,

(i) $\mathbf{x} \leq \mathbf{x}$ trivially.

(ii) If $\mathbf{x} \leq \mathbf{y}$ and $\mathbf{y} \leq \mathbf{x}$ then

$length(\mathbf{x}) \leq length(\mathbf{y})$ and $length(\mathbf{y}) \leq length(\mathbf{x})$; so $length(\mathbf{x}) = length(\mathbf{y})$

and $\forall i : 0 \leq i \leq length(\mathbf{x}) = length(\mathbf{y})$, $x_i = y_i$.

Thus $\mathbf{x} = \mathbf{y}$.

(iii) If $\mathbf{x} \leq \mathbf{y}$ and $\mathbf{y} \leq \mathbf{z}$ then

$length(\mathbf{x}) \leq length(\mathbf{y})$ and $length(\mathbf{y}) \leq length(\mathbf{z})$; so $length(\mathbf{x}) \leq length(\mathbf{z})$

Also $\forall i : 0 \leq i \leq length(\mathbf{x})$, $x_i = y_i$ and $\forall j : 0 \leq j \leq length(\mathbf{y})$, $y_j = z_j$;

so $\forall i : 0 \leq i \leq length(\mathbf{x})$, $x_i = y_i = z_i$.

Thus $\mathbf{x} \leq \mathbf{z}$. □

As is often the case in dealing with partially ordered structures (for example, in denotational semantics), it will be important that chains have least upper bounds. Recall that a *chain* C in a partially ordered set $(X, \leq)$ is a linearly ordered subset of X — that is, one such that for any $x, y \in C$ either $x \leq y$ or $y \leq x$. I denote least upper bounds by the neutral notation 'lub'.

(6) **Lemma** *Every chain in Seq($\mathcal{S}$) has a least upper bound.*

Proof Let $\{\mathbf{x}_i\}_{i\in I}$ be a chain of exseqs (under the prefix ordering). Define $\mathbf{x}$ to be that exseq such that (i) $length(\mathbf{x}) = \text{lub}\{length(\mathbf{y}) | \ \mathbf{y} \in \{\mathbf{x}_i\}_{i\in I}\}$ and (ii) $\forall \mathbf{y} \in \{\mathbf{x}_i\}_{i\in I}$ and $\forall j : 0 \leq j \leq length(\mathbf{y})$ we have $y_j = x_j$. Then $\mathbf{x}$ is that exseq of which all the $\mathbf{x}_i$'s are prefixes; hence $\mathbf{x}$ is an upper bound for $\{\mathbf{x}_i\}_{i\in I}$. Any other upper bound $\mathbf{z}$ for $\{\mathbf{x}_i\}_{i\in I}$ is prefixed by each $\mathbf{x}_i$. So $\forall \mathbf{y} \in \{\mathbf{x}_i\}_{i\in I}$, $length(\mathbf{y}) \leq length(\mathbf{z})$; hence $length(\mathbf{x}) \leq length(\mathbf{z})$. Also, $\forall \mathbf{y} \in \{\mathbf{x}_i\}_{i\in I}$, $\forall j : 0 \leq j \leq length(\mathbf{y})$ $y_j = z_j$; hence $\forall j : 0 \leq j \leq length(\mathbf{x})$ $x_j = z_j$. Thus $\mathbf{x}$ prefixes $\mathbf{z}$; so $\mathbf{x} = \text{lub } \{\mathbf{x}_i\}_{i\in I}$. □

Exseqs will model executions, but my aim is more ambitious than that: I wish to model the programs from which these executions arise. I do so by using the *power construction* expounded (for example) in Brink [19?], Goldblatt [1989] and Grätzer and Whitney [1984]. Recall that in Chapter 5 I used the notion of a power construction as initiated in Jónsson and Tarski ([1951], [1952]) to form from a binary relation $R \subseteq \mathcal{S}$ its power operation $R^\dagger : \mathcal{P}(\mathcal{S}) \longrightarrow \mathcal{P}(\mathcal{S})$. Here I use a power construction to form from the partially ordered semigroup $(Seq(\mathcal{S}), \circ, \leq)$ its *power structure* $(\mathcal{P}(\text{Seq}(\mathcal{S})), \circ, \Leftarrow, E)$, where:

(7) $\mathcal{P}(\text{Seq}(\mathcal{S}))$ is the set of all subsets $X, Y, Z, \ldots$ of $\text{Seq}(\mathcal{S})$;

$X \circ Y = \{\mathbf{x} \circ \mathbf{y} \ | \mathbf{x} \in X \text{ and } \mathbf{y} \in Y\} \quad \forall X, Y \in \mathcal{P}(\text{Seq}(\mathcal{S}))$;

$X \Leftarrow Y$ iff $(\forall \mathbf{x} \in X)(\exists \mathbf{y} \in Y)[\mathbf{x} \leq \mathbf{y}]$ and $(\forall \mathbf{y} \in Y)(\exists \mathbf{x} \in X)[\mathbf{x} \leq \mathbf{y}]$,

$\forall X, Y \in \mathcal{P}(\text{Seq}(\mathcal{S}))$;

$E = \{(s) | s \in S\}$ (i.e. the set of one-component sequences).

(Note that $\Leftarrow$ is the Egli-Milner ordering of powerdomain theory in denotational semantics provided by Egli [1975] and Plotkin [1975] (who attributes it to Milner [1973]).)

The required model of programs is obtained as a substructure of this power structure, namely that consisting of certain special sets of exseqs.

(8) A set $X \in \mathcal{P}(\text{Seq}(\mathcal{S}))$of exseqs is called a *flowset* if it satisfies

Axiom 1: $\forall s \in \mathcal{S}$ $\exists \mathbf{x} \in X$ with $first(\mathbf{x}) = s$, and

Axiom 2: $\forall$ x, y $\in$ X, x $\not<$ y.

The set of all flowsets will be denoted by '$\mathcal{F}$'.

The idea with **Axiom 1** is that any state s is a possible initial state of any program. (This means we assume programs have initialisation property A(3) in Chapter 1.) An execution may not actually progress from s (in which case it is modelled by the exseq (s)), or it may immediately terminate (modelled by the exseq $(s, \perp)$) — but at any rate it is *defined*. One virtue of this idea is that, for any state s, any set of exseqs $\mathbf{x}$ with $first(\mathbf{x}) = s$ provides a natural model for the *execution tree* (denoted as $\mathbf{extree}(\alpha, s)$ in Chapter 1, p 25) of some program α from this initial state s. Note that it is risky to rely on a graphical presentation of such trees, or a definition intended to capture such a graphical presentation. For example, a program α may, from some given initial state s, have the possible execution sequences (s, t_1, u, v_1) and (s, t_2, u, v_2), and only these. Yet the graphical tree representation of **Figure 7** would seem to indicate that (s, t_1, u, v_2) and (s, t_2, u, v_1) are also possible execution sequences. But this is not intended.

The idea with **Axiom 2** is that no initial subsequence of an execution sequence is also an execution sequence. This, in fact, is the manifestation of a rather subtle point concerning nondeterminism.

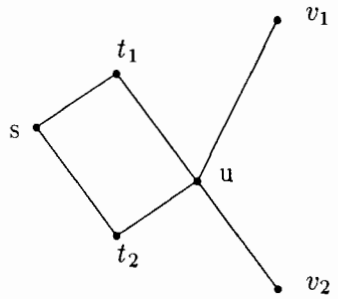


Figure 7

Generally speaking, a program is said to be nondeterministic if from any given state there is no assumption of a *unique* next state. The question raised here, which does not seem to have been addressed before, is whether or not one should uniformly assume the *existence* of a next state. In short: does the notion of nondeterminism allow a program which has already proceeded up to some state s to sometimes terminate at s and sometimes not? In this chapter, I resolve the matter by simply choosing one of the alternatives: **Axiom 2** rules out a notion of nondeterminism on which initial subsequences of executions can also be executions. Note that any extree satisfies **Axiom 2** (but not necessarily **Axiom 1**).

The operation $\circ$ between flowsets is intended to model sequential composition. Think of X and Y respectively as the set of all possible executions of programs α and β. Then $X \circ Y$ corresponds to the set of all possible executions of the program which consists of doing β immediately after α. Technically: for any sequence $\mathbf{x} \in X$, if it does not terminate cleanly leave it; if it does then append at its final state (say) x_n all exseqs in Y starting with x_n. The relation $\leftleftharpoons$ between flowsets is the power order of the prefix relation between exseqs. It says that $X \leftleftharpoons Y$ iff any exseq in X is a prefix of some exseq in Y, and any exseq in Y has some exseq in X as prefix. Thus, intuitively, Y *extends* X. The set E is useful for technical reasons: it will model a program called *null*, which from any initial state s does exactly nothing.

I now come to the calculus of flowsets.

(9) **Theorem** $(\mathcal{F}, \circ, E, \leftleftharpoons)$ *is a partially ordered monoid, with* E *as identity for* $\circ$*, and as minimum under* $\leftleftharpoons$*.*

Proof To establish the monoid part of the Theorem we need to check that the power operation $\circ$ is associative, and that E is a (left- and right-) identity for $\circ$. The former follows since the power operation of any associative operation is again associative. The latter is also easy, but it is not immediate. That $X \circ E = X$ and $E \circ X = X$ for any flowset X must be checked for all possible cases. To establish the former, take $\mathbf{z} \in X \circ E$ arbitrarily. Then $\mathbf{z} = \mathbf{x} \circ (s)$ for some $\mathbf{x} \in X$ and some $s \in \mathcal{S}$ (i.e. $\mathbf{z}$ is defined). So if either $\mathbf{x} \in \mathcal{S}^+$ with $last(\mathbf{x}) = s$ or $\mathbf{x} \in \mathcal{S}^\perp \bigcup \mathcal{S}^\infty$, $\mathbf{z} = \mathbf{x}$; hence $\mathbf{z} \in X$ and thus $X \circ E \subseteq X$. Also for any $\mathbf{x} \in X$ either $\mathbf{x} \in \mathcal{S}^\perp \bigcup \mathcal{S}^\infty$ in which case $\mathbf{x} \circ (s) = \mathbf{x}$ for any $s \in \mathcal{S}$ or $\mathbf{x} \in \mathcal{S}^+$ with $last(\mathbf{x}) = s$ for some $s \in \mathcal{S}$ in which case $\mathbf{x} \circ (s) = \mathbf{x}$. Hence $\mathbf{x} \in X \circ E$ and thus $X \subseteq X \circ E$.

For the latter, take $\mathbf{z} \in E \circ X$ then $\mathbf{z} = (s) \circ \mathbf{y}$ for some $s \in \mathcal{S}$ and some $\mathbf{y} \in X$, namely $\mathbf{y}$ with $first(\mathbf{y}) = s$ (by **Axiom 1**); hence $\mathbf{z} \in X$ and thus $E \circ X \subseteq X$. Also for any $\mathbf{z} \in X$ $first(\mathbf{z}) \in \mathcal{S}$, say $first(\mathbf{z}) = s$; hence $\mathbf{z} = (s) \circ \mathbf{z}$ with $(s) \in E$ and $\mathbf{z} \in X$; hence $\mathbf{z} \in E \circ X$ and thus $X \subseteq E \circ X$.

Secondly we must establish that $\leftleftharpoons$ is a partial order, and E its minimum. It is known

that the power order of any partial order is a quasi-order — that is, reflexive and transitive. So we only need to check that $\Leftarrow$ is anti-symmetric. To do so, let X and Y be flowsets such that $X \Leftarrow Y$ and $Y \Leftarrow X$; we need to show that $X = Y$. For this we show that $X \subseteq Y$, analogous reasoning would then also establish that $Y \subseteq X$. So let $\mathbf{x} \in X$ arbitrarily. Since $X \Leftarrow Y$ there is some $\mathbf{y} \in Y$ such that $\mathbf{x} \leq \mathbf{y}$. But also, since $Y \Leftarrow X$, there is then some $\mathbf{z} \in X$ such that $\mathbf{y} \leq \mathbf{z}$. But then $\mathbf{x} \leq \mathbf{z}$, hence by **Axiom 2** $\mathbf{x} = \mathbf{z}$, hence since $\mathbf{x} \leq \mathbf{y} \leq \mathbf{z}$ we have $\mathbf{x} = \mathbf{y}$, hence $\mathbf{x} \in Y$. Thus $X \subseteq Y$, as required. To see that E is the minimum element of $\Leftarrow$ we just need to invoke **Axiom 1**.

Finally, we must show that for any flowset $X \subseteq Seq(\mathcal{S})$, $E \Leftarrow X$. First, any exseq $\mathbf{x} \in E$ has form $\mathbf{x} = (s)$ for some $s \in \mathcal{S}$; but by **Axiom 1**, $\exists \mathbf{y} \in X$ with $first(\mathbf{y}) = s$; so $\mathbf{x} \leq \mathbf{y}$. Second, for any $\mathbf{y} \in X$ $\exists \mathbf{x} \in E$ namely $\mathbf{x} = (first(\mathbf{y}))$ such that $\mathbf{x} \leq \mathbf{y}$. □

The next question to address is whether chains in $\mathcal{F}$ have least upper bounds. Recall from Chapter 4.1 (p 111) that the least upper bound for a chain (under the inclusion ordering) of relations is simply the union of the relations. Note that the 'obvious' response that flowsets (like relations) are sets and that therefore lub's should be unions is fallacious. The union of flowsets is indeed a set, but it is not necessarily a flowset: **Axiom 2** is easily violated. However, our earlier preparation in Lemma (6) pays off: chains of flowsets will have lub's because chains of exseqs have them.

(10) **Theorem** *Let* $\{A_i\}_{i \in I}$ *be any chain of flowsets under the ordering* $\Leftarrow$. *Let*

$$A = \{\mathbf{x} \mid \mathbf{x} = \textit{lub under } \leq \textit{ of some chain } C \textit{ in } \bigcup_{i \in I} A_i\}.$$

Then $A = lub\ \{A_i\}_{i \in I}$ *under* $\Leftarrow$.

Proof To establish that A is an upper bound of $\{A_i\}_{i \in I}$ let $i \in I$ arbitrarily. Then any exseq $\mathbf{x}$ in A_i prefixes the lub of some chain C in $\bigcup_{i \in I} A_i$ to which $\mathbf{x}$ belongs. Also any $\mathbf{y} \in A$ is prefixed by each exseq in the chain of which it is the lub; hence in particular is prefixed by an exseq in A_i. Thus $A_i \Leftarrow A$, as required.

To establish that A is the lub let B be any other upper bound of $\{A_i\}_{i \in I}$ (i.e. $A_i \Leftarrow B\ \forall i \in I$). Take $\mathbf{x} \in A$ arbitrarily. Let C be the chain of which $\mathbf{x}$ is the lub. Then since B is an upper bound, there is some element of B prefixed by every element of C, hence by

$\mathbf{x}$. Also for any $\mathbf{y} \in B$ there is a chain C of exseqs, one from each A_i, each of which prefixes $\mathbf{y}$. Then *lub* C is in A and prefixes $\mathbf{y}$. □

In §2 I will view the iterative command DO as a repetition of IF commands. I therefore need to build into the calculus of flowsets the notion of repetition; it is precisely for this purpose that I introduced notions of chains and lub's of flowsets. I define *iterated composition* as in Chapter 4.1 (p 111):

(11) For any flowset X: $X^0 = E$

$$X^{n+1} = X^n \circ X, \quad \forall n \geq 0.$$

Then:

(12) **Theorem** *For any flowset* X, $\{X^n\}_{n\geq 0}$ *forms a chain under the ordering* $\leqq$.

Proof We must show that $X^0 \leqq X^1 \leqq X^2 \leqq \ldots$. By theorem (9), $E \leqq X$ for any flowset X, so we only need to show that $\forall n \geq 0$, $X^n \leqq X^{n+1} = X^n \circ X$. First let $\mathbf{x} \in X^n$; then either $\mathbf{x} \in \mathcal{S}^\perp \bigcup \mathcal{S}^\infty$ or $\mathbf{x} \in \mathcal{S}^+$. If $\mathbf{x} \in \mathcal{S}^\perp \bigcup \mathcal{S}^\infty$ then $\mathbf{x} \in X^n \circ X = X^{n+1}$; hence $\exists \mathbf{z} \in X^{n+1}$, namely $\mathbf{z} = \mathbf{x}$ such that $\mathbf{x} \leq \mathbf{z}$. If $\mathbf{x} \in \mathcal{S}^+$ then $last(\mathbf{x}) \in \mathcal{S}$ so by **Axiom 1**, $\exists \mathbf{y} \in X$ with $first(\mathbf{y}) = last(\mathbf{x})$. Thus $\mathbf{x} \circ \mathbf{y} \in X^{n+1}$ with $\mathbf{x} \leq \mathbf{x} \circ \mathbf{y}$.
Second, let $\mathbf{z} \in X^{n+1}$; then by definition of composition, $\mathbf{z} = \mathbf{x} \circ \mathbf{y}$ for some $\mathbf{x} \in X^n$ and some $\mathbf{y} \in X$; hence $\mathbf{x}$ is an exseq in X^n which prefixes $\mathbf{z}$ as required. □

It now follows from Theorem (10) that for any flowset X the chain $\{X^n\}_{n\geq 0}$ must have a lub. This is an important notion, for which I reserve both a notation and a name.

(13) For any flowset X, the *iteration* X^* of X is defined by $X^* = lub\{X^n\}_{n\geq 0}$.

This *iteration* operation corresponds to the reflexive transitive closure of a relation defined in Chapter 4.1 (p 111).

Finally we introduce into the calculus of flowsets an operator which does not explicitly feature in the Dijkstra/Gries algebra of weakest preconditions, but which is quite useful as an aid in modelling such operations. It is the *nondeterministic choice operator* which for programs α and β would correspond to a program $\alpha \vee \beta$, interpreted as: 'Nondeterministically choose

either α or β and then run the chosen program'. In the relational semantics discussed in Chapter 4.1 nondeterministic choice is modelled by set-theoretic union. But in the calculus of flowsets this won't do, since the union of two flowsets need not be a flowset. Instead we choose to model $\alpha \vee \beta$ by what is known as *Hilbert's epsilon operator*. (This operator originates from Church's [1940] ι-operator.)

This operator, ϵ, features strongly in Higher-Order Logic — see for example Andrews [1986], Gordon [1989a], and Gordon [1989b]. It is a variable-binding operator, like quantifiers, and can be used as a primitive logical symbol. For the purpose of this thesis, however, it will suffice to make clear its semantics. Namely, the ϵ-operator acts as a *choice function*: given any set A, ϵ picks out some unknown but fixed element of A, which is then denoted by $\epsilon.A$. In particular,

(14) For any indexed set $\{X_i\}_{i \in I}$ of flowsets

$$\epsilon.\{X_i\}_{i \in I}$$

denotes some particular unspecified but fixed X_i, $i \in I$.

Note that if the indexed set I is finite then $\epsilon.\{X_1, X_2, \ldots, X_n\}$ is some particular one of n flowsets. But we make no finiteness constraints on I in (14), in order to cater for the unbounded nondeterminism of Dijkstra and Scholten [1990].

In conclusion I point out some related work. Blikle [1987] also models programs as sets of computations and presents an algebra of such sets. But Blikle adopts a notion of 'generalised composition', whereas my approach uses the power construction. Another relevant paper is Hoare [1978], which models programs as sets of 'possible traces', along the lines of operational semantics. It is interesting that Hoare ([1978] p 425) proves exactly what I called **Axiom 1** and **Axiom 2** for flowsets.

6.2 Verifying the Gries/Dijkstra Conditions

My aim in this section should be clear, and is easy to state. I model each of the programs in the Dijkstra/Gries language by a flowset, each operation on programs by an operation

on flowsets, and I try to prove the given formulae of the algebra of weakest preconditions in the calculus of flowsets. I also add a few extra features to the Dijkstra/Gries algebra, as an aid to the exposition.

I adopt the square bracket notation '$[\![\cdot]\!]$' of denotational semantics to map each program α onto its *meaning* $[\![\alpha]\!]$, which will be a flowset. I may also specialise this to 'the meaning of a program α at some initial state s', written $[\![\alpha]\!](s)$, which is an *extree* (or execution tree) in the sense of Chapter 1 (p 25, where it is denoted by $\mathbf{extree}(\alpha, s)$): the set of all those exseqs $\mathbf{x}$ in $[\![\alpha]\!]$ such that first($\mathbf{x}$)= s. In Chapter 1 (p 14) the representation with respect to a representation method $\mathbf{R_j(n)}$ (in **Table 3**, p 13, where $\mathbf{j} = 1, 2, 3$; $\mathbf{n} = 1, 2, \ldots, 8$) of the execution of a program α from a state s is denoted by $exrep_{\mathbf{R_j(n)}}(\alpha, s)$. Recalling from §1 that we have chosen representation method $\mathbf{R_3(8)}$ in this chapter, we see that $[\![\alpha]\!](s)$ corresponds to $exrep_{\mathbf{R_3(8)}}(\alpha, s)$ and $[\![\alpha]\!]$ corresponds to $\bigcup_{s \in \mathcal{S}} exrep_{\mathbf{R_3(8)}}(\alpha, s)$.

To begin with:

(1) $[\![skip]\!] = \{(s, s) | s \in \mathcal{S}\}$

$[\![abort]\!] = \{(s, \perp) | s \in \mathcal{S}\}$

$[\![havoc]\!] = \{(s, t) | s \in \mathcal{S} \text{ and } t \in \mathcal{S}\}$

$[\![null]\!] = \{(s) | s \in \mathcal{S}\} = E$

Recall from Chapter 4.6 (p 141, 142) that the execution relations for *skip* and *abort* are exactly the flowsets given in (1). I introduce the atomic program *null* because it will be useful in defining IF. (This program was not used in Chapter 4 because there is no way of representing 'a state related to no state' as an input-output pair.) By analogy with the relational model, I model sequential composition of programs by composition of flowsets. That is:

(2) For any programs α and β, $[\![\alpha; \beta]\!] = [\![\alpha]\!] \circ [\![\beta]\!]$.

The assignment statement, as was mentioned in Chapter 4.1, is problematic in the Dijkstra/Gries algebra insofar as it is the only command which deals directly with program variables. But the problems raised by assignment are extraneous to the modelling proposed here, and does not affect the issues I discuss. I therefore simply adopt the Dijkstra/Gries

technique (also used for the relational model in Chapter 4.1) of indicating notationally a change in state effected by a change in the value of a program variable. Namely:

(3) For any state $s \in \mathcal{S}$, '$s[e/z]$' will denote that state which differs from s only in that the value of the program variable z is replaced by the value of the expression e evaluated in s.

It is then easy to model the assignment statement as a flowset. (Note: As in Gries [1981] and Chapter 4.1 (p 110), I assume e to be well-defined in every s.)

(4) For any program variable z and expression e,

$$[\![z := e]\!] = \{(s, s[e/z]) | s \in \mathcal{S}\}$$

To model IF I use both the Hilbert epsilon operator and the null command. I first define nondeterministic choice.

(5) For any programs α and β, $[\![\alpha \vee \beta]\!] = \epsilon.\{[\![\alpha]\!], [\![\beta]\!]\}$.

I then define the meaning of IF at an arbitrary initial state s, thus obtaining an extree, and take the union of all of these extrees. As follows:

(6) For any programs α_1 and α_2, any predicates B_1 and B_2, and any state $s \in \mathcal{S}$, the *meaning of* '$if\ B_1 \rightarrow \alpha_1\ [\!]\ B_2 \rightarrow \alpha_2\ fi$' *at* s is given by:

$$[\![if\ B_1 \rightarrow \alpha_1\ [\!]\ B_2 \rightarrow \alpha_2\ fi]\!](s) = \begin{cases} [\![\alpha]\!](s) & \text{if } s \in B_1 \text{ and } s \notin B_2 \\ [\![\beta]\!](s) & \text{if } s \notin B_1 \text{ and } s \in B_2 \\ [\![\alpha \vee \beta]\!](s) & \text{if } s \in B_1 \text{ and } s \in B_2 \\ [\![null]\!](s) & \text{if } s \notin B_1 \text{ and } s \notin B_2 \end{cases}$$

and the meaning of IF itself is:

$[\![if\ B_1 \rightarrow \alpha_1\ [\!]\ B_2 \rightarrow \alpha_2\ fi]\!] = \bigcup_{s \in \mathcal{S}} [\![if\ B_1 \rightarrow \alpha_1\ [\!]\ B_2 \rightarrow \alpha_2\ fi]\!](s)$.

The intention here should be clear. How does '$if\ B_1 \rightarrow \alpha_1\ [\!]\ B_2 \rightarrow \alpha_2\ fi$' execute from any initial state s? If B_1 is true but B_2 is not α_1 will be executed from s; if B_1 is false but B_2 is true then α_2 will be executed; if both are true exactly one of α_1 and α_2 will be nondeterministically selected and executed, and if neither B_1 nor B_2 is true nothing will happen. Two typical special cases of IF are '$if\ B\ do\ \alpha$', which I equate to '$if\ B \rightarrow \alpha\ [\!]\ \neg B \rightarrow null\ fi$',

and *if B do α else β fi*', which I abbreviate to '*if* $B \to \alpha \;[]\; \neg B \to \beta$'. It is then easy to read off their meanings from (6) above. This definition does not appear to be as simple as the definition of the execution relation $[if\ B_1 \to \alpha_1 \;[]\; B_2 \to \alpha_2\ fi]$ in (4.6)(e)′ (p 143). The main reason is that I have not introduced, for any predicate (or guard) B, a special program $B?$ as was done in Chapter 4.1 (p 110) to give the relational semantics of IF.

With Gries and Dijkstra I assume the guards to be well-defined in every state, so that the four possibilities enumerated above are the only ones. On the other hand, unlike Gries ([1981] p 132), Dijkstra ([1976] p 34) and Dijkstra and Scholten ([1990] p 144) I do *not* say that if no guard is true then IF does not terminate because there is a danger of confusion (as mentioned in Chapter 2.1 (p 35)) concerning what happens for any program α, when the composition 'IF;α' is executed from a state where no guard of IF is true. Instead I say that if no guard is true *nothing* should happen. What will be made clear by my exposition, I trust, is that having (literally) the *null*-option available makes for a tidy treatment of IF, both technically and conceptually. Such a change is warranted because my aim is to model the *algebra* of weakest preconditions, and that in doing so I am not constrained by any particular intuitive *semantics* of the constructs involved. A final point: each extree by assumption satisfies **Axiom 2** (of §1), hence a union of extrees over *every* state $s \in \mathcal{S}$ will be a flowset. Thus $[\![if\ B_1 \to \alpha_1 \;[]\; B_2 \to \alpha_2\ fi]\!]$ is well-defined as a flowset.

For the iterative command DO, in the simple form '*while B do* α', the earlier preparation (in §1) really pays off. First I need the following lemma:

(7) Lemma to **Definition (8)** *For any program* β, $[\![\beta]\!]^n = [\![\beta^n]\!]$, $\forall n \geq 0$.

Proof By induction on n.

For $n = 0$, $[\![\beta^0]\!] = [\![null]\!] = E$ and $[\![\beta]\!]^0 = E$.

Assume as induction hypothesis that $[\![\beta]\!]^n = [\![\beta^n]\!]$ for some n. Then

$$\begin{aligned}
[\![\beta]\!]^{n+1} &= [\![\beta]\!]^n \circ [\![\beta]\!] \text{ (by definition (1.11))} \\
&= [\![\beta^n]\!] \circ [\![\beta]\!] \text{ (by induction hypothesis)} \\
&= [\![\beta^n;\beta]\!] \text{ (by definition (2))} \\
&= [\![\beta^{n+1}]\!] \text{ (by definition of composition)}
\end{aligned}$$

□

The definition is quite simple: DO is the iteration of IF.

(8) $[\![while\ B\ do\ \alpha]\!] = [\![if\ B\ do\ \alpha]\!]^*\ \ (= lub\{[\![if\ B\ do\ \alpha]\!]^n\}_{n\geq 0} = lub\{[\![(if\ B\ do\ \alpha)^n]\!]\}_{n\geq 0})$

That is, by definitions (1.11) and (1.13) DO is defined as the least upper bound of the chain of flowsets arising from repeating the IF command. (Note: This is the second equality in (8). The third is proved in Lemma 7.) Intuitively, to perform '$while\ B\ do\ \alpha$' consists of repeatedly doing the following, until it has no further effect: check whether B is true, and if so do α. Recall the problems encountered in Chapter 4.6 (p 146) with describing the semantics of '$while\ B\ do\ \alpha$' in terms of execution relations. Definition (8) shows one virtue of the flowsets model over the relational model based on execution relations (in Chapter 4.6): '$while\ B\ do\ \alpha$' can easily be modelled as flowset.

I now come to the central definition, which is that of weakest precondition: for any program α and predicate Q, Dijkstra ([1976] p 16,17), Gries ([1981] (7.1) p 108) and Dijkstra and Scholten ([1990] p 129) are unanimous that '$wp(\alpha, Q)$' must denote the set of all those states such that execution of α begun in one of them is guaranteed to terminate, and when it does it satisfies Q. My definition will capture this intuition, but it adds a clarification: '$wp(\alpha, Q)$' will denote the set of all states from which α terminates *cleanly* (and satisfies Q upon termination). In this context, if α terminates but does not terminate cleanly (that is, terminates messily) it cannot satisfy Q, since $\perp$ is not a state.

(9) For any program α and predicate Q,

$$wp(\alpha, Q) = \{s \in \mathcal{S} | (\forall \mathbf{x} \in [\![\alpha]\!](s))[\mathbf{x} \in \mathcal{S}^+ \text{ and } last(\mathbf{x}) \in Q]\}$$

I can now verify the formulae of the algebra of weakest preconditions (presented in Chapter 2.2 (p 38, 40)).

(10) **Theorem** *For any program α and predicates Q and R:*

(a) (Gries [1981] (7.3)) **Law of the Excluded Miracle:** $wp(\alpha, \emptyset) = \emptyset$.

(b) (Gries [1981] (7.4)) **Distributivity of Conjunction:**

$$wp(\alpha, Q) \bigcap wp(\alpha, R) = wp(\alpha, Q \bigcap R).$$

(c) (Gries [1981] (7.5)) **Law of Monotonicity:**

$$\text{If } Q \subseteq R \text{ then } wp(\alpha, Q) \subseteq wp(\alpha, R).$$

(d) (Gries [1981] (7.6)) **Distributivity of Disjunction:**

$$wp(\alpha, Q) \bigcup wp(\alpha, R) \subseteq wp(\alpha, Q \bigcup R).$$

Proof All of these depend upon simple logical properties, such as in (b) the distribution of universal quantification over conjunction.

(a) If $wp(\alpha, \emptyset) \neq \emptyset$, say $s \in wp(\alpha, Q)$ then every $\mathbf{x} \in [\![\alpha]\!](s)$ terminates cleanly and in $\emptyset$, a contradiction. Hence $wp(\alpha, \emptyset) = \emptyset$.

(b) $s \in wp(\alpha, Q \bigcap R)$ iff $(\forall \mathbf{x} \in [\![\alpha]\!](s))[\mathbf{x} \in \mathcal{S}^+$ and $last(\mathbf{x}) \in Q \bigcap R]$

iff $(\forall \mathbf{x} \in [\![\alpha]\!](s))[\mathbf{x} \in \mathcal{S}^+$ and $last(\mathbf{x}) \in Q$ and $last(\mathbf{x}) \in R]$

iff $(\forall \mathbf{x} \in [\![\alpha]\!](s))[\mathbf{x} \in \mathcal{S}^+$ and $last(\mathbf{x}) \in Q]$ and

$(\forall \mathbf{x} \in [\![\alpha]\!](s))[\mathbf{x} \in \mathcal{S}^+$ and $last(\mathbf{x}) \in R]$

iff $s \in wp(\alpha, Q)$ and $s \in wp(\alpha, R)$

iff $s \in wp(\alpha, Q) \bigcap wp(\alpha, R)$

(c) Take $s \in wp(\alpha, Q)$ arbitrarily then every $\mathbf{x} \in [\![\alpha]\!](s)$ terminates cleanly and in Q and hence in R (since $Q \subseteq R$). So $s \in wp(\alpha, R)$ as required.

(d) Note $Q \subseteq Q \bigcup R$ and $R \subseteq Q \bigcup R$, then by (c) $wp(\alpha, Q) \subseteq wp(\alpha, Q \bigcup R)$ and $wp(\alpha, R) \subseteq wp(\alpha, Q \bigcup R)$; hence $wp(\alpha, Q) \bigcup wp(\alpha, R) \subseteq wp(\alpha, Q \bigcup R)$. □

To check that in the context of this chapter the converse of Theorem (10)(d) does not hold in general an example such as that of Gries ([1981] p 111) would suffice. But the converse *does* hold for deterministic programs.

(11) A program α is said to be *deterministic* iff for every $s \in \mathcal{S}$ $[\![\alpha]\!](s)$ is a singleton set.

That is, from any initial state α can proceed to execute in exactly one way.

(12) **Theorem** *For any predicates Q and R, and any deterministic program α*
(Gries [1981] (7.7)) $wp(\alpha, Q) \bigcup wp(\alpha, R) = wp(\alpha, Q \bigcup R)$.

Proof By theorem (10(d)) the left to right inclusion holds; so for the reverse inclusion take any $s \in wp(\alpha, Q \bigcup R)$. Then the (unique) exseq in $[\![\alpha]\!](s)$, say $\mathbf{x}$ terminates cleanly and in $Q \bigcup R$. Hence $\mathbf{x}$ terminates cleanly and either in Q or in R; so $s \in wp(\alpha, Q) \bigcup wp(\alpha, R)$. □

The atomic programs are easy to characterise from Definition (1).

(13) **Theorem** *For any predicate* Q

(a) (Gries [1981] (8.1))	$wp(skip, Q) = Q.$
(b) (Gries [1981] (8.2))	$wp(abort, Q) = \emptyset.$
(c) (Dijkstra and Scholten [1990], (7.12))	$wp(havoc, Q) = \mathcal{S}.$
(d)	$wp(null, Q) = Q.$

Proof By definition (1):

(a) Execution of *skip* is guaranteed to terminate cleanly after one step and leaves the state unchanged; hence $wp(skip, Q) = Q$.

(b) Execution of *abort* never terminates cleanly but is guaranteed to terminate in $\perp \notin \mathcal{S}$; hence $wp(abort, Q) = \emptyset$.

(c) Execution of *havoc* is guaranteed to terminate cleanly but in any possible state; hence $wp(havoc, Q) = \mathcal{S}$.

(d) From any initial state s, *null* has that same state also as a terminal state, hence $wp(null, Q) = Q$. □

I now come to composition: the place where the relational models using representation methods $\mathbf{R_j(n)}$ (for $\mathbf{j} = 1, 2, 3$; $\mathbf{n} = 1, 2, 3, 4, 5, 6$) and $\mathbf{R_j(n)}$ (for $\mathbf{j} = 1, 2$; $\mathbf{n} = 7, 8$) fail. Recall Theorem (4.4.2) established a sufficient condition under which Gries (8.3) holds: *total* relations (Chapter 4.1, p 109) must be used to model programs. That is, for Gries (8.3) to hold each state must be related to at least one outcome in the sense that something must happen when a program is activated in a state. Note that this condition corresponds to my **Axiom 1** for flowsets.

(14) **Theorem** *For any programs* α *and* β*, and any predicate* Q*,*

$$wp(\alpha; \beta, Q) = wp(\alpha, wp(\beta, Q)).$$

Proof Left to right: Let $s \in wp(\alpha; \beta, Q)$, then by (9) any $\mathbf{z} \in [\![\alpha; \beta]\!](s)$ terminates cleanly, and in Q. To show that $s \in wp(\alpha, wp(\beta, Q))$, let $\mathbf{u} \in [\![\alpha]\!](s)$ arbitrarily. If $\mathbf{u} \in \mathcal{S}^{\perp}$ or $\mathbf{u} \in \mathcal{S}^{\infty}$ then also by (1.3) (and (2)) I would have $\mathbf{u} \in [\![\alpha; \beta]\!](s)$, which would then contradict the fact that $\mathbf{u}$ must terminate cleanly. So $\mathbf{u} \in \mathcal{S}^{+}$, hence $last(\mathbf{u}) \in \mathcal{S}$. To show that $last(\mathbf{u}) \in wp(\beta, Q)$, consider any $\mathbf{v} \in [\![\beta]\!](last(\mathbf{u}))$. Then $\mathbf{u} \circ \mathbf{v} \in [\![\alpha; \beta]\!](s)$, hence by assumption $\mathbf{u} \circ \mathbf{v}$ terminates cleanly, and in Q. But then $\mathbf{v}$ must terminate cleanly, and in Q. Hence $last(\mathbf{u}) \in wp(\beta, Q)$, as required.

Right to left: Let $s \in wp(\alpha, wp(\beta, Q))$; then any $\mathbf{x} \in [\![\alpha]\!](s)$ terminates cleanly and in $wp(\beta, Q)$. So $last(\mathbf{x}) \in wp(\beta, Q)$; hence any $\mathbf{y} \in [\![\beta]\!](last(\mathbf{x}))$ terminates cleanly and in Q. To show that $s \in wp(\alpha;\beta, Q)$, take any $\mathbf{z} \in [\![\alpha;\beta]\!](s)$. Then if $\mathbf{z} \in \mathcal{S}^{\perp} \bigcup \mathcal{S}^{\infty}$ $\mathbf{z} \in [\![\alpha]\!](s)$, which would contradict the fact that every exseq in $[\![\alpha]\!](s)$ terminates cleanly. So $\mathbf{z} \in \mathcal{S}^{+}$, say $\mathbf{z} = \mathbf{u} \circ \mathbf{v}$ for some $\mathbf{u} \in [\![\alpha]\!](s) \bigcap \mathcal{S}^{+}$ and some $\mathbf{v} \in [\![\beta]\!](last(\mathbf{u}))$. Then since $last(\mathbf{u}) \in wp(\beta, Q)$, $\mathbf{v}$ and hence $\mathbf{z}$ terminates cleanly and in Q. Hence $s \in wp(\alpha;\beta, Q)$ as required. □

As with the definition of the assignment statement I also pass lightly over its weakest pre-condition result: issues such as definability and non-classical conjunction raised by Gries [1981] (9.1.1) are (as in the Chapter 4) not germane to my discussion. What I should do is check:

(15) **Theorem** *For any program variable z, expression e and predicate Q:*

$$wp(\text{`}z := e\text{'}, Q) = \{s \mid s[e/z] \in Q\}.$$

Proof Let $s \in wp(\text{`}z := e\text{'}, Q)$ then every $\mathbf{x} \in [\![z := e]\!](s)$ terminates cleanly and in $Q$. But by definition (4) $\mathbf{x} = (s, s[e/z])$; hence $s[e/z] \in Q$. For the reverse direction, let $s$ be such that $s[e/z] \in Q$. To show that $s \in wp(\text{`}z := e\text{'}, Q)$ take any $\mathbf{x} \in [\![z := e]\!](s)$, then by definition (4) $\mathbf{x} = (s, s[e/z])$. Hence $\mathbf{x}$ terminates cleanly and in Q; so $s \in wp(\text{`}z := e\text{'}, Q)$. □

I now come to the IF statement, '$if\ B_1 \rightarrow \alpha_1 \ [\!]\ B_2 \rightarrow \alpha_2\ fi$'. As mentioned, the Dijkstra/Gries idea is that if no guard is true IF does not terminate. Accordingly, in Gries [1981] (10.3b) $wp(\text{IF}, Q)$ is given as an intersection of three facts: some guard is true, if B_1 is true then we have $wp(\alpha_1, Q)$, and if B_2 is true then we have $wp(\alpha_2, Q)$. My treatment, as pointed earlier, differs from that of Dijkstra/Gries in also covering explicitly the case where no guard is true: in that case $wp(\text{IF}, Q)$ is just Q. It seems to me that this better captures the intuition behind IF than the Dijkstra/Gries idea that IF is non-terminating when no guard is true. On my treatment I get:

(16) **Theorem** *For any programs α and β, and any predicates B_1, B_2, and Q,*

$wp(if\ B_1 \to \alpha_1 \,[]\, B_2 \to \alpha_2\ fi, Q)$

$= [B_1 \cap \neg B_2 \cap wp(\alpha_1, Q)] \cup [\neg B_1 \cap B_2 \cap wp(\alpha_2, Q)] \cup [B_1 \cap B_2 \cap wp(\alpha_1 \vee \alpha_2, Q)]$

$\cup [\neg B_1 \cap \neg B_2 \cap Q]$.

Proof Left to right: Let $s \in wp(if\ B_1 \to \alpha_1 \,[]\, B_2 \to \alpha_2\ fi, Q)$. Then every $\mathbf{x} \in [\![if\ B_1 \to \alpha_1 \,[]\, B_2 \to \alpha_2\ fi]\!](s)$ terminates cleanly and in Q. To show s is in the right hand side I distinguish four mutually exclusive and jointly exhaustive cases: either $s \in B_1 \cap \neg B_2$ or $s \in \neg B_1 \cap B_2$ or $s \in B_1 \cap B_2$ or $s \in \neg B_1 \cap \neg B_2$. I only consider the first case; the others are similar: If $s \in B_1 \cap \neg B_2$ then by definition (6) $[\![if\ B_1 \to \alpha_1 \,[]\, B_2 \to \alpha_2\ fi]\!](s) = [\![\alpha_1]\!](s)$; hence every $\mathbf{x} \in [\![\alpha_1]\!](s)$ terminates cleanly and in Q. So $s \in B_1 \cap \neg B_2 \cap wp(\alpha_1, Q)$.

Right to left: Let $s \in$ right hand side arbitrarily.

To show $s \in wp(if\ B_1 \to \alpha_1 \,[]\, B_2 \to \alpha_2\ fi, Q)$ I consider four cases, either: $s \in B_1 \cap \neg B_2$ or $s \in \neg B_1 \cap B_2$ or $s \in B_1 \cap B_2$ or $s \in \neg B_1 \cap \neg B_2$. I need only consider the first case; the others are similar: If $s \in B_1 \cap \neg B_2$ then $s \in wp(\alpha_1, Q) = wp(if\ B_1 \to \alpha_1 \,[]\, B_2 \to \alpha_2\ fi, Q)$ (by assumption and by definition (6)). □

This presentation of wp(IF,Q) exactly parallels the definition of IF in (6). That is, wp(IF,Q) breaks down as follows: If the first guard is true and the second false I are dealing with $wp(\alpha_1, Q)$; if the first guard is false and the second true I are dealing with $wp(\alpha_2, Q)$; if both guards are true I are dealing with the weakest precondition of *one* of α_1 or α_2 (without knowing which), and if no guard is true the weakest precondition is Q itself.

What can we say about the weakest precondition for (nondeterministic) choice operator '$\vee$'? Only this:

(17) **Theorem** *For any programs α and β, and any predicate Q,*

$$wp(\alpha \vee \beta, Q) \supseteq wp(\alpha, Q) \cap wp(\beta, Q).$$

Proof Let $s \in wp(\alpha, Q) \cap wp(\beta, Q)$; then every $\mathbf{x} \in [\![\alpha]\!](s)$ must terminate cleanly and in Q, and so must every $\mathbf{x} \in [\![\beta]\!](s)$. To show that $s \in wp(\alpha \vee \beta, Q)$ take any $\mathbf{x} \in [\![\alpha \vee \beta]\!](s)$. Then $\mathbf{x} \in \epsilon.\{[\![\alpha]\!](s), [\![\beta]\!](s)\}$; so $\mathbf{x}$ must terminate cleanly and in Q. Hence $\mathbf{x} \in wp(\alpha \vee \beta, Q)$. □

The converse of (17) fails, since if every $\mathbf{x} \in [\![\alpha \vee \beta]\!](s)$ terminates cleanly and in Q this only tells us (by (5)) that every $\mathbf{x}$ in *one* of $[\![\alpha]\!](s)$ and $[\![\beta]\!](s)$ terminates cleanly and in Q. In consequence we can prove the analogue of Gries [1981] (10.3b) in one direction only.

(18) **Theorem** *For any programs α and β, and any predicates B_1, B_2 and Q,*

$wp(if\ B_1 \rightarrow \alpha_1 \ [\!]\ B_2 \rightarrow \alpha_2\ fi, Q)$

$\supseteq [(B_1 \cup B_2) \cap (\neg B_1 \cup wp(\alpha_1, Q)) \cap (\neg B_2 \cup wp(\alpha_2, Q))] \cup [\neg B_1 \cap \neg B_2 \cap Q]$.

Proof

Let s be an element of the set on the right hand side of '$\supseteq$'; there are then two cases. If $s \in \neg B_1 \cap \neg B_2 \cap Q$ then by (13)(d) $s \in wp(null, Q)$ and by (6) and (8) this is exactly $wp(if\ B_1 \rightarrow \alpha_1 \ [\!]\ B_2 \rightarrow \alpha_2\ fi, Q)$. In the second case $s \in B_1 \cup B_2$, if $s \in B_1$ then it is also in $wp(\alpha_1, Q)$ and if $s \in B_2$ then it is also in $wp(\alpha_2, Q)$. Distinguish three subcases: $s \in B_1 \cap \neg B_2$ or $s \in \neg B_1 \cap B_2$ or $s \in B_1 \cap B_2$. In the first subcase $s \in wp(\alpha_1, Q)$ which *in this case* by (6) and (9) equals $wp(if\ B_1 \rightarrow \alpha_1 \ [\!]\ B_2 \rightarrow \alpha_2\ fi, Q)$. In the second subcase $s \in wp(\alpha_2, Q)$ which likewise in *that case* equals $wp(if\ B_1 \rightarrow \alpha_1 \ [\!]\ B_2 \rightarrow \alpha_2\ fi, Q)$. In the third case $s \in wp(\alpha_1, Q) \cap wp(\alpha_2, Q)$, which by (16) is contained in $wp(\alpha \vee \beta, Q)$, which by (6) and (9) *in this case* equals $wp(if\ B_1 \rightarrow \alpha_1 \ [\!]\ B_2 \rightarrow \alpha_2\ fi, Q)$. □

I spoke of 'the analogue' of Gries [1981] (10.3b): this indicates my addition of the extra possibility that no guard is true. With or without this addition it remains true that what Gries takes to be wp(IF, Q) is *included* in what I take to be wp(IF,Q). The point is that my wp(IF,Q) is thus *weaker* than wp(IF,Q) in Gries; since finding the *weakest* precondition is what the game is all about I count this as a virtue of my approach.

For '$if\ B\ do\ \alpha$' and '$if\ B\ do\ \alpha\ else\ \beta\ fi$' I get as special cases from (16):

(19) **Corollary** *For any programs α and β, and any predicates B and Q,*

$wp(if\ B\ do\ \alpha, Q) = [B \cap wp(\alpha, Q)] \cup [\neg B \cap Q]$

$wp(if\ B\ do\ \alpha\ else\ \beta\ fi, Q) = [B \cap wp(\alpha, Q)] \cup [\neg B \cap wp(\beta, Q)]$ □

Finally, I come to the iterative command, '$while\ B\ do\ \alpha$'. Like Gries ([1981] p 140) I define for any given predicate Q a sequence $H_n(Q)$ of predicates, where $H_n(Q)$ represents the set

of all states from which execution of DO terminates (cleanly!) in n or fewer iterations, with Q true. But my definition simplifies that of Gries. Namely:

(20) For '*while* B *do* α' define predicates $H_n(Q)$, $n \geq 0$, by:

$H_0(Q) = \neg B \cap Q$

$H_{n+1}(Q) = wp(if\ B\ do\ \alpha, H_n(Q)), \quad \forall n \geq 0.$

The simplification is possible because of my treatment of IF in the case where no guards are true. To prove that nothing is omitted by the simplification I require two Lemmas.

(21) **Lemma** $\neg B \cap H_n(Q) = \neg B \cap Q, \quad \forall n \geq 0.$

Proof By induction on n as follows:

For $n = 0$, $\neg B \cap H_0(Q) = \neg B \cap (\neg B \cap Q) = \neg B \cap Q$.

Assume as induction hypothesis that $\neg B \cap H_n(Q) = \neg B \cap Q$ for some n. Then

$$\begin{aligned} \neg B \cap H_{n+1}(Q) &= \neg B \cap wp(if\ B\ do\ \alpha, H_n(Q)) \text{ (by (20))} \\ &= \neg B \cap ([B \cap wp(\alpha, H_n(Q))] \cup [\neg B \cap H_n(Q)]) \text{ (by Corollary (19))} \\ &= \neg B \cap [\neg B \cap Q] \\ &= \neg B \cap Q \end{aligned}$$

Hence by the principle of mathematical induction, $\forall n \geq 0$, $\neg B \cap H_n(Q) = \neg B \cap Q$ □

This says that any state from which DO terminates in n or fewer iterations, and in which B is false, is also a state in which Q is true.

(22) **Lemma** $H_{n+1}(Q) = [\neg B \cap Q] \cup [B \cap wp(\alpha, H_n(Q))], \quad \forall n \geq 0.$

Proof For any $n \geq 0$,

$$\begin{aligned} H_{n+1}(Q) &= wp(if\ B\ do\ \alpha, H_n(Q)) \text{ (by (20))} \\ &= [\neg B \cap H_n(Q)] \cup [B \cap wp(\alpha, H_n(Q))] \text{ (by Corollary (19))} \\ &= [\neg B \cap Q] \cup [B \cap wp(\alpha, H_n(Q))] \text{ (by Lemma (21)).} \end{aligned}$$ □

This gives exactly the *form* of Gries's $H_{n+1}(Q)$. An inductive argument then suffices to show that it is also the same *set*. So I get:

(23) **Theorem** *$H_n(Q)$ as defined by* Gries ([1981] p 140) *on the basis of his definition (10.3b) (p 132) of wp(IF,Q) is the same set $\forall n$ as $H_n(Q)$ defined in Definition (20) on the basis of wp(IF,Q) given in Theorem (16), arising from our Definition (6) of IF.*

Proof By induction on n.

For $n = 0$, $H_0(Q)_{Gries} = \neg B \bigcap Q = H_0(Q)_{(B\&R)}$.

Assume as induction hypothesis $H_n(Q)_{Gries} = H_n(Q)_{(B\&R)}$ for some n. Then

$$\begin{aligned} H_{n+1}(Q)_{Gries} &= H_0(Q)_{Gries} \bigcup wp(if\ B\ do\ \alpha, H_n(Q)_{Gries}) \text{ (Gries [1981] p 140)} \\ &= [\neg B \bigcap Q] \bigcup [B \bigcap (\neg B \bigcup wp(\alpha, H_n(Q)_{Gries})] \text{ (Gries [1981] 10.3b)} \\ &= [\neg B \bigcap Q] \bigcup [B \bigcap wp(\alpha, H_n(Q)_{Gries})] \\ &= [\neg B \bigcap Q] \bigcup [B \bigcap wp(\alpha, H_n(Q)_{(B\&R)})] \text{ (by induction hypothesis)} \\ &= H_{n+1}(Q)_{(B\&R)} \text{ (by Lemma (22)).} \end{aligned}$$

□

It remains to verify Gries [1981] (11.2).

(24) **Lemma** *For 'while B do α', and any predicate Q, $\{H_n(Q)\}_{n\geq 0}$ forms a chain under the set-theoretic ordering $\subseteq$.*

Proof I establish by induction on n that $H_n(Q) \subseteq H_{n+1}(Q)$, $\forall n \geq 0$.

For $n = 0$, $H_0(Q) = \neg B \bigcap Q \subseteq [\neg B \bigcap Q] \bigcup [B \bigcap wp(\alpha, \neg B \bigcap Q)] = H_1(Q)$.

Assume as induction hypothesis $H_n(Q) \subseteq H_{n+1}(Q)$ for some n. Then,

$$\begin{aligned} H_{n+1}(Q) &= [\neg B \bigcap Q] \bigcup [B \bigcap wp(\alpha, H_n(Q))] \text{ (by Lemma (22))} \\ &\subseteq [\neg B \bigcap Q] \bigcup [B \bigcap wp(\alpha, H_{n+1}(Q))] \text{ (by induction hypothesis and monotonicity)} \\ &= H_{n+2}(Q) \text{ (by Lemma (22)).} \end{aligned}$$

Hence by the principle of mathematical induction, $H_n(Q) \subseteq H_{n+1}(Q)$, $\forall n \geq 0$. □

This means that Gries really characterises $wp(DO, \alpha)$ as the least upper bound of the chain (under $\subseteq$) of the $H_n(Q)$'s. And so do I. Again I need two Lemmas. Both demonstrate my approach of defining DO in terms of IF.

(25) **Lemma** $H_n(Q) = wp((if\ B\ do\ \alpha)^n, \neg B \bigcap Q)$, $\forall n \geq 0$.

Proof [Note: For any program α, $\alpha^0 = null$ and $\alpha^{n+1} = \alpha^n; \alpha$.]

By induction on n.

For $n = 0$, $H_0(Q) = \neg B \bigcap Q$ and $wp((if\ B\ do\ \alpha)^0, \neg B \bigcap Q) = wp(null, \neg B \bigcap Q) = \neg B \bigcap Q$. Assume as induction hypothesis $H_n(Q) = wp((if\ B\ do\ \alpha)^n, \neg B \bigcap Q)$ for some n. Then,

$$
\begin{aligned}
H_{n+1}(Q) &= wp(if\ B\ do\ \alpha, H_n(Q)) \text{ (by (20))} \\
&= wp(if\ B\ do\ \alpha, wp((if\ B\ do\ \alpha)^n, \neg B \bigcap Q)) \text{ (by induction hypothesis)} \\
&= wp((if\ B\ do\ \alpha); (if\ B\ do\ \alpha)^n, \neg B \bigcap Q) \text{ (by Theorem (14))} \\
&= wp((if\ B\ do\ \alpha)^{n+1}, \neg B \bigcap Q) \text{ (since ; is associative).}
\end{aligned}
$$

Hence by the principle of mathematical induction, $\forall n \geq 0$,

$H_n(Q) = wp((if\ B\ do\ \alpha)^n, \neg B \bigcap Q)$. □

(26) **Lemma** *For any $s \in \mathcal{S}$, if $[\![while\ B\ do\ \alpha]\!](s)$ is finite then $\exists n \in \mathcal{N}$ such that $[\![while\ B\ do\ \alpha]\!](s) = [\![(if\ B\ do\ \alpha)^n]\!](s)$, and $t \notin B$ for every leaf t of this extree.*

Proof By saying 'the tree is finite' I mean 'only has branches of finite length'. (Since I are dealing with unbounded nondeterminism there may well be infinitely many branches). From (8) I get that $[\![while\ B\ do\ \alpha]\!] = lub\{[\![(if\ B\ do\ \alpha)^n]\!]\}_{n\geq 0}$, hence if for any particular $s \in \mathcal{S}$ $[\![while\ B\ do\ \alpha]\!](s)$ is finite it follows from Theorem (1.12) that there must be a least number $m \in \mathcal{N}$ such that

$$[\![(if\ B\ do\ \alpha)^m]\!](s) = [\![(if\ B\ do\ \alpha)^{m+1}]\!](s) = \ldots$$

But then $[\![while\ B\ do\ \alpha]\!](s) = [\![(if\ B\ do\ \alpha)^m]\!](s)$, and $t \notin B$ (since otherwise $[\![(if\ B\ do\ \alpha)^{m+1}]\!](s)$ would extend $[\![(if\ B\ do\ \alpha)^m]\!](s)$). □

The idea here is quite simple: a terminating DO from an initial state s is precisely the n-fold composition of an IF-statement, for some $n \in \mathcal{N}$.

(27) **Theorem** (Gries [1981] (11.2)) *For any program α and predicates B and Q,*

$wp(while\ B\ do\ \alpha, Q) = \bigcup_{n\geq 0} H_n(Q) = \bigcup_{n\geq 0} wp((if\ B\ do\ \alpha)^n, \neg B \cap Q)$.

Proof Left to right: Let $s \in wp(while\ B\ do\ \alpha, Q)$. Then every exseq $\mathbf{x}$ in $[\![while\ B\ do\ \alpha]\!](s)$ terminates cleanly and in Q. Hence $[\![while\ B\ do\ \alpha]\!](s)$ is finite, so by Lemma (26) $\exists m \in \mathcal{N}$ such that $[\![while\ B\ do\ \alpha]\!](s) = [\![(if\ B\ do\ \alpha)^m]\!](s)$ (and $t \notin B$ for every leaf t of this extree). But then every exseq $\mathbf{x} \in [\![(if\ B\ do\ \alpha)^m]\!](s)$ terminates cleanly and in Q, hence in $\neg B \cap Q$. Thus by Lemmas (25) and (24), $s \in wp((if\ B\ do\ \alpha)^m, \neg B \cap Q) = H_m(Q) \subseteq \bigcup_{n\geq 0} H_n(Q)$.

Right to left: Let $s \in \bigcup_{n\geq 0} wp((if\ B\ do\ \alpha)^n, \neg B \cap Q)$, say $s \in wp((if\ B\ do\ \alpha)^m, \neg B \cap Q)$ for some $m \in \mathcal{N}$. Then every exseq $\mathbf{x} \in [\![(if\ B\ do\ \alpha)^m]\!](s)$ terminates cleanly and in $\neg B \cap Q$. But then

$$[\![(if\ B\ do\ \alpha)^m]\!](s) = [\![(if\ B\ do\ \alpha)^{m+1}]\!](s) = \ldots$$

and hence $[\![(if\ B\ do\ \alpha)^m]\!](s) = (lub\{[\![(if\ B\ do\ \alpha)^n]\!]\}_{n\geq 0})(s) =$ $[\![while\ B\ do\ \alpha]\!](s)$ by (8). Hence every $\mathbf{x}$ in $[\![while\ B\ do\ \alpha]\!](s)$ terminates cleanly and in Q and so $s \in wp(while\ B\ do\ \alpha, Q)$. □

6.3 Invariants

Recall from Chapter 2.2 (p 41, 42) that with DO, as with IF, the weakest precondition is not always the most useful precondition. The sought-after precondition is called an *invariant* of the loop. To model the notion of an invariant in the context of this chapter I come forward with two suggestions.

(1) **Suggestion 1** Instead of restricting the notion of an invariant to loops, define it for any program α.

That is, for any program α a predicate $I \subseteq \mathcal{S}$ will be called an invariant of α iff, $\forall s \in I$, if α is executed from s then every final state is again an element of I. However there is an immediate problem: α may not terminate cleanly, or may not terminate at all, so that an appropriate final state may not exist. For this I have:

(2) **Suggestion 2** Think of invariants by analogy with subalgebras: 'I is an invariant of α' is analogous to a subset of an algebra being *closed* under a given operation.

The virtue of this suggestion is that the problem just raised has been exhaustively investigated in Universal Algebra, and so we may borrow from there. Since α may not terminate (cleanly, or at all), we may think of it as analogous to a *partial operation* in an algebra. The question of how to define invariants for programs which do not terminate cleanly is then analogous to this: *What is the correct notion of subalgebra for partial algebras?* For this, consider the comment of Grätzer ([1978] p 79):

> For algebras there is ony one reasonable way to define the concepts of subalgebra, homomorphism and congruence relation. For partial algebras we will define

three different types of subalgebra, three types of homomorphism, and two types of congruence relation ... all of these concepts have their merits and their drawbacks, and each particular situation determines which one should be used.

Time limits disallowed further investigation, so I simply report that of the three kinds of subalgebra considered by Grätzer I may use two in the present context to give the following alternative notions of an invariant of a program α.

(3) **Alternative 1** A predicate I is called an *invariant* of a program α iff $\forall s \in I$ the extree $[\![\alpha]\!](s)$ is finite and all leaves are $\in I$.

(4) **Alternative 2** A predicate I is called an *invariant* of a program α iff $\forall s \in I$, *if* the extree $[\![\alpha]\!](s)$ is finite *then* all its leaves are $\in I$.

Recall from Chapter 2.2 (p 41) that in the Dijkstra/Gries formulation of invariants termination is not built in — it must be proved separately by a bound function. Thus Dijkstra/Gries implicitly select **Alternative 2**, hence, for current purposes, so do I.

I now prove in the context of this chapter a version of the theorem called 'The Basic Theorem for the Repetitive Construct' (also 'The Fundamental Invariance Theorem for Loops') in Dijkstra ([1976] p 38), Theorem (11.6) ('a theorem concerning a loop, an invariant and a bound function') in Gries ([1981] p 144) and the 'Main Repetition Theorem' in Dijkstra and Scholten ([1990] p 180). Before proving my version I need the following Lemma:

(5) **Lemma** $wp(\textit{while } B \textit{ do } \alpha, Q) = wp(\textit{while } B \textit{ do } \alpha, \neg B \bigcap Q)$

Proof By Theorem (2.27) it suffices to show that $\forall n \geq 0,\ H_n(Q) = H_n(\neg B \bigcap Q)$. We use induction on n as follows:

For $n = 0$, $H_0(Q) = \neg B \bigcap Q$ and $H_0(\neg B \bigcap Q) = \neg B \bigcap (\neg B \bigcap Q) = \neg B \bigcap Q$.

Assume as induction hypothesis $H_n(Q) = H_n(\neg B \bigcap Q)$ for some n. Then,

$$\begin{aligned} H_{n+1}(Q) &= [\neg B \bigcap Q] \bigcup [B \bigcap wp(\alpha, H_n(Q))] \text{ (by Lemma (2.22))} \\ &= [\neg B \bigcap Q] \bigcup [B \bigcap wp(\alpha, H_n(\neg B \bigcap Q))] \text{ (by induction hypothesis)} \\ &= [\neg B \bigcap (\neg B \bigcap Q)] \bigcup [B \bigcap wp(\alpha, H_n(\neg B \bigcap Q))] \\ &= H_{n+1}(\neg B \bigcap Q) \text{ (by Lemma (2.22))} \end{aligned}$$

Hence by the principle of mathematical induction, $\forall n \geq 0,\ H_n(Q) = H_n(\neg B \bigcap Q)$. □

(6) **Theorem** *For any predicates I, B and Q, and any program α, if*

(a) $\neg B \bigcap I \subseteq Q$, and

(b) $B \bigcap I \subseteq wp(\alpha, I)$, and

(c) $[\![while\ B\ do\ \alpha]\!](s)$ *is finite,* $\quad \forall s \in I$,

then

$$I \subseteq wp(while\ B\ do\ \alpha, Q).$$

Proof It suffices to show that $I \subseteq wp(while\ B\ do\ \alpha, I)$ since by (a) and monotonicity of wp, $wp(while\ B\ do\ \alpha, \neg B \bigcap I) \subseteq wp(while\ B\ do\ \alpha, Q)$ and using an inductive argument it is easy to verify that

$$wp(while\ B\ do\ \alpha, I) = wp(while\ B\ do\ \alpha, \neg B \bigcap I).$$

So let $s \in I$ then either $s \in B$ or $s \in \neg B$. If $s \in \neg B$ then $[\![while\ B\ do\ \alpha]\!](s) = [\![(if\ B\ do\ \alpha)^0]\!](s) = null(s) = \{(s)\}$. Hence every exseq in this extree terminates cleanly and in I, so $s \in wp(while\ B\ do\ \alpha, I)$. Now suppose $s \in B$. We must show that $\forall \mathbf{x} \in [\![while\ B\ do\ \alpha]\!](s)$, $\mathbf{x}$ terminates cleanly and in I. By (c) and Lemma (2.26) $\exists m \in \mathcal{N}$ such that $[\![while\ B\ do\ \alpha]\!](s) = [\![(if\ B\ do\ \alpha)^m]\!](s)$ (and $t \notin B$ for every leaf of this extree). So we need only show that every exseq $\mathbf{x}$ in $[\![(if\ B\ do\ \alpha)^m]\!](s)$ terminates cleanly and in I. But $[\![(if\ B\ do\ \alpha)^m]\!] = [\![if\ B\ do\ \alpha]\!]^m$, hence any such $\mathbf{x}$ has m nested initial subsequences $\mathbf{x}_i$, $1 \leq i \leq m$, such that $last(\mathbf{x}_i) \in B$ for $1 \leq i < m$. But then since $s\ (= first(\mathbf{x})) \in B \bigcap I$ we get from (b) that $last(\mathbf{x}_i) \in I$ for $1 \leq i \leq m$. Hence, in particular, $last(\mathbf{x}) \in I$, as required. □

Index of Notation

Bibliography

(Note: an item annotated with a * after the year of publication is a secondary reference, that is, one I found referenced in another primary source but did not consult the original myself.)

Andrews, P.B. [1986]. *An Introduction to Mathematical Logic and Type Theory: To Truth Through Proof.* Academic Press.

Apt, K.R. [1984]. Ten years of Hoare's logic: A survey part II: Nondeterminism. *Theoretical Computer Science* **28**. p 83–109.

Back, R.J.R. [1980]. Semantics of unbounded nondeterminism. In *Proceedings of the International Colloquium on Automata, Languages and Programming 80, Lecture Notes in Computer Science 85.* Springer-Verlag: Berlin-Heidelberg-New York. p 51–63.

Backhouse, R.C. [1986]. *Program Construction and Verification.* Prentice-Hall: Englewood Cliffs, New Jersey.

Blikle, A. [1977]. A comparative review of some program verification methods. *Lecture Notes in Computer Science 53 Mathematical Foundations of Computer Science.* (J. Gruska (ed)). Springer-Verlag: Berlin-Heidelberg-New York. p 17–33.

Blikle, A. [1981]. The clean termination of iterative programs. *Acta Informatica* **16**. p 199–217.

Blikle, A. [1987]. Proving programs by sets of computations. *Lecture Notes in Computer Science 288.* Springer-Verlag: Berlin-Heidelberg-New York. p 333–358.

Boom, H.J. [1982]. A weaker precondition for loops. *ACM Transactions of Programming Languages and Systems* **4** (4). p 668–677.

Brink, C. [1978]. *The Algebra of Relations.* Ph D. Thesis. University of Cambridge.

Brink, C. [1981]. Boolean modules. *Journal of Algebra* **71** (2). p 291–313.

Brink, C. [19?]. Power structures. (To appear in *Algebra Universalis*).

Brink, C. and I. Rewitzky. [19?]. Modelling the algebra of weakest preconditions. (To appear in *South African Computer Journal*).

Church, A. [1940]*. A formulation of the simple theory of types. *Journal of Symbolic Logic* **5**. p 56–68.

De Bakker, J.W. and W.P. de Roever. [1973]. A calculus for recursive program schemes. In *Proceedings of the 1st International Colloquium on Automata, Languages and Programming.* (M. Nivat (ed)). North Holland. p 167-196.

De Bakker, J.W. [1976]*. Semantics and termination of nondeterministic programs. In *Proceedings of the 3rd International Colloquium on Automata, Languages and Programming.* (S. Michaelson and R. Milner (eds)). Edinburgh, Scotland. p 435–477.

De Bakker, J.W. [1978]. Recursive programs as predicate transformers. In *Formal Descriptions of Programming Concepts.* (E.J. Neuhold (ed)). North-Holland: Amsterdam. p 165–181.

De Bakker, J.W. [1980]. *Mathematical Theory of Program Correctness.* Prentice Hall: Englewood Cliffs, New Jersey.

De Roever, W.P. [1976]. Dijkstra's predicate transformer, non-determinism, recursion and termination. *Lecture Notes in Computer Science 45, Mathematical Foundations of Computer Science.* Springer-Verlag: Berlin-Heidelberg-New York. p 472–481.

Dijkstra, E.W. [1975]. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM* **18** (8). p 453–458.

Dijkstra E.W. [1976]. *A Discipline of Programming.* Prentice-Hall: Englewood Cliffs, New Jersey.

Dijkstra, E.W. and C.S. Scholten. [1990]. *Predicate Calculus and Program Semantics.* Springer-Verlag: New York.

Dromey, G. [1989]. *Program Derivation: The Development of Programs from Specifications.* Addison-Wesley: Singapore.

Egli, H. [1975]*. A mathematical model for nondeterministic computations. Research Report ETH. Zurich, Switzerland.

Emerson, E.A. [1990]. Temporal and modal logic. In *Handbook of Theoretical Computer Science Vol B Formal Models and Semantics.* (J. van Leeuwen (ed)). p 995–1072.

Enderton, H.B. [1977]. *Elements of Set Theory.* Academic Press: New York.

Floyd, R.W. [1967a]*. Assigning meanings to programs. In *Proceedings Symposium of Applied Mathematics* **19**. American Mathematical Society: Providence R.I. p 19–32.

Floyd, R.W. [1967b]. Nondeterministic algorithms. *Journal of the ACM* **4**. p 636–644.

Goldblatt, R. [1989]. Varieties of complex algebras. *Annals of Pure and Applied Logic* **44**. p 173–242.

Gordon, M.J.C. [1989a]. Mechanising programming logics in Higher Order Logic. In *Current Trends in Hardware Verification and Automated Theorem Proving.* (G. Birtwistle and P.A. Subrahmanyam (eds)). Springer-Verlag. p 388–439.

M.J.C. Gordon *et al.* [1989b]. *The HOL System: DESCRIPTION.* Cambridge Research Center. SRI International.

Grätzer, G. [1978]. *Universal Algebra.* 2nd Edition. Springer-Verlag: Berlin.

Grätzer, G. and S. Whitney. [1984]. Infinitary varieties of structures closed under the formation of complex structures. *Colloquium Mathematicum* **XLVIII**. p 1–5.

Gries, D. [1979a]. Current ideas in programming methodology. *Lecture Notes in Computer Science 69: Program Construction.* (G. Goos and J. Hartmanis (eds)). Springer-Verlag: Berlin-Heidelberg-New York. p 75–93.

Gries, D. [1979b]. Basic axiomatic definitions. *Lecture Notes in Computer Science 69: Program Construction.* (G. Goos and J. Hartmanis (eds)). Springer-Verlag: Berlin-Heidelberg-New York. p 94–99.

Gries, D. [1981]. *The Science of Programming.* Springer-Verlag: New York.

Guerreiro, P. [1980]. A relational model for nondeterministic programs and predicate transformers. In *Proceedings of the 4th International Colloquium on Automata, Languages and Programming, Paris, Lecture Notes in Computer Science 83.* Springer-Verlag: Berlin-Heidelberg-New York. p 136–146.

Guerreiro, P. [1981]*. Semantique relationnelle des programmes non-deterministes et des processus communicants. *Theses de 3eme Cycle.* University of Grenoble I.

Guerreiro, P. [1982]. Another characterization of weakest preconditions. *Lecture Notes in Computer Science 137.* Springer-Verlag: Berlin-Heidelberg-New York. p 164–177.

Hansoul, G. [1983]. A duality for Boolean algebras with operators. *Algebra Universalis* **17**. p 37–49.

Harel, D. [1979a]. First order dynamic logic. *Lecture Notes in Computer Science 68.* Springer-Verlag: Berlin-Heildeberg-New York.

Harel, D. [1979b]*. On the total correctness of nondeterministic programs. *IBM Research Report.* **RC7691**.

Harel, D. and V. Pratt. [1978]*. Nondeterminism in logics of programs. (preliminary report). In *Proceedings of the 5th ACM Symposium on the Principles of Programming Languages.* Tuscon, Arizona. ACM: New York. p 203–213.

Hehner, E.C.R. [1984]. Predicative programming, Part I. *Communication of the ACM* **27**. p 134–143.

Hesselink, W.H. [1990]. Command algebras, recursion and program transformation. *Formal Aspects of Computing* **2**. p 60–104.

Hindley, J.R. and J.P. Seldin. [1986]. *Introduction to Combinatorics and Lambda Calculus.* Cambridge University Press.

Hitchcock, P. and D.M.R. Park. [1973]*. Induction rules and termination proofs. In *Proceedings of the 1st Colloquium on Automata, Languages and Programming.* (M. Nivat (ed)). North Holland. p 225–251.

Hoare, C.A.R. [1969]. An axiomatic basis for computer programming. *Communications of the ACM* **12** (10). p 576–583.

Hoare, C.A.R. and P.E. Lauer. [1974]. Consistent and complementary formal theories of the semantics of programming languages. *Acta Informatica* **3**. p 135–153.

Hoare, C.A.R. [1978]. Some properties of predicate transformers. *Journal of the ACM* **25** (3). p 461–480.

Hoare, C.A.R. and He Jifeng. [1987]. The weakest prespecification. *Information Processing Letters* **24** (2). North-Holland. p 127–132.

Holt, R.C. [1991]. Healthiness versus realizability in predicate transformers. Manuscript. University of Toronto. August (1989). (Revised April 1990). (Submitted to *Journal of Formal Aspects of Computing* (1991)).

D. Jacobs and D. Gries. [1985]. General correctness: A unification of partial and total correctness. *Acta Informatica* **22**. p 67–83.

Jones, C.B. [1986]*. *Systematic Software Development Using VDM.* Prentice-Hall International.

B. Jónsson and A. Tarski. [1951]. Boolean algebras with operators I. *American Journal of Mathematics* **73**. p 891–939.

B. Jónsson and A. Tarski. [1952]. Boolean algebras with operators II. *American Journal of Mathematics* **74**. p 127–162.

Kleene, S.C. [1956]*. Representation of events in nerve nets and finite automata. In *Automata studies* **34**. (C.E. Shannon and J. McCarthy (eds)). Princeton University Press. p 3–41.

Kozen, D. [1980]. A representation theorem for models of $*$-free PDL. In *Lecture Notes in Computer Science 85: Automata, Languages and Programming.* (J. de Bakker and J. van Leeuwen (eds)). Springer-Verlag: Berlin-Heidelberg-New York. p 351–362.

Lampson, B.W., J.J. Horning, R.L. London, J.G. Mitchell and G.J. Popek. [1977]*. Report on the programming language Euclid. *SIGPLAN Notices* **12** (2).

Main, M.G. [1987]. A powerdomain primer. *Bulletin of the EATCS* **33**. p 115–147.

Majster-Cederbaum, M.E. [1980]. A simple relation between relational and predicate transformer semantics for nondeterministic programs. *Information Processing Letters* **4**. p 190–192.

Manes, E.G. and M.A. Arbib. [1986]. *Algebraic Approaches to Program Semantics.* Springer-Verlag: Berlin-Heidelberg-New York.

Manna, Z. [1969]*. The correctness of programs. *Journal of Computer and System Sciences* **3**. p 119–127.

Manna, Z. and A. Pnueli. [1974]. Axiomatic approach to total correctness of programs. *Acta Informatica* **3**. p 243–263.

McCarthy, J. [1963]*. A basis for a mathematical theory of computation. In *Computer Programming and Formal Systems.* (P. Braffort and D. Hirschberg (ed)). North-Holland. p 33–69.

Milner, R. [1973]*. Processes: a mathematical model for computational agents. In *Logic Colloquium '73.* North-Holland: Amsterdam.

Morgan, C. [1990]. *Programming from Specifications.* Prentice-Hall: Englewood Cliffs.

Naur, P. [1966]*. Proof of algorithms by general snapshots. *BIT* **6**. p 310–316.

Nelson, G. [1989]. A generalization of Dijkstra's calculus *ACM Transactions of Programming Languages and Systems* **11** (4). p 517–561.

Norcliffe, A. and G. Slater. [1991]. *Mathematics of Software Construction.* Ellis Horwood: London.

Parnas, D.L. [1985]*. Technical correspondence. *Communications of the ACM* **28** (5). p 534–536.

Peirce, C.S. [1870]*. Description of a notation for the logic of relatives, resulting from an amplification of the conceptions of Boole's calculus of logic. *Memoirs American Acad* **9**. p 317–378.

Plotkin, G.D. [1976]. A powerdomain construction. *SIAM Journal of Computing* **5**. p 452–487.

Pratt, V.R. [1976]. Semantical considerations on Floyd-Hoare logic. In *Proceedings of the 17th IEEE Symposium on Foundation of Computer Science.* p 109–121.

Pratt, V.R. [1979a]. Dynamic logic. In *Foundations of Computer Science III, part 2.* (J.W. de Bakker and J. van Leeuwen (eds)). p 53 –82.

Pratt, V.R. [1979b]. Process logic. In *Proceedings of the 6th Annual ACM Symposium on Principles of Programming Languages.* p 93–100.

Pratt, V.R. [1990]. Action logic and pure induction. (To appear in *Jelia-90*).

Sanderson, J.G. [1981]. A Relational Theory of Computing. *Lecture Notes in Computer Science 82.* (G. Goos and J. Hartmanis (eds)). Springer-Verlag: Berlin-Heidelberg-New York.

Schmidt, D. [1986]. *Denotational Semantics: A Methodology for Language Development.* Allyn and Bacon: Boston.

Schmidt, G. and T. Ströhlein. [1985]. Relation Algebras: Concepts of point and representability. *Discrete Mathematics* **54**. p 83–92.

Smullyan, R.M. [1968]. *First-Order Logic.* Springer-Verlag: Berlin.

Smyth, M.B. [1978]. Power domains. *Journal of Computer and System Sciences* **16**. p 23–36.

Spivey, J.M. [1989]. *The Z notation: A reference manual.* Prentice Hall: Hemel Hempstead.

Stone, M.H. [1936]. The theory of representation for Boolean Algebras. *Transactions of the American Mathematical Society* **40**. p 37–111.

Tarski, A. [1941]. On the calculus of relations. *Journal of Symbolic Logic* **6**. p 73–89.

Wand, M. [1977]. A characterization of weakest preconditions. *Journal of Computer and System Sciences* **15**. p 209–212.