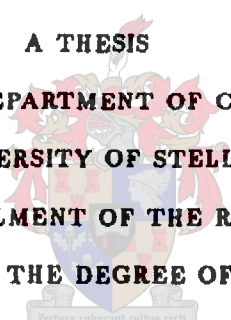


# Providing Mechanical Support for Program Development in a Weakest Precondition Calculus

A THESIS  
SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE  
OF THE UNIVERSITY OF STELLENBOSCH  
IN PARTIAL FULFILMENT OF THE REQUIREMENTS  
FOR THE DEGREE OF  
MASTER OF SCIENCE

The crest of the University of Stellenbosch is centered behind the text. It features a shield with various symbols, including a book and a torch, topped with a crown and flanked by two figures. A banner is at the base.

By  
Charlotte Christene Ackerman  
April 1993

Supervised by: Mr P.J.A. de Villiers

# Declaration

**I the undersigned hereby declare that the work contained in this thesis is my own original work and has not previously in its entirety or in part been submitted at any university for a degree.**

**Signature:**

**Date:**

# Abstract

Formal methods aim to apply the rigour of mathematical logic to the problem of guaranteeing that the behaviour of (critical) software conforms to predetermined requirements. The application of formal methods during program construction centers around a formal specification of the required behaviour of the program. A development attempt is successful if the resulting program can be formally proven to conform to its specification. For any substantial program, this entails a great deal of effort. Thus, some research efforts have been directed at providing mechanical support for the application of formal methods to software development.

E.W. Dijkstra's calculus of weakest precondition predicate transformers [39, 38] represents one of the first attempts to use program correctness requirements to guide program development in a formal manner. The calculus provides a set of rules that may be used to "discover" programs by considering the formal specifications that they have to fulfill. Programs are expressed in a powerful nondeterministic mini language, that can be seen as a minimal subset of modern imperative programming languages.

In this thesis we examine the suitability of Dijkstra's calculus for mechanically supported application as well as what kind of mechanical tools will best support its use. For this purpose, a small prototype implementation was undertaken. It is found that formalizing Dijkstra's calculus for purposes of mechanical support has an unfortunate side effect, in that limitations are placed on the notation and strategies that a program developer may employ. It also causes additional complexity in the logical formulae to be manipulated. Abstraction mechanisms provide a powerful tool to combat these problems. In the second part of this thesis, we consider a series of abstraction mechanisms that may be incorporated into the calculus for this purpose, terminating in a generalization of Dijkstra's calculus such that

**program development by formal stepwise refinement is supported.**

# Opsomming

Formele metodes poog om die strengheid van wiskundige logika te gebruik om te waarborg dat die gedrag van (kritiese) programmatuur voldoen aan gegewe vereistes. Die toepassing van formele metodes tydens programontwikkeling sentreer rondom a formele spesifikasie van die verlangde programgedrag. 'n Ontwikkelingspoging is suksesvol as daar formeel bewys kan word dat die resulterende program aan sy spesifikasie voldoen. Vir enige substansiële program, verteenwoordig dit 'n aansienlike hoeveelheid werk. Verskeie navorsingspogings is gerig op die daarstelling van meganiese ondersteuning vir die gebruik van formele metodes tydens ontwikkeling van sagteware.

E.W. Dijkstra se calculus van swakste voorkondisie (“weakest precondition”) predikaattransformators [39, 38] is een van die eerste pogings om vereistes vir programkorrektheid op 'n formele en konstruktiewe wyse tydens programontwikkeling te gebruik. Die calculus verskaf 'n aantal reëls wat gebruik kan word om programme te “ontdek” deur beskouing van die formele spesifikasies waaraan hul moet voldoen. Programme word uitgedruk in 'n klein, kragtige, nie-deterministiese taal, wat gesien kan word as 'n minimale subversameling van moderne imperatiewe programmeertale.

In hierdie tesis word die geskiktheid van Dijkstra se calculus vir meganies ondersteunde toepassing ondersoek, sowel as watter vorm van meganiese ondersteuning die beste sal wees. Hiervoor is 'n klein prototipe implementasie onderneem. Daar word bevind dat formalisering van Dijkstra se calculus ten einde meganiese ondersteuning moontlik te maak, die ongelukkige newe-effek het dat beperkings geplaas word op die notasie en strategieë wat 'n programontwikkelaar mag gebruik. Dit veroorsaak ook addisionele kompleksiteit in die logiese formules wat gemanipuleer moet word. Abstraksie meganismes verskaf 'n kragtige werktuig om hierdie

probleme te bekamp. In die tweede gedeelte van hierdie tesis, word 'n reeks abstraksie meganismes wat in die calculus opgeneem kan word vir hierdie doel, beskou. Dit word afgesluit met 'n beskrywing van 'n veralgemening van Dijkstra se calculus wat programontwikkeling deur formele stapsgewyse verfyning ondersteun.

# Acknowledgements

**My thanks to**

- **my supervisor, Pieter de Villiers for his guidance and advice;**
- **my family and friends for their diligent support**
- **Marius Ackerman for his unfailing interest in and enthusiasm about the use of formal methods, his helpful discussions on various topics at all hours, his financial and moral support and his faith in me.**

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
<b>I</b>	<b>Perspectives from a Prototype Implementation</b>	<b>12</b>
<b>2</b>	<b>A Semantic Framework</b>	<b>13</b>
2.1	Preliminary Notation . . . . .	14
2.2	State of a Computation . . . . .	15
2.3	Nondeterministic Mechanisms . . . . .	15
2.4	State Transformations . . . . .	15
2.5	Predicates and Predicate Transformers . . . . .	17
2.6	Weakest Preconditions . . . . .	19
2.7	Weakest Liberal Preconditions . . . . .	21
<b>3</b>	<b>Dijkstra's Programming Calculus</b>	<b>23</b>
3.1	Program Constructs of a Guarded Command Language . . . . .	24
3.2	Data Types . . . . .	34
3.3	The Scope of Identifier Declarations . . . . .	38

<b>3.4</b>	<b>Specifications and Annotations . . . . .</b>	<b>40</b>
<b>3.5</b>	<b>Developing Guarded Command Programs . . . . .</b>	<b>43</b>
<b>3.5.1</b>	<b>An Example . . . . .</b>	<b>45</b>
	Problem Specification and Data Declarations . . . . .	46
	Constructing a Solution . . . . .	47
<b>4</b>	<b>Mechanical Support for Dijkstra's Calculus . . . . .</b>	<b>52</b>
<b>4.1</b>	<b>Considerations for a Prototype System . . . . .</b>	<b>53</b>
<b>4.2</b>	<b>Generating (Weakest) Preconditions . . . . .</b>	<b>57</b>
<b>4.2.1</b>	<b>Formalizing the Syntax . . . . .</b>	<b>57</b>
<b>4.2.2</b>	<b>Representing and Manipulating Expressions . . . . .</b>	<b>59</b>
	Suitable data structures . . . . .	60
	Variable renaming . . . . .	61
<b>4.2.3</b>	<b>Variable Initialization . . . . .</b>	<b>63</b>
	An unsatisfactory solution . . . . .	64
	Dijkstra's regimen . . . . .	65
	Hemerik's suggestions . . . . .	66
	Final considerations . . . . .	67
	Representation of initial values in program annotations . . . . .	68
<b>4.2.4</b>	<b>A Meaningful Type System . . . . .</b>	<b>69</b>
	Handling arrays and records . . . . .	71
<b>4.2.5</b>	<b>Multiple Assignment to Compound Variables . . . . .</b>	<b>75</b>
<b>4.2.6</b>	<b>Quantified Expressions . . . . .</b>	<b>79</b>

<b>4.3</b>	<b>Proof Obligations . . . . .</b>	<b>80</b>
<b>4.3.1</b>	<b>Generating Verification Conditions . . . . .</b>	<b>80</b>
	Internal annotations . . . . .	81
<b>4.3.2</b>	<b>A Suitable Specification Language . . . . .</b>	<b>84</b>
	Dealing with undefinedness . . . . .	88
	Concluding remarks on the specification language . . . . .	94
<b>4.3.3</b>	<b>Perspectives on Program Development Support . . . . .</b>	<b>95</b>
	Initial requirements . . . . .	95
	Development strategies . . . . .	96
	Final considerations . . . . .	98
<b>4.3.4</b>	<b>Perspectives on Proof Support . . . . .</b>	<b>100</b>
	General . . . . .	100
	Simplification of logical formulae . . . . .	103
<b>4.4</b>	<b>Conclusion . . . . .</b>	<b>105</b>
<b>II</b>	<b>Providing for the Formal Use of Abstraction</b>	<b>107</b>
<b>5</b>	<b>Abstraction Mechanisms for Dijkstra's Calculus</b>	<b>108</b>
<b>5.1</b>	<b>Procedural Abstraction . . . . .</b>	<b>109</b>
<b>5.1.1</b>	<b>Local Blocks Revisited . . . . .</b>	<b>109</b>
<b>5.1.2</b>	<b>Parameterized Statements . . . . .</b>	<b>111</b>
	Discussion of assumptions and restrictions . . . . .	112
<b>5.1.3</b>	<b>A Procedure Construct . . . . .</b>	<b>114</b>

Proof rules . . . . .	116
Incorporating procedures into Dijkstra's calculus . . . . .	118
5.1.4 Functions . . . . .	119
5.1.5 Recursion . . . . .	121
Semantics of recursive procedure calls . . . . .	122
Direct recursion . . . . .	122
Indirect recursion . . . . .	123
Introducing parameters . . . . .	124
Proof rules . . . . .	124
5.1.6 Procedural Refinement . . . . .	126
5.2 Data Abstraction . . . . .	127
5.2.1 Hoare's Method . . . . .	128
Concepts . . . . .	128
Correctness proofs . . . . .	129
Discussion . . . . .	131
5.2.2 Gries-Prins Modules . . . . .	133
Developing correct modules . . . . .	135
5.2.3 Data Refinement . . . . .	138
5.3 Abstraction in specifications . . . . .	138
5.4 Conclusion . . . . .	141
<b>6 An Extended Semantic Framework</b>	<b>143</b>
6.1 A Lattice of Predicate Transformers . . . . .	143

6.2	Healthiness Conditions Reconsidered . . . . .	144
6.3	A Relation of Refinement . . . . .	146
7	A Program Refinement Calculus	147
7.1	Abstract Programs . . . . .	148
7.1.1	Language Extensions . . . . .	149
	A specification statement . . . . .	149
	Local blocks . . . . .	151
	Program conjunction . . . . .	152
	Procedures, parameters, and modules . . . . .	152
	Iteration and recursion . . . . .	154
7.1.2	Data Types . . . . .	155
7.2	Formalized Stepwise Refinement . . . . .	155
7.2.1	Procedural (Algorithmic) Refinement . . . . .	155
	Top-down development . . . . .	156
	Refinement laws . . . . .	157
7.2.2	Data Refinement . . . . .	159
7.2.3	Types and Invariants . . . . .	161
7.3	Some Perspectives on Mechanical Support . . . . .	164
8	Conclusion	166
A	Basic Concepts, Notation and Terminology	168
A.1	Relations . . . . .	168

<b>A.2 Orderings . . . . .</b>	<b>170</b>
<b>A.3 Lattices . . . . .</b>	<b>171</b>
<b>A.4 Ordinals and Cardinals . . . . .</b>	<b>172</b>
<b>A.5 Fixed Points . . . . .</b>	<b>173</b>
<b>B A CFG for Annotated Guarded Command Programs</b>	<b>174</b>
<b>C The predicate <math>\text{domain}(e)</math></b>	<b>177</b>
<b>D Substitution in Program Constructs</b>	<b>180</b>
<b>E An Annotated Program and its VCs</b>	<b>181</b>

# Chapter 1

## Introduction

Since the 1960's, the rapid increase in size and complexity of software systems has produced an epidemic of projects that exceed their deadlines and budgets and result in unreliable products.

This situation persists even today. The genre of horror stories concerning computer malfunctions based both on documented fact and fable continues to grow. A leading computer journal<sup>1</sup> has gone as far as running a regular column to record substantiated tales of this kind. In the mind of the public such incidents understandably lead to growing distrust of and concern over the use of computers in critical applications such as medical systems, defense, aviation, factories, power plants and financial institutions.

Programmers have traditionally endeavoured to ensure the quality of software through systematic testing. Since exhaustive testing is impossible for practical programs, a number of carefully chosen test cases have to be constructed to be somehow "representative" of the entire spectrum of possibilities. Since program outputs are in general not continuous functions of their inputs, making extrapolation from a sample of empirical results unreliable for predicting the outcome for all cases. These facts as well as practical experience show that program testing can never provide a sufficient level of assurance of the quality of life critical software.

This leads us to contend that the rigour of mathematical proof is the only way of substantiating claims about software correctness.

---

<sup>1</sup>ACM SIGSOFT Software Engineering Notes

The extensive use of mathematical logic in the description and prediction of the attributes of programs has become known as *formal methods*. Formal methods have been applied to various aspects of the construction of correct, reliable, and efficient software. These include:

**program specification** —the provision of a formal specification of the intended behaviour of a program;

**program verification** —demonstrating by means of a mathematical proof that a program is consistent with its (formal) specification;

**program transformation** —performing correctness preserving transformations on a specification or program for the sake of refinement or optimization;

**program synthesis** —extracting a program from a constructive proof that a result satisfying a specification exists.

In this thesis our concern is with program verification—specifically using an axiomatic method in the tradition of Floyd [51] and Hoare [84].

## Historical Perspective

### Floyd Uses Assertions to Describe Program Behaviour

The cornerstone of axiomatic methods of program verification was laid by Floyd in a seminal paper, “Assigning Meanings to Programs” [51], delivered at a conference in 1967. Floyd uses a flowchart model of programs with nodes representing program statements and arcs the flow of control. His approach comprises attaching logical assertions to the arcs of a program flowchart. An assertion must be true whenever control passes over the arc to which it is attached. Program behaviour is specified by providing both an *entry-* and an *exit-condition* for a program. The entry-condition expresses the information that can be assumed about the initial state of the program variables, while the exit-condition details the state of program variables after termination. Using these entry- and exit-conditions, corresponding *intermediate assertions* are generated to annotate the internal arcs of the flowchart.

One has to prove that the exit-condition will be satisfied by the final state attained, after execution starting from a state that satisfies the entry-condition. These proofs are built around rules, given by Floyd, to derive from an entry-condition associated with a node, and the node itself, the strongest exit-condition that can be guaranteed on leaving that node.

Floyd was also the first to use well-ordered sets to prove program termination. His technique involves the use of so-called *W-functions*. A *W-function* maps the program variables to a well-ordered set (an ordered set with no infinitely decreasing sequences). A program is shown to terminate by demonstrating for each arc that the value of the *W-function* decreases every time control passes over that arc.

### Hoare's Deductive System for Program Correctness

Hoare's 1969 paper, "An Axiomatic Basis for Computer Programming" [84], represents the next milestone in the development of axiomatic program verification methods. He proposes that statements of program properties be viewed as theorems in a deductive system. An ALGOL-like language is used as programming notation. Analogous to Floyd's entry- and exit-conditions, two logical assertions, a *precondition* and a *postcondition*, are used to specify the behaviour of program constructs. The now familiar notation

$$P\{S\}R$$

is introduced by Hoare, to mean that if assertion  $P$  is true before execution of program  $S$  begins, then the assertion  $R$  will be true on its completion. If such a fact can be formally established in Hoare's deductive system, we have a theorem, i.e.

$$\vdash P\{S\}R.$$

The axioms of Hoare's logical system comprise basic facts about "elementary" program operations such as integer arithmetic, as well as an axiom schema describing the effect of executing an assignment instruction, while inference rules are given to describe the effect of sequential composition of program statements, iteration, and other compound statements.

To prove that a program  $S$  is consistent with its specification (consisting of a precondition  $P$  and postcondition  $R$ ), one starts with axioms about the elementary program statements and applies Hoare's inference rules repeatedly until the theorem  $\vdash P\{S\}R$  is deduced.

This approach specifically excludes proofs of program termination. Manna and Pnueli later extended Hoare's work, presenting a more extensive deductive system for an ALGOL-like language that includes a treatment of program termination [113]. The notation

$$\{P\}S\{R\}$$

has been used to denote that if assertion  $P$  is true before execution of program  $S$  begins, then  $S$  will terminate and the assertion  $R$  will be true on its completion.

### Dijkstra's Programming Calculus

The foundational work of Floyd, Hoare, and others set the scene for constructive verification techniques. Experience with the application of these early verification methods quickly indicated that programs written without close guidance from their specifications, can be extremely difficult to prove consistent with such specifications. To overcome this problem, program development has to be interweaved with the construction of assertions that describe its behaviour and may be used in demonstrating program correctness.

Dijkstra responded to this challenge by showing how assertions may be used to guide the development of programs within a deductive system. In the classical reference works [39] and [38] he developed a formal discipline or calculus for the derivation of programs that meet their stated specifications. Dijkstra's calculus provides a set of rules to be used in "discovering" programs by considering the assertions they have to guarantee and allows a program and its correctness proof to be constructed hand in hand. The basic procedure centers around deciding on an assertion to be satisfied (a postcondition) and using the calculus to find a program construct (and corresponding precondition) that will guarantee its truth.

Dijkstra's main deductive tool is a set of functions called *weakest precondition predicate transformers*. These are used to formalize the semantics of program constructs and drive the program development process. In general, a predicate transformer is a function that associates

one assertion with another. In particular:

The weakest precondition of a program construct  $S$  with respect to an assertion (postcondition)  $R$ , denoted  $wp_S(R)$  (or  $wp(S, R)$ ), is that assertion (precondition) that is satisfied by all program states such that execution starting from any one of them will terminate in a state satisfying  $R$ , i.e.

$$\{P\}S\{R\}$$

is equivalent to

$$P \Rightarrow wp_S(R).$$

The fact that the weakest precondition transformer works “backwards” from a postcondition to be established, makes it extremely well suited to the goal-directed activity of program composition.

Because existing programming languages did not satisfy his requirements, Dijkstra invented a unique “guarded command” language notation for the presentation of programs (algorithms) developed in the weakest precondition calculus. Apart from primitive commands like **skip** and **assignment**, the language contains an alternative and a repetitive construct that are built up from *guarded commands*. A *guarded command* is a command (program statement) list prefixed by a Boolean expression (guard), such that the command list will only be executed if the Boolean expression evaluates to true, e.g.

$$i \neq n \rightarrow i := i + 1$$

A novel characteristic of Dijkstra’s guarded command language is that it allows the expression of *nondeterministic* programs in a natural way, e.g. in the guarded command language statement

$$\text{if } x \leq MAX \rightarrow x := x + 1 \square x \geq MAX \rightarrow x := x - 1 \text{ fi}$$

a choice of which statement to execute when  $x = MAX$ , is to be made nondeterministically. A nondeterministic mechanism is such that its initial state does not uniquely determine its ensuing activities.

The guarded command language formulated by Dijkstra is in effect a mini language that embodies the bare essentials of any imperative programming language. Nondeterminism is a powerful abstraction mechanism in the calculus, allowing one to describe in a concise and elegant manner, a whole class of algorithms instead of a single deterministic one and to separate concerns of correctness and efficiency by delaying decisions for the sake of efficient implementation to later stages of development.

## **The Anti-formal Methods Stance and Some Limitations of Formal Verification**

Formal methods have been opposed by some computer science researchers. Anti-formal methods lobbyists (e.g. [35, 49]) have presented arguments including:

- Formal program verification does not seem to “scale up”, i.e., there is little evidence to suggest that a large, complex verification may be accomplished as the sum of a number of smaller, simpler verifications.
- “No matter how high the payoff, no one will ever be able to force himself to read the incredibly long, tedious verifications of real-life systems, and, unless they can be read, understood and refined, the verifications are worthless.” [35]
- Progress from the theoretical possibility of program verification towards verified programs becoming a “reality” in the wider computing industry, has to date been embarrassingly slow.

The author agrees with the view, expressed in [49], that one has to separate algorithms from programs when issues of verification arise. The notion of an algorithm is more abstract than that of a program. The intended interpretation of an algorithm is an abstract machine for which there is not supposed to be a physical counterpart, i.e., its properties can be established by definition. On the other hand the intended interpretation of a program is an abstract machine that is supposed to have a physical machine counterpart. The properties of physical machines can only be established inductively. (See [49]) for a detailed exposition.)

Avra Cohn's discussion of the scope and limitations of the Viper microprocessor verification effort [31] includes a similar caution concerning hardware "verification":

"Ideally, one would like to prove that a chip correctly implemented its intended behaviour in all circumstances; we could then claim that the chip's behaviour was predictable and correct. In reality, neither an actual *device* nor an *intention* are objects to which logical reasoning can be applied.

:

In short, verification involves two or more *models* of a device, where the models bear an uncheckable and possibly imperfect relation both to the intended design and to the actual device. This point is not merely a philosophical quibble; ..."

Formal methods, like any other methods, have their limitations. This should not sound the death knell of continued research effort in this area, because formal methods have already made a positive contribution to the construction of reliable software:

- Formal specification methods have, and should continue to make, valuable contributions to software engineering practices.
- Research into program verification has had very "practical" spin-offs in the form of program validation techniques. These are also developments in the right direction—towards the construction of more reliable software.
- By placing a mathematical foundation under the craft of programming, formal methods provide intellectual tools that allow a firmer and more precise grip on complex subject matter. Teaching prospective programmers the rigours of formal methods, can only have a beneficial influence on the way in which they think about and construct software.

## About This Thesis

This thesis concerns one of the fundamental issues in computer science—the construction of reliable software. We have chosen to approach it via formal methods, specifically Dijkstra's weakest precondition programming calculus. In making this decision, the following

observations were decisive:

- The guarded command language, being a “minimal subset” of modern imperative languages, is easy to learn and allows a programmer to focus on correctness issues without being distracted by language idiosyncrasies, while algorithms can easily be transformed into standard programming notations after development.
- The guarded command language provides a simple and concise notation for the expression of algorithms.
- Algorithms are completely abstract entities, for which consistency with a formal specification can be formally established.
- Dijkstra’s calculus emphasizes a separation of correctness concerns from those of efficiency, allowing a useful abstraction when developing an algorithm.
- The weakest precondition calculus is specifically tailored towards the construction of correct algorithms, as opposed to *a posteriori* correctness proofs, and provides a unique “calculational approach” to algorithm development. This makes it eminently suitable for teaching program development in a formal framework.
- Dijkstra’s calculus is amenable to various extensions to experiment with more complex features such as parallelism, fairness, formalized stepwise refinement and correctness preserving program transformations.

The need for efficient program development has necessitated the use of increasingly sophisticated mechanical tools to assist programmers. Such tools have become indispensable in software production environments and have achieved undeniable success in allowing efficient editing, storage, retrieval, and general administration of programs, syntax and semantic analysis, type checking, optimization, and assistance for program testing and “debugging”. The evolution of the role of machines as programming assistants has also brought about certain negative effects, including a reliance on sophisticated debugging aids to establish confidence in unstable software, experimentation at the terminal or programming by trial and error.

Introductory texts on rigorous programming disciplines, such as [38, 66, 14, 40, 79], concentrate on the principles involved and assume no tools save pencil, paper, and intellectual

machinery. Yet the opinion has been voiced, and is shared by the author, that if formal methods are to enjoy wider usage, their application will have to be supported by mechanical tools that will relieve a programmer from at least some of the detailed, time consuming manipulations they involve.

Starting with such milestone mechanical program verification systems as those of [92] and [58], much research has been done in this area; some of which is reported in [1, 2, 3]. Publications such as [143, 70] describing implementations based on the weakest precondition paradigm are, however, quite rare and only recently have works such as [4, 80], using the language of guarded commands for the expression of algorithms, appeared.

### **Thesis Aim**

This thesis is a survey of theoretical aspects regarding the following questions:

- How suitable, if at all, is Dijkstra's weakest precondition calculus to be the basis of mechanical tools for formal program (algorithm) construction?
- What kind of mechanical tools would best suit the use of Dijkstra's calculus?

### **Thesis Outline**

The investigation is divided into two parts: Part 1 introduces Dijkstra's language of guarded commands and its accompanying weakest precondition programming calculus and describes some obstacles encountered in providing basic mechanical support for application of the calculus, as well as suggestions for overcoming them. This comprises chapters 2 through 4. Part 2 addresses one of the major obstacles to effective mechanically supported program development using Dijkstra's calculus as identified in Part 1, i.e. a lack of formal abstraction mechanisms. A survey of abstraction mechanisms that could be incorporated into the calculus is presented, and effects of their inclusion upon the calculus itself and its mechanisation are discussed. Part 2 spans chapters 5 through 8.

In Chapter 2, a semantic framework is presented as a formal frame of reference for Dijkstra's

calculus. Where necessary, the reader may reference Appendix A for some mathematical background on material in this chapter. The treatment includes the concepts of the state of a computation, the state space, state transformations, predicates, and predicate transformers; ending with a discussion of the weakest precondition and weakest liberal precondition predicate transformers. Apart from introducing notation, this chapter is intended to provide a concise, fairly formal description of the key constituents of Dijkstra's calculus, identify their key properties and lemmas on which some of the results to follow, are based. A strong theoretical basis is essential for any verification system, manual or otherwise, as a mechanical verification system without a solid theoretical foundation is of doubtful value.

Chapter 3 provides a summary of Dijkstra's weakest precondition calculus. The constructs that comprise the guarded command language are presented in turn, together with an operational description and their weakest precondition semantics. Where applicable, theorems that simplify the application of the theory in practical situations are presented. Discussions on the basic data types used in guarded command programs, scope rules for declarations and specifications and annotations of programs are also included. The last section of this chapter explains and illustrates the main strategies for algorithm development in the calculus on a programming problem. The contents of this chapter, introduces the reader to the syntax and semantics and application of the main agents that a mechanical verification system will have to manipulate.

Chapter 4 contains the investigation into the suitability of Dijkstra's calculus for mechanical support. Discussions range from the need to formalize certain aspects of the calculus such as the syntax, type system and specification language to various considerations for the calculation of preconditions and proof obligations. Some suggestions are made on proof support and support for interactive program development. When providing mechanical support, one has to sacrifice the informal part of the mixture of formal and informal reasoning and presentation that one finds in manual applications of Dijkstra's calculus. One is limited by having to be completely formal and work within the scope of rigorously delineated syntax and method. Unless the explosion of complexity in formulas to be manipulated and the excessive burden of proof obligations that this leads to can be minimized, the usefulness of a mechanical system will be severely limited. One way to combat the limiting effects of complete formality is to

provide abstraction mechanisms that will allow more powerful means of expression within the bounds of formality enforced by the system, more concise notation, more generality and the removal of excessive details from immediate consideration.

In Chapter 5 some mechanisms for abstraction are considered for inclusion in Dijkstra's calculus. Three main possibilities are explored: procedural abstraction, data abstraction, and abstraction mechanisms in the specification language. Dijkstra's calculus imposes limitations on the usage of abstraction and refinement during program development. Such limitations may be removed by extending the semantic framework of Chapter 2. This is done in Chapter 6. The extended formal foundation obtained as a result, may be used to construct a *refinement calculus* based on the weakest precondition paradigm, such as developed by R.J.R. Back [9, 8], Joseph Morris [130, 128] and Carroll Morgan [121].

Chapter 7 is a discussion of the constructs and uses of such a refinement calculus. In this calculus, specifications also become programs and executable programs are simply programs that have been refined to such an extent that no unimplementable constructs remain. Step-wise refinement is a formal manipulation in the refinement calculus, a fact that has interesting implications for mechanical support. Both procedural and data refinements are considered. Some perspectives on mechanical support for application of the refinement calculus are discussed.

Conclusions are presented in Chapter 8.

## **Part I**

# **Perspectives from a Prototype Implementation**

## Chapter 2

# A Semantic Framework

What is your substance, whereof are you made,  
That millions of strange shadows on you tend?  
Since every one hath, every one, one shade,  
And you, but one, can every shadow lend.

*William Shakespeare, Sonnet 53*

A theoretical foundation is important for any formal method. The aim of the semantic framework for Dijkstra's weakest precondition calculus presented here, is to provide a suitable level of formality for discussions to follow in Chapters 3, 6 and 7.

The well known concepts of the state of a computation and the state space are defined, followed by a characterization of boundedly nondeterministic mechanisms as state transformation functions. Predicates and predicate transformers are introduced, specifically weakest precondition and weakest liberal precondition predicate transformers and the relationship between state transformations and weakest precondition transformers are highlighted. These predicate transformers provide a means of defining programming language semantics in a way that supports systematic program development from formal specifications.

The lattice theoretical framework adopted here has been used by a number of authors, since it allows an elegant treatment of predicates and predicate transformers and is particularly helpful when defining the semantics of iterative and recursive mechanisms. (The reader may

also refer to [41] for a more thorough and formal introduction to the theory of predicate transformer semantics than that presented here.) In Chapter 6 this framework is extended to accommodate the more abstract mechanisms introduced in Chapter 7. Appendix A should be referenced, where necessary, for background material on relations, orderings and lattices as employed here.

## 2.1 Preliminary Notation

### The Form of Arguments

Following [41, Chapter 4], a formal argument to show  $R \equiv Q$  (similarly  $R \Rightarrow Q$ ) for predicates  $R$  and  $Q$ , will be written in the following concise and calculational format

$$\begin{array}{l}
 R \\
 \equiv \quad \{\text{hint why } R \equiv X\} \\
 X \\
 \equiv \quad \{\text{hint why } X \equiv Q\} \\
 Q
 \end{array}$$

### Quantified Expressions

We use the notation (*operation dummies* : *range* : *term*), to denote the combination of the values assumed by the *term*, according to the *operation*, as the *dummies* vary over the given *range*. Unless the type of a dummy variable is obvious from the context, it is given as part of the range, e.g.

$$(\forall x, y : x \in \mathcal{N} \wedge y \in \mathcal{N} : x > y \Rightarrow x - y > 0).$$

The range itself is omitted if it is obvious from the context, e.g.

$$(\forall x, y :: (\neg x \vee y) \equiv (x \Rightarrow y)).$$

## 2.2 State of a Computation

The state of a computation is a mapping associating identifiers with values and may be interpreted as reflecting the values of all program variables at a given instant during program execution.

Let  $C_{Var}$  denote a computation involving a countable set of program variables,  $Var$ , taking their values from a set of values,  $Val$ . A *state of  $Var$*  [8] is a function

$$\sigma : Var \mapsto Val$$

The *state space of  $Var$*  is the set of all states of  $Var$ ; denoted  $\Sigma_{Var}$ . In general  $\Sigma_{Var}$  may be thought of as the cartesian product of the relevant data domains.

## 2.3 Nondeterministic Mechanisms

We distinguish between deterministic and nondeterministic computation mechanisms. Upon activation, the activity of a *deterministic mechanism* is fully determined by its initial state, i.e. its behaviour is fully reproducible. For a *nondeterministic mechanism*, activation in a particular initial state will result in one out of a class of possible activities. Nondeterminism is said to be *bounded* if a finite number of alternatives for continuing computation are available at any moment.

## 2.4 State Transformations

The effect of activating a (nondeterministic) mechanism under an initial value assignment to its variables, can be described by means of a function between an initial state and a nonempty set of possible final states.

Let  $\perp$  denote a special *undefined state* (used to represent nontermination) and  $\Sigma_{Var}^\perp$  the state space  $\Sigma_{Var}$  extended with this state, i.e.  $\Sigma_{Var} \cup \{\perp\}$ . We now define a (*boundedly nondeterministic*) *state transformation on  $Var$*  [8, 140, 149] as a function

$$t : \Sigma_{Var} \mapsto \{X \subseteq \Sigma_{Var}^{\perp} : X = \Sigma_{Var}^{\perp} \text{ or } (X \neq \emptyset \text{ and } X \subseteq \Sigma_{Var} \text{ and } X \text{ is finite})\}$$

The meaning of a mechanism  $S_{Var}$ , involving only variables from the set  $Var$ , is given by a state transformation  $t_S$  on  $Var$ , describing the effect of activating  $S$  in an environment containing the program variables in  $Var$ . For an initial value assignment  $\sigma$  to  $Var$ ,  $t_S(\sigma)$  represents the set of possible final assignments to these variables, when  $S$  has terminated. The presence of the undefined state  $\perp$  in  $t_S(\sigma)$  indicates the possibility of nontermination of  $S$  for this initial state. Requiring  $S$  to be boundedly nondeterministic, dictates that for each initial state  $\sigma$ , either  $t_S(\sigma)$  is finite (and nonempty) or  $\perp \in t_S(\sigma)$ , see [141, page 455]. Our definition of  $t_S$  identifies all sets containing  $\perp$  with each other in the above definition—equating them all to  $\Sigma_{Var}^{\perp}$ . This renders the possibility of nontermination indistinguishable from the guarantee of nontermination and is done for the sake of conformance to Dijkstra’s weakest precondition semantics (see [140, page 541–542] and [93, page 70]). Similar approaches that offer different views on this issue may be found in [141, 34, 71, 93, 155].

In Nelson’s taxonomy of programming semantics calculi [133], the semantic model outlined above is called the *total correctness model*. This model allows an initial state to be related to proper (terminating) outcomes as well as the “looping” outcome, without distinguishing programs that may not terminate from those that definitely will not. Each initial state is related to at least one final state—indicating that “something” must happen when a mechanism is activated in an initial state (see [93, 133] for descriptions of more general models). Although no provision is made for distinguishing between run-time errors and infinite looping behaviour as causes of nontermination, the state space may be extended suitably if this is desired. As indicated in [133], various technical approaches can be applied to such a semantic model, e.g. relations on states, Hoare logic, predicates or predicate transformers. Dijkstra’s calculus of guarded commands [38] approaches the total correctness model via predicate transformers.

## 2.5 Predicates and Predicate Transformers

A predicate transformer is a function that associates a predicate on the state space with another predicate on the state space. The predicate transformer approach to program semantics is based on the fact that a set of states of a computation can be characterized by a logical formula in terms of program variables, and vice versa.

Using  $Bool$  to denote the set of truth values,  $Bool = \{true, false\}$ , we consider the poset  $(Bool, \overset{Bool}{\sqsubseteq})$  as obtained by partially ordering  $Bool$  as follows

$$x \overset{Bool}{\sqsubseteq} y \text{ iff } x \Rightarrow y.$$

Now  $(Bool, \overset{Bool}{\sqsubseteq})$  is a complete lattice with top element  $true$  and bottom element  $false$  (see Appendix A.3).

A predicate on  $Var$  can be seen as an assignment of a truth value to every state in the state space,  $\Sigma_{Var}$ . Thus a predicate on  $Var$  is a function

$$R : \Sigma_{Var} \mapsto Bool$$

A state  $\sigma$  of  $Var$  is said to satisfy a predicate  $R_{Var}$  iff  $R(\sigma) = true$ . We sometimes characterize a set of states of  $Var$  by a predicate  $R_{Var}$ , viz. the set of all states of  $Var$  that satisfy  $R_{Var}$ ; denoted  $s_R (\subseteq \Sigma_{Var})$ . The converse also applies.

Notationally, we do not distinguish between the syntactic (predicate formulas) and semantic (truth-valued functions) nature of predicates, see [12, page 605]. In the sequel we take the liberty of referring to predicates also as “assertions” or “conditions”.

## Notation

$\mathcal{F}$  denotes “false”; the predicate that is not satisfied by any state, i.e.  
 $s_{\mathcal{F}} = \emptyset$ ;

$\mathcal{T}$  denotes “true”; the predicate that is satisfied by every state, i.e.  
 $s_{\mathcal{T}} = \Sigma_{Var}$ .

$R_{e1, e2, \dots}^{x1, x2, \dots}$  denotes the predicate obtained by simultaneously replacing all free occurrences of the variables  $x1, x2, \dots$  in the predicate  $R$  by expressions  $e1, e2, \dots$  respectively. Such a substitution is only valid if it results in a syntactically well formed predicate. Renaming of bound variables in  $R$  may be necessary before the substitution commences, to prevent variable occurrences in the expressions  $e_i$  from becoming bound. This syntactic manipulation is known as *textual substitution*.

$[R]$  is used to denote a predicate  $R_{Var}$ , universally quantified over all its free variables;

$\equiv$  denotes equality between predicates, so that

$$[R \equiv Q]$$

means that  $R$  and  $Q$  are equal as functions;

$\Rightarrow$  denotes the strength order between predicates, so that

$$[R \Rightarrow Q]$$

is read “ $R$  implies  $Q$ ”, “ $R$  is at least as strong as  $Q$ ” or “ $Q$  is at least as weak as  $R$ ”.

We denote the set of all predicates on  $Var$  by  $Pred_{Var}$ . By results from Appendix A.3,  $(Pred_{Var}, \sqsubseteq)$  is a complete lattice for the pointwise extended partial order on  $Bool$ , i.e. for predicates  $R_{Var}$  and  $Q_{Var}$ :

$$R \sqsubseteq Q \equiv (\forall \sigma : \sigma \in \Sigma_{Var} : R(\sigma) \stackrel{Bool}{\sqsubseteq} Q(\sigma)).$$

The top element of  $Pred_{Var}$  is the predicate  $\mathcal{T}$  which assigns the value *true* to every state, while the bottom element is the predicate  $\mathcal{F}$  which assigns the value *false* to every state.

The following two lemmas from [130] show that the partial order on predicates corresponds to logical implication and that the least upper bound of a set of predicates corresponds to their disjunction:

**Lemma 2.5.0.1** *For  $P, Q \in Pred_{Var}$ :*

$$P \sqsubseteq Q \equiv [P \Rightarrow Q]$$

**Lemma 2.5.0.2** *For any set of predicates  $X = \{R_i : i \in I\} \subseteq Pred_{Var}$ :*

$$\bigsqcup X \equiv (\exists i : i \in I : R_i)$$

## 2.6 Weakest Preconditions

The goal-directed nature of programming, suggests a prominent role for a predicate  $R$ , called a *postcondition*, embodying the desired condition on the final state reached after activation of some mechanism  $S$ . In reality such a postcondition could be satisfied by a number of states—accommodating nondeterministic behaviour of  $S$ . A corresponding *precondition* is used to characterize those initial states for which activation of  $S$  is guaranteed to terminate in a final state satisfying  $R$ .

Let  $R_{Var}$  denote a postcondition to be satisfied upon termination of a mechanism  $S_{Var}$ . The *weakest precondition* corresponding to  $R$  [38], denoted  $wp(S, R)$ , is a predicate that characterizes the set of all initial states such that activation of  $S_{Var}$  is guaranteed to terminate in a state satisfying  $R_{Var}$ . In terms of a state transformation [8]:

$$[(\forall \sigma : \sigma \in \Sigma_{Var} : wp(S, R)(\sigma) \equiv t_S(\sigma) \subseteq s_R)]$$

Note that  $\perp \notin s_R$ , so  $t_S(\sigma)$  is not allowed to contain  $\perp$  either. This guarantees termination of  $S$  for the initial state  $\sigma$ . Requiring  $t_S(\sigma)$  to be a subset of  $s_R$ , refers to Dijkstra's “demonic”

(or “blind”) interpretation of nondeterminism [38, 93, 133], in which the “worst” execution path is chosen. A demonic implementation of nondeterminism may choose any execution path, since an initial state  $\sigma$  is considered suitable ( $\sigma \in s_{wp(S,R)}$ ) iff execution is guaranteed to reach a final state in  $s_R$ . This is in contrast with the “angelic”(or “clairvoyant”) interpretation of nondeterminism [93, 133], where the “best” execution path is chosen. Here an initial state is suitable iff at least one execution path leads to a state in  $s_R$ . Such an implementation employs backtracking or parallel evaluation.

The semantics of a mechanism  $S_{Var}$  involving only variables from the set  $Var$ , may be given by a *predicate transformer on  $Var$* , i.e. a function

$$p_S : Pred_{Var} \mapsto Pred_{Var}$$

The weakest precondition predicate transformers  $wp_S$ , such that

$$[(\forall R : R \in Pred_{Var} : wp_S(R) \equiv wp(S, R))]$$

are useful for this purpose ([140] may be referenced for a detailed exposition of the relation between the predicate transformer and state transformation approaches to programming language semantics).

Let  $S_{Var}$  be an arbitrary mechanism and  $Q_{Var}, R_{Var}$  arbitrary predicates on  $Var$ . Dijkstra identified the following “healthiness” properties that hold for weakest precondition predicate transformers of executable mechanisms [38]:

**Law of the Excluded Miracle (Strictness):**  $[wp_S(\mathcal{F}) \equiv \mathcal{F}]$

**Monotonicity:**  $[(Q \Rightarrow R) \Rightarrow (wp_S(Q) \Rightarrow wp_S(R))]$

**Conjunctivity:**  $[wp_S(Q) \wedge wp_S(R) \equiv wp_S(Q \wedge R)]$

**Disjunctivity:**  $[wp_S(Q) \vee wp_S(R) \Rightarrow wp_S(Q \vee R)]$

**Or-Continuity:**  $[wp_S(\exists n : n \in \mathcal{N} : R_n) \equiv (\exists n : n \in \mathcal{N} : wp_S(R_n))]$  for any weakening sequence of predicates, i.e. such that  $(\forall n : n \in \mathcal{N} : [R_n \Rightarrow R_{n+1}])$

The technical effect of the Law of the Excluded Miracle is to exclude mechanisms that are *partial*, i.e. that correspond to state transformations that map some initial state to the empty set (of outcomes), while the or-continuity restriction excludes mechanisms that exhibit unbounded nondeterminism.

In [87], Hoare investigates various “weakest precondition” predicate transformers in the context of Dijkstra’s healthiness criteria. He shows that Dijkstra’s  $wp_S$  function is not the *weakest* healthy one, but the best for practical programming, since it does not require clairvoyant (angelic) implementations (see above).

## 2.7 Weakest Liberal Preconditions

For a deterministic mechanism  $S$ , knowledge of how its associated predicate transformer  $wp_S$  acts on any predicate  $R$ , completely determines its possible behaviour. The complete characterization of a nondeterministic mechanism requires more. For that purpose the notion of the *weakest liberal precondition* corresponding to a postcondition  $R$ , is employed. Denoted  $wlp(S, R)$ , this condition characterizes the set of all initial states such that activation of  $S_{Ver}$ , if it terminates, will reach a final state satisfying  $R_{Ver}$ . The relationship between the weakest liberal precondition and the weakest precondition can be stated as

$$[(\forall R : R \in Pred_{Ver} : wp(S, R) \equiv wlp(S, R) \wedge wp(S, T))]$$

As above, the weakest liberal precondition predicate transformers, denoted  $wlp_S$ , are functions on  $Pred_{Ver}$  such that

$$[(\forall R : R \in Pred_{Ver} : wlp_S \equiv wlp(S, R))]$$

For any mechanism  $S$  and a postcondition  $R$ , the possible behaviour of  $S$  can be completely described by observing that every initial state of  $S$  falls into exactly one of the (disjoint) sets characterized by the following predicates [38]:

- $wp_S(R) (\equiv (wlp_S(R) \wedge wp_S(T)))$

Activation of  $S$  will establish the truth of  $R$ .

- $wp_S(\neg R) \ (\equiv (wlp_S(\neg R) \wedge wp_S(T)))$

Activation of  $S$  will establish the truth of  $\neg R$ .

- $wlp_S(\mathcal{F}) \ (\equiv (wlp_S(R) \wedge wlp_S(\neg R)))$

Activation of  $S$  will fail to produce a properly terminating activity.

- $wp_S(T) \wedge \neg wlp_S(R) \wedge \neg wlp_S(\neg R)$

Activation of  $S$  will produce a terminating activity, but the initial state does not determine whether the final state will satisfy  $R$ .

- $wlp_S(R) \wedge \neg wp_S(T)$

If activation of  $S$  produces a final state, that state will satisfy  $R$ , but whether termination takes place or not, is not determined by the initial state.

- $wlp_S(\neg R) \wedge \neg wp_S(T)$

If activation of  $S$  produces a final state, that state will not satisfy  $R$ , but whether termination takes place or not, is not determined by the initial state.

- $\neg(wlp_S(R) \vee wlp_S(\neg R) \vee wp_S(T))$

The initial state does not determine whether activation of  $S$  will produce a terminating activity, nor whether  $R$  will be satisfied in the case of termination.

From the definition of  $wlp_S$ , the following properties follow [38]:

- $[wlp_S(T) \equiv \mathcal{T}]$ ;
- $[(wlp_S(\mathcal{F}) \wedge wp_S(T)) \equiv \mathcal{F}]$

Weakest liberal preconditions are considered more closely in [131], where it is shown that they may be defined in several ways. They are also used in characterizing the property of program robustness.

## Chapter 3

# Dijkstra's Programming Calculus

In Chapter 1 the reader was informally introduced to Dijkstra's programming calculus, that provides rules, based on weakest precondition predicate transformers, for deriving a program (algorithm) from a postcondition that states its desired outcome. Having provided a formal framework in the previous chapter, we now set the scene for an investigation into the suitability of Dijkstra's calculus for mechanical support, by providing a concise overview of the syntax, semantics and typical usage of its key constituents: program constructs, data types and specifications.

Dijkstra introduced the main components of his weakest precondition calculus in [39]. This was followed by a full presentation in [38]. Since then it has been used, with various extensions and alterations, in numerous reference works including the following popular books on programming methodology, [14, 40, 43, 66, 121]. The material presented here, is based, with a few exceptions, on [38] and [66].

The program constructs of Dijkstra's guarded command language are presented first. Weakest precondition predicate transformers are used to characterize their semantics. In the case of the alternative and repetitive constructs, theorems that give guidance for their correct use in practical situations, are also stated.

The data types that occur most frequently in program developments in [38] and [66] are integers, Booleans, arrays and records. The use of these types in the context of Dijkstra's

calculus is discussed in section 3.2. This is followed by a description of the scope rules that apply to guarded command programs.

Program development in Dijkstra's calculus is guided by a functional specification in the form of two first order assertions—a precondition and a postcondition. In addition, internal program annotations such as loop invariants, bound functions and assertions to assist in the construction of correctness proofs, usually appear in guarded command programs. We state some considerations for a language suitable for the expression of such program specifications and internal assertions.

To consolidate the presentation, this chapter is concluded with a discussion and illustration of typical strategies used during program development based on Dijkstra's calculus.

### **3.1 Program Constructs of a Guarded Command Language**

A most important, but also a most elusive, aspect of any tool is its influence on the habits of those who train themselves in its use. If the tool is a programming language, this influence is—whether we like it or not—an influence on our thinking habits.

*Edsger W. Dijkstra, Preface to [38]*

In [38, Chapter 1] Dijkstra expounds the advantages of using a formal notation for the description of algorithms. An algorithm allows compact arguments about characteristics of the mechanism it describes and the choice of the guarded command language as a vehicle for the presentation of algorithms, reflects the desire to retain this compactness.

The feature from which the language derives its name—guarded commands—are not program statements in their own right, but are used as building blocks of both the alternative and repetitive constructs. Within these constructs, the use of guarded commands allow for the introduction of nondeterminism. The guarded command language is very small; apart from the primitive commands **abort**, **skip** and **assignment**, programs can be constructed using sequential composition, conditional composition (alternative construct), iteration and

encapsulation (local blocks). In the presentation to follow, the semantics of each construct is formulated in terms of weakest precondition predicate transformers, but an operational description is also supplied. Their syntax is stated informally; a formal description in the form of a context free grammar may be found in Appendix B. In the case of the alternative and repetitive constructs, program development typically involves use of a precondition that is not the weakest precondition itself, but one that is at least as strong. Such a precondition may provide more insight into the program under development and avoid a particularly lengthy and involved weakest precondition calculation. For these cases, theorems from the literature are given, to test the suitability of a potential precondition without calculating the weakest precondition itself.

### 1. No-Op

**Syntax:** `skip`

**Operational Semantics:** Execution leaves the current program state unchanged.

**Predicate Transformer Semantics:**

$$[wp(\text{skip}, R) \equiv R] \quad (1)$$

### 2. Failure

**Syntax:** `abort`

**Operational Semantics:** Execution fails to reach a final state.

**Predicate Transformer Semantics:**

$$[wp(\text{abort}, R) \equiv \mathcal{F}] \quad (2)$$

### 3. Assignment

**Syntax:**  $x := e$  (simple assignment)

Also:  $x_1, \dots, x_n := e_1, \dots, e_n$  (multiple or concurrent assignment),

where  $x, x_1, \dots, x_n$  are distinct variables and  $e, e_1, \dots, e_n$  are expressions.

**Operational Semantics:** A simple assignment, describes a state change involving a single variable, whereas multiple assignments are used to describe state changes

involving more than one variable. We elaborate on the semantics of the concurrent assignment, from which the meaning of the simple assignment should also be clear. Execution involves evaluating the expressions  $e_i$ , without affecting the program state, to obtain values  $v_i$  and then changing the program state by assigning each of these values to the variables  $x_i$  of corresponding subscripts, in any order. If evaluation of one of the expressions  $e_i$  leads to a not properly terminating activity, the whole construct is allowed to fail to terminate properly.

**Predicate Transformer Semantics:**

$$\begin{aligned} [wp((x_1, \dots, x_n := e_1, \dots, e_n), R) \equiv \\ (\forall i : i \geq 1 \wedge i \leq n : \text{domain}(e_i)) \text{ and } R_{e_1, \dots, e_n}^{x_1, \dots, x_n}] \end{aligned} \quad (3)$$

where the predicate  $\text{domain}(e_i)$  characterizes the set of states in which the expression  $e_i$  can be evaluated without causing failure. A recursive definition of  $\text{domain}(e)$  based on the structure of expressions is given in Appendix C.

**Example:**

$$wp((x, y := x \text{ div } y, y + 1), x > 100) \equiv (y \neq 0 \text{ and } x \text{ div } y > 100)$$

#### 4. Local block

**Syntax:**  $[[\text{var } x : T \mid S]],$

where  $x$  is a variable,  $T$  a type and  $S$  a program construct. This syntax is very similar to that of constructs in [40, 125, 121].

**Operational Semantics:** Execution of a local block construct as depicted above, extends the program state with a new variable  $x$  of type  $T$ , for the duration of the execution of the program  $S$ , whereafter  $x$  is released. No restriction is placed on the name of the new variable; in particular, local redeclaration of global identifiers is allowed. The introduction of multiple variables within a single local block is also permitted and involves using a list of variable declarations, such as  $\text{var } x_1 : T_1; x_2 : T_2; \dots; x_n : T_n$ , instead of the single declaration  $\text{var } x : T$ .

**Predicate Transformer Semantics:** This semantics is from [81]

$$[wp([[\text{var } x : T \mid S]], R) \equiv wp(S_x^x, R)] \quad (4)$$

where  $y$  is a fresh variable, i.e. it is not part of the existing state space and  $S_y^x$  denotes the program construct obtained by systematically substituting  $y$  for  $x$  in  $S$ . A definition of substitution in program constructs can be found in Appendix D. As remarked in [81], this definition is not always correct. A problem occurs if  $y$  is free in  $wp(S_y^x, R)$ . Since this can only happen if  $x$  is not initialized by  $S$ , we must state the additional proviso that  $x$  is duly initialized. The handling of variable initialization and other issues related to the semantics of the local block construct are discussed in more detail in sections 3.3 and 4.2.3.

**Example:**

$$\begin{aligned} wp([[\text{var } x : \text{integer} \mid x := 1]], x = n) &\equiv wp(y := 1, x = n) \\ &\equiv x = n \end{aligned}$$

## 5. Sequential composition

**Syntax:**  $S_1; S_2$

**Operational Semantics:** Sequential composition is the primary mechanism for combining basic programs to form more complex ones and it is traditionally depicted by the semicolon operator. Execution of the above sequential composition will activate program  $S_1$  in the current program state and upon its termination, activate program  $S_2$  in the resulting program state.

**Predicate Transformer Semantics:**

$$[wp(S_1; S_2, R) \equiv wp(S_1, wp(S_2, R))] \quad (5)$$

**Example:**

$$\begin{aligned} wp((x := k; y := x + 1), x > 0 \wedge y > 0) &\equiv wp(x := k, x > 0 \wedge x > -1) \\ &\equiv k > 0 \wedge k > -1 \\ &\equiv k > 0 \end{aligned}$$

## 6. Conditional composition (alternative construct)

**Syntax:**

$$\begin{array}{l}
 \text{if } G_1 \rightarrow S_1 \\
 \square G_2 \rightarrow S_2 \\
 \square \dots \\
 \square G_n \rightarrow S_n \\
 \text{fi,}
 \end{array}$$

where the  $G_i$  are Boolean expressions—called *guards*; and the  $S_i$  are program constructs.

In what follows, we will use the abbreviations:

$$\begin{array}{l}
 \text{IF for } \text{if } G_1 \rightarrow S_1 \\
 \quad \square G_2 \rightarrow S_2 \\
 \quad \square \dots \\
 \quad \square G_n \rightarrow S_n \\
 \text{fi} \\
 \\
 GG \text{ for } (\exists i :: G_i)
 \end{array}$$

**Operational Semantics:** An alternative construct allows the selection and execution of one command from among a given set. As can be seen above, it consists of a set of guarded commands. During execution, a true guard is selected from among the  $G_i$  and its corresponding program  $S_i$  is activated. If more than one guard is true, this construct gives rise to nondeterminism. If none of the guards is true, the result will be failure, i.e. equivalent to activating an abort command. Also, if evaluation of a guard leads to a not properly terminating activity, then the whole construct is allowed to fail to terminate properly.

**Predicate Transformer Semantics:**

$$\begin{aligned}
 [wp(\text{IF}, R)] &\equiv \\
 &(\forall i :: \text{domain}(G_i)) \text{ cand } (GG \wedge (\forall i :: G_i \Rightarrow wp(S_i, R)))
 \end{aligned} \tag{6}$$

**Example:**

$$wp(\text{if } x \geq 0 \rightarrow \text{skip} \square x \leq 0 \rightarrow x := -x \text{ fi}, x \geq 0)$$

$$\begin{aligned}
&\equiv (x \geq 0 \vee x \leq 0) \wedge (x \geq 0 \Rightarrow x \geq 0) \wedge (x \leq 0 \Rightarrow -x \geq 0) \\
&\equiv T
\end{aligned}$$

**Useful Theorem:** In practical situations it is not always necessary to calculate the weakest precondition of an alternative construct. It may happen that a previous program development step provides a stronger assertion that has to serve as precondition for an alternative construct yet to be developed. The following theorem from [38] allows one to test whether a given assertion is indeed a precondition of a proposed alternative construct, without having to calculate its weakest precondition.

**Theorem 3.1.0.1** *Consider the alternative construct IF and two predicates  $Q$  and  $R$  satisfying:*

- (a)  $[Q \Rightarrow GG];$
- (b)  $[(\forall i : 1 \leq i \wedge i \leq n : (Q \wedge G_i) \Rightarrow wp(S_i, R))];$

*then:*

$$[Q \Rightarrow wp(\text{IF}, R)]$$

## 7. Iteration (repetitive construct)

**Syntax:**

```

do   $G_1 \rightarrow S_1$ 
  □  $G_2 \rightarrow S_2$ 
  □ ...
  □  $G_n \rightarrow S_n$ 
od,

```

where the  $G_i$  are Boolean expressions—called *guards*; and the  $S_i$  are program constructs.

We also introduce the following abbreviation:

```

DO for do   $G_1 \rightarrow S_1$ 
          □  $G_2 \rightarrow S_2$ 
          □ ...
          □  $G_n \rightarrow S_n$ 
od

```

**Operational Semantics:** An iterative construct allows the repeated selection and execution of one of a given set of commands. The construct is made up of a set of guarded commands. When executed, a true guard is selected from among the  $G_i$  and its corresponding program  $S_i$  is activated. This process is repeated until none of the guards is true. If more than one guard is true, nondeterminism will result. No assumptions are made as to fair selection of guarded commands. As soon as no guard is true, the result is proper termination, i.e. equivalent to activating a **skip** command. The whole construct is allowed to fail to terminate properly if evaluation of a guard leads to an activity that does not terminate properly.

**Predicate Transformer Semantics:** Following [42], we require **DO** to be semantically equivalent to its first unfolding:

```

if  $G_1$      $\rightarrow$   $S_1$ ; DO
 $\square$   $G_2$      $\rightarrow$   $S_2$ ; DO
 $\square$  ...
 $\square$   $G_n$      $\rightarrow$   $S_n$ ; DO
 $\square$   $\neg GG$     $\rightarrow$  skip
fi

```

Assuming all guards to be well-defined, a few manipulations show that  $wp(\mathbf{DO}, R)$  is a solution of the following equation in predicate  $X$ :

$$[X \equiv (GG \wedge (\forall i :: G_i \Rightarrow wp_{S_i}(X))) \vee (\neg GG \wedge R)] \quad (7)$$

$$\begin{aligned}
& wp(\mathbf{DO}, R) \\
& \equiv \{ \text{definition of } wp(\mathbf{IF}, R) \text{ and the above semantic} \\
& \quad \text{equivalence} \} \\
& (GG \vee \neg GG) \wedge (\forall i :: G_i \Rightarrow wp(S_i, wp(\mathbf{DO}, R))) \wedge (\neg GG \Rightarrow \\
& \quad wp(\mathbf{skip}, R)) \\
& \equiv \{ \text{predicate calculus and definition of } wp(\mathbf{skip}, R) \} \\
& (GG \wedge (\forall i :: G_i \Rightarrow wp(S_i, wp(\mathbf{DO}, R)))) \vee (\neg GG \wedge R)
\end{aligned}$$

The predicate  $wp(\mathbf{DO}, R)$  is defined as the strongest solution of (7), i.e. that solution  $A$  such that, if  $B$  is any solution, then  $[A \Rightarrow B]$ . The equation (7) is of the form  $[X \equiv f(X)]$  with  $f$  a predicate transformer. Now we have:

- $wp_{S_i}(X)$  is an or-continuous predicate transformer
- $\Rightarrow$  {definition of chain continuity and Lemma 2.5.0.2}
- $f$  is a chain continuous function on  $(Pred_{Var}, \sqsubseteq)$
- $\Rightarrow$  {Theorem A.5.0.1}
- $f$  has a least fixed point
- $\Rightarrow$  {definition of least fixed point and Lemma 2.5.0.1}
- $\mu x.f(x)$  is the strongest solution of  $[X \equiv f(X)]$ .

Also, from Theorem A.5.0.1:

$$\mu x.f(x) \equiv \bigsqcup \{f^n(\mathcal{F})\}_{n \geq 0}$$

where

$$\begin{aligned} & \bigsqcup \{f^n(\mathcal{F})\}_{n \geq 0} \\ \equiv & \quad \{\text{Lemma 2.5.0.2}\} \\ & (\exists n : n \geq 0 : f^n(\mathcal{F})) \end{aligned}$$

Thus  $[X \equiv f(X)]$  has  $(\exists n : n \geq 0 : f^n(\mathcal{F}))$  as its strongest solution.

**Useful Theorem:** In practical situations the above definition does not provide direct guidance for program development; hence the popularity of the following theorem from [38], which allows the use of a predicate stronger than the weakest precondition in driving development steps. In a sense, this predicate, the “invariant”, embodies the essence of a repetitive construct. Hand-in-hand with the development of a repetitive construct, goes an argument for its termination, based on a well-ordered set. Our version of this theorem corresponds to the formulation in [42].

**Theorem 3.1.0.2 (Fundamental Invariance Theorem for Loops)** *Consider the repetitive construct  $\mathbf{DO}$ , a predicate  $P$  and an integer function  $t$  on the state space satisfying:*

- (a)  $[(P \wedge GG) \Rightarrow (t > 0)]$ ;
- (b)  $[(\forall i : 1 \leq i \leq n : (P \wedge G_i \wedge t = x) \Rightarrow wp(S_i, P \wedge t < x))]$  for all integer  $x$ ;

*then:*

$$[P \Rightarrow wp(\mathbf{DO}, \neg GG \wedge P)],$$

where  $wp(\text{DO}, \neg GG \wedge P)$  is defined as the strongest solution of the following equation in  $X$ :

$$[X \equiv (GG \wedge (\forall i :: G_i \Rightarrow wp(S_i, X))) \vee (\neg GG \wedge R)].$$

**Note:** The predicate  $P$  is traditionally called the *invariant* of the repetitive construct, while  $t$  is known as the *variant (bound) function* and carries the termination argument.

**Example:** Consider the following algorithm to establish

$$z = a * b$$

without the use of multiplication.

```

{b ≥ 0}
x, y, z := a, b, 0;
DO : do  y ≠ 0 ∧ even(y) → y, x := y div 2, x + x
      □  odd(y)          → y, z := y - 1, z + x
od
{R : z = a * b}

```

The weakest precondition of the above loop may now be calculated by finding  $f^0(\mathcal{F}), f^1(\mathcal{F}), f^2(\mathcal{F}), f^3(\mathcal{F}), \dots$  and forming the predicate  $(\exists n : n \geq 0 : f^n(\mathcal{F}))$ , where

$$f(X) \equiv (GG \wedge (\forall i :: G_i \Rightarrow wp(S_i, X))) \vee (\neg GG \wedge R)$$

Constructing the first four terms of this sequence:

$$\begin{aligned}
f^0(\mathcal{F}) &\equiv \mathcal{F} \\
f^1(\mathcal{F}) &\equiv (((y \neq 0 \wedge \neg \text{odd}(y)) \vee \text{odd}(y)) \wedge (y = 0 \vee \text{odd}(y)) \wedge \neg \text{odd}(y)) \\
&\quad \vee (y = 0 \wedge \neg \text{odd}(y) \wedge z = a * b) \\
&\equiv y = 0 \wedge z = a * b \\
f^2(\mathcal{F}) &\equiv ((y \neq 0 \vee \text{odd}(y)) \wedge (y = 0 \vee \text{odd}(y) \vee (y \text{ div } 2 = 0 \wedge \\
&\quad \neg \text{odd}(y \text{ div } 2) \wedge z = a * b)) \wedge (\neg \text{odd}(y) \vee (y - 1 = 0 \wedge \\
&\quad \neg \text{odd}(y - 1) \wedge z + x = a * b)) \vee (y = 0 \wedge z = a * b)
\end{aligned}$$

$$\begin{aligned}
&\equiv (\text{odd}(y) \wedge (\neg \text{odd}(y) \vee (y = 1 \wedge z + x = a * b)) \vee (y = 0 \wedge \\
&\quad z = a * b)) \\
&\equiv (y = 1 \wedge z + x = a * b) \vee (y = 0 \wedge z = a * b) \\
f^3(\mathcal{F}) &\equiv ((y \neq 0 \vee \text{odd}(y)) \wedge (y = 0 \vee \text{odd}(y) \vee (y \text{ div } 2 = 1 \wedge \\
&\quad z + 2 * x = a * b) \vee (y \text{ div } 2 = 0 \wedge z = a * b))) \wedge (\neg \text{odd}(y) \vee \\
&\quad (y - 1 = 1 \wedge z + 2 * x = a * b) \vee (y - 1 = 0 \wedge z + x = a * b))) \vee \\
&\quad (y = 0 \wedge z = a * b) \\
&\equiv ((\text{odd}(y) \vee (y = 2 \wedge z + 2 * x = a * b)) \wedge (\neg \text{odd}(y) \vee (y = 2 \wedge \\
&\quad z + 2 * x = a * b) \vee (y = 1 \wedge z + x = a * b))) \vee (y = 0 \wedge z = a * b)) \\
&\equiv (y = 2 \wedge z + 2 * x = a * b) \vee (y = 1 \wedge z + x = a * b) \vee \\
&\quad (y = 0 \wedge z = a * b)
\end{aligned}$$

leads us to “guess” the general term:

$$f^j(\mathcal{F}) \equiv (\exists i : 0 \leq i \wedge i < j : y = i \wedge z + y * x = a * b)$$

which can be proven by induction on  $j$ .

Hence

$$\begin{aligned}
wp(DO, R) &\equiv \\
&(\exists n : n \geq 0 : (\exists i : 0 \leq i \wedge i < n : y = i \wedge z + y * x = a * b))
\end{aligned}$$

Alternatively, we could use Theorem 3.1.0.2 and prove that

$$P \equiv y \geq 0 \wedge z + x * y = a * b$$

is an invariant and

$$t = y$$

a bound function for the loop  $DO$ . It then remains to prove

$$[(\neg GG \wedge P) \Rightarrow R].$$

This method has the advantage of being easier to associate with the algorithm under development as well as being more amenable to the application of heuristics (to help in finding a suitable invariant and bound function).

### 3.2 Data Types

Programs expressed in Dijkstra's guarded command language refer to data objects of various types. We give an overview of the most frequently used data types as found in examples of program development in [38, 66]. A formal description of the syntax of variable declarations and reference may be found in Appendix B.

The data types Boolean and integer are basic to Dijkstra's programming calculus. Integer subranges, in particular the natural numbers, are employed as well. We use "Modula-2-like" declarations, e.g.

```
var
  completed : boolean;
  count     : integer
```

Integer and Boolean arrays are also used extensively. An array is viewed as a function of one argument with a finite domain consisting of consecutive integers. We require that, given two array variables, the question of whether their values are equal, be decidable. Two array variables have equal values iff, as functions, their domains are the same and their values are equal in each point of the domain. This requirement necessitates finite domains. Furthermore, the domain must be available for examination.

Dijkstra's proposal to make this possible, is to use universal types "integer array" and "Boolean array", and consider the domain as part of any value of such a type. Considering an array variable *ar*, information about the domain may be extracted from its value by means of three primitive integer-valued functions [38] (see table 1).

<i>ar.lob</i>	representing the lower domain bound
<i>ar.hib</i>	representing the upper domain bound
<i>ar.dom</i>	representing the number of points in the array domain

Table 1: Dijkstra's functions for extracting array domain information.

These functions satisfy:

$$ar.dom = ar.hib - ar.lob + 1 \geq 0$$

This approach also lends itself naturally to the definition of various operators that change array domains [38] (see table 2).

<b><i>ar</i> : shift(<i>n</i>)</b>	shifts an array domain <i>n</i> places upwards, or downwards, depending on the sign of <i>n</i>
<b><i>ar</i> : hiext(<i>x</i>)</b> <b><i>ar</i> : loext(<i>x</i>)</b>	extend an array domain with one point at the high or low end, respectively, and assign the value of variable <i>x</i> as the function value in the new point
<b><i>ar</i> : hirem</b> <b><i>ar</i> : lorem</b>	remove a point from the high or low end, respectively, of an array domain
<b><i>x, ar</i> : hipop</b> <b><i>x, ar</i> : lopop</b>	the same as above, except that the function value in the lost domain point is preserved, by assigning it to a variable provided for that purpose

Table 2: Dijkstra's functions for altering array domains.

Finally Dijkstra introduces two operators that affect function values of an array without changing its domain [38] (see table 3).

<b><i>ar</i> : swap(<i>i, j</i>)</b>	exchanges the function values in domain points <i>i</i> and <i>j</i>
<b><i>ar</i> : alt(<i>i, x</i>)</b>	changes the function value in the domain point <i>i</i> to the value of variable <i>x</i>

Table 3: Dijkstra's functions for changing array function values.

In section 4.2.4 below, the treatment of arrays is simplified somewhat, by considering the domain of an array as a fixed entity that must be unambiguously stated when an array variable is declared. This obviates the need for most of the primitive functions detailed above. The following example shows a typical array declaration containing explicit domain bounds:

**var**

***ar* : array[0..5] of integer**

For an array variable *ar*,

***ar*[*i*]**

is used to denote function application to the argument *i*—*ar*[*i*] is only defined if the argument

$i$  is defined and falls within the domain of the function in  $ar$ . Following [66] we also introduce notation for an array that has been altered in a specific domain point:

Let  $ar$  be an array,  $i$  a suitable argument for the function  $ar$  and  $e$  an expression of the same type as the individual array values. Now

$$ar([i] : e)$$

denotes the array that is the same as  $ar$  except that its value in the domain point  $i$  is  $e$ :

$$ar([i] : e)[j] = \begin{cases} e & \text{if } i = j \\ ar[j] & \text{if } i \neq j \end{cases}$$

In keeping with tradition, the assignment  $ar := ar([i] : e)$  is abbreviated to  $ar[i] := e$ .

We extend the above notation where necessary to allow for redefinition at several points of an array domain. The following example reflects changes at two domain points,  $i$  and  $j$ , of array  $ar$ :

$$ar([i] : e, [j] : f)[k] = \begin{cases} f & \text{if } j = k \\ e & \text{if } j \neq k \wedge i = k \\ ar[k] & \text{if } j \neq k \wedge i \neq k \end{cases}$$

Dijkstra also uses record types, [38, Chapter 15]. The functional view of arrays may be extended to record types, viewing a record as a function from a finite set of labels (field names) to a set of values, see [66, page 92]. The following is an example declaration of a record variable:

```
var
  game : record
    || turns : integer;
       won   : boolean
    ||
```

With reference to the above declaration, the notation

*game.won*

is used for function application to the argument *won*. Analogous to arrays we introduce notation for a record that has been changed in a specific domain point [66]:

Let  $\equiv$  denote syntactic equality. Let *rec* be a record with field names *m* and *n*, with *f* an expression of the same type as field *m*. Then

$$rec(.m : f).n$$

is the record that is the same as *rec* except that its value in the domain point *m* is *f*:

$$rec(.m : f).n = \begin{cases} f & \text{if } m \equiv n \\ rec.n & \text{if } m \not\equiv n \end{cases}$$

As for arrays, the assignment  $rec := rec(.m : f)$  is abbreviated to the traditional form  $rec.m := f$ .

As in [66] we expand our view of arrays and records to include arrays of arrays (multidimensional arrays), arrays of records, records of arrays and records of records. In keeping with the above, these are interpreted as higher order functions (see section 4.2.4).

To allow full generality in referring to subarrays of records and subrecords of arrays, the above notation for alteration in a single domain point is expanded as follows [66]:

Let  $\equiv$  denote syntactic equality. Let *ar* be an array and *rec* a record with field names *m* and *n*. Let *e* and *f* be expressions of appropriate types. Let *s* and *t* be suitable selectors for *ar* and *rec*, respectively. The term *selector* refers to a finite sequence of field names (each prefixed with a dot) and subscript expressions (individually enclosed in square brackets) e.g.  $[i][j].x$ . The null selector is denoted by  $\varepsilon$  and forms the identity element of the catenation operation,  $\circ$ , on identifiers and selectors. Thus  $s \circ \varepsilon = s$  for a selector *s*.

$$ar(\varepsilon : e) = e$$

$$rec(\varepsilon : f) = f$$

$$\begin{aligned}
ax([i] \circ s : e)[j] &= \begin{cases} ax[j](s : e) & \text{if } i = j \\ ax[j] & \text{if } i \neq j \end{cases} \\
rec(.m \circ t : f).n &= \begin{cases} rec.n(t : f) & \text{if } m \equiv n \\ rec.n & \text{if } m \not\equiv n \end{cases}
\end{aligned}$$

Simple character variables as well as character arrays are often used in guarded command programs, without a formal treatment of this data type being presented. In what follows we use a character data type at least including all capital and lower case letters of the alphabet as well as the ten decimal digits. We assume that the character data type is totally ordered.

Enumerated types are also useful in program development, e.g. use of the type “colour” in the problem of the Dutch national flag [38, Chapter 14]. Other types that are used in connection with the programming calculus include sets, power sets, bags and sequences such as found in [66, 121].

### 3.3 The Scope of Identifier Declarations

The guarded command language presented above has a block structure such as that found in ALGOL, Modula-2 or Pascal, allowing the introduction of identifiers at the exact point where the need for them arise, followed by their release after having served their purpose. A local block is delimited by the opening and closing brackets `[[` and `]]`, indicating the boundaries within which locally declared identifiers form part of the state space. Within these boundaries, the state space is temporarily extended to give access to the local identifiers.

As in Modula-2 and Pascal, no restriction is placed on the choice of local identifiers—essentially allowing temporary redeclaration of a global identifier within a local block. This is clearly reflected in the weakest precondition semantics of the local block construct:

$$[wp([[\text{var } x : T \mid S]], R) = wp(S_y^x, R)]$$

where  $y$  is a fresh variable and  $S$  initializes  $x$ .

As indicated in the definition, name clashes are handled by renaming all local identifiers. Another option is to temporarily rename those global identifiers appearing in the postcondition

that are redeclared within the local block (see [81, page 118] or [130] for examples of this approach). This would lead to the following weakest precondition semantics for local blocks:

$$[wp([var\ x : T \mid S], R) = (wp(S, R_y^x))_x^y]$$

where  $y$  does not occur free in  $R$  or  $S$  and  $x$  is initialized by  $S$ .

In contrast to the former approach, i.e. renaming of local identifiers, this has the advantage of not requiring name substitutions within program constructs as well as limiting the number of identifiers to be renamed to the minimum. However, a more subtle (and unwanted) consequence of this choice of semantics becomes apparent when we consider extending our guarded command language to include procedures, as we shall see in Chapter 5. Further discussion is delayed until we approach that subject.

Permitting redeclaration of identifiers, allows a degree of freedom in the development of local blocks, since the author of a local block need only be aware of those global identifiers accessed within that block. This feature has also been criticized [38], because the state space is contracted implicitly rather than explicitly and disallowing its use leads to a simplification in the weakest precondition semantics of the local block construct.

The fact that locally declared identifiers are only accessible within their block of declaration is in accordance with good programming practice—among other reasons because it limits the program text a programmer has to consider in the case of an unintended change to the value of a local variable. Some criticism has been leveled at the absence of a similar limit on the accessibility of variables declared in surrounding blocks (“global” variables), see [38, Chapter 10]. To fulfill this need, Dijkstra proposes formally viewing local blocks as a mechanism for altering the state space, rather than exclusively for extending it. This opens a question of whether enumeration of all global identifiers accessible within a local block should be mandatory at the start of every local block, or only in cases where contraction of the state space (limited inheritance of global identifiers) is specifically required. In [38], Dijkstra opts for the former alternative. As this burdens a programmer with having to write possibly long lists of identifiers at the start of each local block, we will not pursue this suggestion here. Later discussions on the introduction of abstraction mechanisms such as procedures and modules, provide further perspectives on this issue.

### 3.4 Specifications and Annotations

'Would you tell me, please, which way I ought to go from here?'

'That depends a good deal on where you want to get to,' said the Cat.

'I don't much care where—' said Alice.

'Then it doesn't matter which way you go,' said the Cat.

*Lewis Carroll, Alice's Adventures in Wonderland*

Development of a guarded command program involves the construction of one or more program statements that together realize a functional specification of program behaviour. Such a specification states the desired relation between the initial and final states of a computation. Since input and output mechanisms are not standard and difficult to axiomatize, it is customary to assume that the initial state of a computation is directly determined by the input and the final state by what is to be the output. As described in Chapter 2, we use a predicate, called a postcondition, to characterize the allowable final states of a computation and another predicate, the precondition, to characterize the corresponding initial states.

We use the following basic notation for specifications<sup>1</sup>:

$$[pre, post] \tag{8}$$

where *pre* is the precondition and *post* the postcondition.

To satisfy specification (8) we would have to develop a program *S* such that:

$$[pre \Rightarrow wps(post)]$$

or, stated operationally:

If execution of *S* is started in a state satisfying *pre*, then it is guaranteed to terminate in a state satisfying *post*.

For completeness, a functional specification should also contain a list of all those variables whose values may be changed. This list is called the *frame* [125]. Our notation is the same

---

<sup>1</sup>This notation is not part of the guarded command language

as in [124, 122, 125, 121, 123]:

$$x : [pre, post] \quad (9)$$

where *pre* is the precondition, *post* the postcondition and *x* the frame.

It is important to realize that merely expressing a functional specification in the given notation is by no means a guarantee of satisfiability. A specification (8) is satisfiable iff

$$(\exists S :: [Q \Rightarrow wps(R)]).$$

A guarded command program is annotated with assertions expressing the program specification, internal assertions such as loop invariants and bound functions. Internal annotations are to be used in proving the program's consistency with its specification. We follow the convention of interspersing the guarded commands of a program with annotations enclosed in curly braces, "{" and "}". In particular, each loop must be annotated with an invariant and a bound function. Figure 1 shows a guarded command program annotated with its specification, as well as a loop invariant and a bound function. Placing an assertion in a program is used to express properties that the state space must possess at that point of execution. Such assertions act as lemmas or subgoals in proofs concerning the program.

```

{pre : b ≥ 0}
x, y, z := a, b, 0;
{invar P : y ≥ 0 ∧ z + x * y = a * b}
{bound t : y}
DO : do  y ≠ 0 ∧ even(y)  →  y, x := y div 2, x + x
      □  odd(y)           →  y, z := y - 1, z + x
od
{post : z = a * b}

```

Figure 1: An annotated loop.

Till now we have placed little emphasis on the syntactic issues surrounding predicates (conditions), but in time we shall have to be more specific about a suitable language in which program assertions and specifications can be represented by formulas. An obvious candidate is the set of first-order formulae over the state space and operators available in the guarded command language. As pointed out in [11, 81], this language is not sufficiently powerful to express preconditions of iterative constructs. In terms of our semantic framework, this set

of formulae does not exhibit a suitable lattice structure. From Lemmas 2.5.0.1 and 2.5.0.2 we see that the partial order on predicates should correspond to logical implication, while the least upper bound of a set of predicates must correspond to their disjunction. Such a disjunction is not always expressible in the above language. We reproduce an example from [81]:

Consider the set of conditions  $S = \{n = 1, n = 1 \times 2, n = 1 \times 2 \times 3, \dots\}$ .

The least upper bound of  $S$  is  $n = 1 \vee n = 1 \times 2 \vee n = 1 \times 2 \times 3 \dots$ , i.e. an infinite disjunction.

First order formulae equivalent to this disjunction would be

$$(\exists i : i \geq 1 : n = i!)$$

or

$$(\exists i : i \geq 1 : n = (\prod j : 1 \leq j \wedge j \leq i : j))$$

both of which contain operators not present in the guarded command language.

In [38, 66, 40] and others, the language of first order formulae over the state space and the guarded command program operators, extended by the use of special quantifiers (“sum”, “product” and “number”) and conditional connectives (**cand** and **cor**) in the presence of undefined expressions, is used to express conditions. A specification language encompassing a bigger set of operators than a programming language is quite useful in providing more compact ways of expression, but poses the problem of determining whether the extended language is closed under infinite disjunction or conjunction.

A theoretically simpler solution is proposed in [9, 11], namely that infinite disjunctions and conjunctions be allowed in the language of conditions. This leads to the use of formulas of the infinitary logic  $L_{\omega_1, \omega}$ , which is an extension of ordinary first-order logic allowing disjunctions and conjunctions over countable sets of formulae, but quantification only over finite sequences of variables. A completely formal treatment of this and other infinitary logics can be found

$$\forall x (\bigvee_{i=0}^{\infty} x = i)$$

Figure 2: A formula of  $L_{\omega_1\omega}$ .

in [101]. In [10] this logic is shown to be the weakest one sufficiently rich to express the conditions required for guarded commands.

Thus the pre- and postconditions used in specifications of guarded command programs, are formulae of a suitable first order logic, such as  $L_{\omega_1\omega}$ , and may be manipulated according to the rules of this logic. This, together with the relevant theories about integers and other data types involved in a specification, may form the basis of a formal argument (proof) that a given mechanism satisfies (9) as an equation in  $S$ .

### 3.5 Developing Guarded Command Programs

Dijkstra intended weakest precondition predicate transformers to be used as the basis for a calculus of program development. The term calculus is used to signify a set of formal rules which, if successfully applied, will result in the derivation of a program which is consistent with its specification. As in the case of integral calculus though, simple mechanical application of rules do not guarantee success.

In [66], various *program development strategies* and principles, which use weakest precondition calculations in a constructive manner, are formulated. Weakest precondition calculations are generally used for the following purposes [43]:

1. to identify a suitable guard to combine with a command in a guarded command construct;
2. to identify a command that will ensure that an invariant is maintained;
3. to identify a subgoal (necessary precondition) to be established before a command can be executed.

Given a precondition  $Q$  and a postcondition  $R$  program development typically starts by selecting, under guidance of the weakest precondition calculus, a program construct  $S_1$  for which  $wp(S_1, R)$  is such that either

- $[Q \Rightarrow wp(S_1, R)]$   
where  $[R_1 \Rightarrow R]$ , in which case the program is complete; or
- $[Q_1 \Rightarrow wp(S_1, R)]$   
with  $Q_1$  a subgoal for establishing the original postcondition  $R$ , in which case we repeat the above for the specification  $[Q, Q_1]$ . In this fashion we obtain a program consisting of the sequential composition of commands  $S_n, S_{n-1}, \dots, S_1$ .

The two main program construction strategies proposed by Gries, i.e. those for developing alternative commands and loops, are derived from Theorems 3.1.0.1 and 3.1.0.2 above. These strategies are as follows [66]:

**Strategy for developing an alternative command:** Invent guarded commands until the proposed precondition  $Q$  of the construct implies that at least one guard is true ( $Q \Rightarrow GG$ ). To invent a guarded command:

- find a command  $C$  which will establish postcondition  $R$  in at least some cases;
- find a Boolean expression  $G$  satisfying  $G \Rightarrow wp(C, R)$  (or  $Q \wedge G \Rightarrow wp(C, R)$ );

and put them together to form  $G \rightarrow C$ .

**Strategy for developing a loop:** Formulate a suitable invariant  $P$  and a bound function  $t$  (see below). Invent guarded commands until:

- the loop invariant and the negation of the disjunction of the guards ( $\neg GG$ ) implies that the postcondition  $R$  holds ( $P \wedge \neg GG \Rightarrow R$ );
- the loop invariant and the disjunction of the guards implies that the value of the bound function remains bounded from below by zero ( $(P \wedge GG) \Rightarrow (t > 0)$ ).

To invent a guarded command:

- create a command  $C$  which makes progress towards termination (decreases the bound function);
- develop a corresponding guard  $G$  to ensure that the invariant is maintained;

and put these together to form  $G \rightarrow C$ .

A bound function is normally stated in terms of the guards of the guarded commands that comprise a loop. There are no hard-and-fast rules for developing bound functions, but some guidelines can be found in [66, 43].

As for bound functions, there are no mechanical rules for inventing loop invariants. Application of existing heuristics need to be guided by a good measure of insight and ingenuity. A general heuristic for finding a loop invariant is to weaken the postcondition of a loop construct [66]. There are various ways of weakening a predicate, of which the following are often successful in deriving an invariant from a postcondition [66]:

1. deleting a conjunct;
2. replacing a constant by a variable, together with suitable bounds on its allowable values;
3. enlarging the range of allowable values for a variable.

Sometimes input variables have to be modified themselves to form part of the result. In such cases an invariant has to express the fact that part of the input remains unchanged, requiring the additional technique of combining the pre- and postconditions of a loop (see [66, page 211]).

Other development strategies, specifically for stepwise refinement of guarded command programs may be found in [43].

### 3.5.1 An Example

To illustrate Gries's main development strategies, we show part of the construction of a solution to "Rubin's problem":

Determine whether a row containing only zeros is to be found in a given nonempty, two dimensional array.

The formulation and specification of this problem is based on [43, page 414], which may be referenced for a different derivation of the same solution as well as the origins of the problem.

### Problem Specification and Data Declarations

We assume declarations for two natural number constants, *MAXCOL* and *MAXROW*, which give the dimensions of the array, as well as the following variables:

*matrix* : array [1..*MAXROW*][1..*MAXCOL*] of integer  
*found* : boolean  
*lastrow, lastcol* : integer  
*curcol, currow* : integer

The variables *currow* and *curcol* give the row and column positions, respectively, which have already been checked for zero entries, while *lastrow* and *lastcol* indicate the row and column, respectively, up to which one has to keep looking for zeros.

We will also need three predicate abbreviations:

1. The predicate *nonzero*,

$$\begin{aligned} \text{nonzero}(\text{matrix}, \text{maxcol}, \text{rowpos}, \text{colpos}, \text{lastcol}) \equiv \\ (\forall \text{col} : 0 < \text{col} \wedge \text{col} \leq \text{colpos} : \text{matrix}[\text{rowpos}][\text{col}] = 0) \\ \wedge (\text{lastcol} = \text{maxcol} \text{ cor } \text{matrix}[\text{rowpos}][\text{colpos} + 1] \neq 0) \end{aligned}$$

which states that, in a particular row (*rowpos*), all the entries up to a certain column position (*colpos*) are zeros and, unless *lastcol* is the last column of the matrix, the position *colpos* + 1 contains a non-zero entry.

2. The predicate *allzero*,

$$\begin{aligned} \text{allzero}(\text{matrix}, \text{maxcol}, \text{rowpos}) \equiv \\ (\forall \text{col} : 0 < \text{col} \wedge \text{col} \leq \text{maxcol} : \text{matrix}[\text{rowpos}][\text{col}] = 0) \end{aligned}$$

which states that a particular row contains only zero entries.

3. The predicate *inbounds*,

$$\begin{aligned} \text{inbounds}(\text{maxindex}, \text{index}, \text{lastindex}) \equiv \\ (0 \leq \text{index}) \wedge (\text{index} \leq \text{lastindex}) \wedge (\text{lastindex} \leq \text{maxindex}) \end{aligned}$$

which states an ordering on array indices *maxindex*, *index*, and *lastindex*.

The precondition *pre* states our requirements for the initial values of *MAXROW*, *MAXCOL* and *matrix*

$$\begin{aligned} \text{pre} \equiv & (\text{MAXROW} > 0) \wedge (\text{MAXCOL} > 0) \wedge \\ & (\forall \text{row} : 0 < \text{row} \wedge \text{row} \leq \text{MAXROW} : (\forall \text{col} : 0 < \text{col} \wedge \text{col} \leq \text{MAXCOL} : \\ & \text{matrix}[\text{row}][\text{col}] \in \text{INTEGERS})) \end{aligned}$$

The postcondition *post* states that there is either no row containing only zero entries, or the row at position *currow* + 1 is such a row:

$$\begin{aligned} \text{post} \equiv & (\forall \text{row} : 0 < \text{row} \wedge \text{row} \leq \text{currow} : (\exists \text{col} : 0 < \text{col} \wedge \text{col} < \text{MAXCOL} : \\ & \text{nonzero}(\text{matrix}, \text{MAXCOL}, \text{row}, \text{col}, \text{col})) \wedge \\ & (\text{lastrow} = \text{MAXROW} \text{ cor } \text{allzero}(\text{matrix}, \text{MAXCOL}, \text{currow} + 1)) \\ & \wedge (\text{lastrow} = \text{currow}) \wedge (\text{found} = (\text{currow} \neq \text{MAXROW})) \end{aligned}$$

### Constructing a Solution

It is easy to see that the last conjunct of *post* may be established by the command:

$$\text{found} := \text{currow} \neq \text{MAXROW}$$

To obtain the next subgoal, we calculate  $\text{wp}(\text{found} := \text{currow} \neq \text{MAXROW}, \text{post})$ . This gives:

$$\begin{aligned} \text{post}_1 \equiv & (\forall \text{row} : 0 < \text{row} \wedge \text{row} \leq \text{currow} : (\exists \text{col} : 0 < \text{col} \wedge \text{col} < \text{MAXCOL} : \\ & \text{nonzero}(\text{matrix}, \text{MAXCOL}, \text{row}, \text{col}, \text{col}))) \wedge \\ & (\text{lastrow} = \text{MAXROW} \text{ cor } \text{allzero}(\text{matrix}, \text{MAXCOL}, \text{currow} + 1)) \\ & \wedge (\text{lastrow} = \text{currow}) \end{aligned}$$

We will develop a loop  $DO_1$  to establish  $post_1$ . Each iteration will examine one row of the matrix in turn. If a row containing only zeros is found, we need not consider further rows and the loop terminates immediately, otherwise iteration continues until all rows have been considered. To avoid unnecessary repetition, we do not give details of the overall development of  $DO_1$ . The reader may use the following loop invariant

$$\begin{aligned} P_1 \equiv & \text{inbounds}(MAXROW, currow, lastrow) \wedge (\forall row : \\ & 0 < row \wedge row \leq currow : (\exists col : 0 < col \wedge col < MAXCOL : \\ & \text{nonzero}(\text{matrix}, MAXCOL, row, col, col))) \wedge \\ & (lastrow = MAXROW \text{ cor } \text{allzero}(\text{matrix}, MAXCOL, currow + 1)) \end{aligned}$$

the bound function

$$t_1 \equiv lastrow - currow$$

and the predicate

$$\begin{aligned} post_2 \equiv & \text{inbounds}(MAXROW, currow, lastrow) \wedge (\forall row : \\ & 0 < row \wedge row \leq currow : (\exists col : 0 < col \wedge col < MAXCOL : \\ & \text{nonzero}(\text{matrix}, MAXCOL, row, col, col))) \wedge (lastrow = MAXROW) \\ & \wedge (currow <> lastrow) \wedge (t_1 = T_1) \wedge (curcol = lastcol) \\ & \wedge \text{nonzero}(\text{matrix}, MAXCOL, currow + 1, curcol, lastcol) \end{aligned}$$

(where  $T_1$  is a logical constant representing the initial value of the bound function  $t_1$ ) to confirm developments up to the point reflected in figure 3.

We now develop a loop  $DO_2$ , using  $post_2$  as postcondition. Each iteration of the loop will examine one column of row  $currow + 1$  for a zero entry. The loop terminates as soon as a nonzero entry is found, otherwise it continues until all columns have been examined.

We find an invariant  $P_2$  by weakening  $post_2$ . We use the technique of changing a constant

```

{pre}
⋮
{invar  $P_1$ }
{bound  $t_1$ }
 $DO_1$  : do    $currow \neq lastrow$       →
           ⋮
           {post2}
            $IF_1$  : if  $curcol = MAXCOL$  →  $lastrow := currow$ 
                  □  $curcol \neq MAXCOL$  →  $currow := currow + 1$ 
           fi
           od;
{post1}
 $found := currow \neq MAXROW$ 
{post}

```

Figure 3: Partial solution to Rubin's problem.

( $curcol = lastcol$ ) in  $post_2$  to a variable which ranges over the column positions to be considered ( $0 \leq curcol \wedge curcol \leq lastcol$ ):

$$\begin{aligned}
P_2 \equiv & \text{inbounds}(MAXROW, currow, lastrow) \wedge (\forall row : 0 < row \\
& \wedge row \leq currow : (\exists col : 0 < col \wedge col < MAXCOL : \\
& \text{nonzero}(\text{matrix}, MAXCOL, row, col, col)) \wedge (lastrow = MAXROW) \wedge \\
& (currow <> lastrow) \wedge (t_1 = T_1) \wedge \text{inbounds}(MAXCOL, \\
& curcol, lastcol) \wedge \text{nonzero}(\text{matrix}, MAXCOL, currow + 1, curcol, lastcol)
\end{aligned}$$

It is clear that iteration must continue until  $curcol = lastcol$ , which literally means that the last column position which could give us success has been examined. From this we derive the bound function

$$t_2 \equiv lastcol - curcol$$

and we choose for our first guarded command, the guard

$$G_2 \equiv curcol \neq lastcol$$

Immediately we have  $P_2 \wedge \neg G_2 \Rightarrow post_2$ , indicating that not more than one guarded command will be necessary. Also  $P_2 \wedge G_2 \Rightarrow t_2 > 0$ .

We now develop a command to be guarded by  $G_2$ . This command must decrease  $t_2$  and reestablish  $P_2$ . We can immediately identify two sensible ways of decreasing the bound function:

1. Increment the column index  $curcol$  by one, i.e. use the command  $curcol := curcol + 1$ . This is a logical step to take when the next column contains a zero.
2. Set  $lastcol$  to the current column position, i.e.  $lastcol := curcol$ . This should be done when the next column does not contain a zero, as the current column then contains the last consecutive zero from the beginning of this row.

This case analysis shows that we need an alternative construct  $IF_2$  comprising at least two guarded commands.

For the first case we calculate:

$$\begin{aligned} wp(curcol := curcol + 1, P_2) \equiv & \\ & inbounds(MAXROW, currow, lastrow) \wedge (\forall row : 0 < row \wedge \\ & row \leq currow : (\exists col : 0 < col \wedge col < MAXCOL : \\ & nonzero(matrix, MAXCOL, row, col, col)) \wedge (lastrow = MAXROW) \wedge \\ & (currow <> lastrow) \wedge (t_1 = T_1) \wedge inbounds(MAXCOL, curcol + 1, \\ & lastcol) \wedge nonzero(matrix, MAXCOL, currow + 1, curcol + 1, lastcol) \end{aligned}$$

The only conjunct which is not implied by  $P_2 \wedge G_2$  is  $matrix[currow + 1][curcol + 1] = 0$ . We choose this expression as the guard for the command  $curcol := curcol + 1$ .

For the second case we calculate:

$$\begin{aligned} wp(lastcol := curcol, P_2) \equiv & \\ & inbounds(MAXROW, currow, lastrow) \wedge (\forall row : 0 < row \wedge \\ & row \leq currow : (\exists col : 0 < col \wedge col < MAXCOL : \\ & nonzero(matrix, MAXCOL, row, col, col)) \wedge (lastrow = MAXROW) \wedge \\ & (currow <> lastrow) \wedge (t_1 = T_1) \wedge inbounds(MAXCOL, curcol, \\ & curcol) \wedge nonzero(matrix, MAXCOL, currow + 1, curcol, curcol) \end{aligned}$$

For the above to be implied by  $P_2 \wedge G_2$ ,  $matrix[currow + 1][curcol + 1] \neq 0$  must hold. We choose this expression as the guard for the command  $lastcol := curcol$ .

## CHAPTER 3. DIJKSTRA'S PROGRAMMING CALCULUS

51

The completed program is shown in figure 4. The reader may verify that the initializations are correct.

```

{pre}
currow, lastrow := 0, MAXROW;
{invar  $P_1$ }
{bound  $t_1$ }
DO1: do   currow  $\neq$  lastrow            $\rightarrow$    curcol, lastcol := 0, MAXCOL;
        {invar  $P_2$ }
        {bound  $t_2$ }
        DO2: do   curcol  $\neq$  lastcol        $\rightarrow$ 
            IF2: if   matrix[currow + 1][curcol + 1] = 0  $\rightarrow$    curcol := curcol + 1
                 $\square$    matrix[currow + 1][curcol + 1]  $\neq$  0  $\rightarrow$    lastcol := curcol
                fi
            ed;
        {post2}
        IF1: if   curcol = MAXCOL  $\rightarrow$    lastrow := currow
             $\square$    curcol  $\neq$  MAXCOL  $\rightarrow$    currow := currow + 1
            fi
        od;
{post1}
found := currow  $\neq$  MAXROW
{post}

```

Figure 4: Solution to Rubin's problem.

## Chapter 4

# Mechanical Support for Dijkstra's Calculus

We now start our investigation of the main concern of this treatise, namely the suitability of Dijkstra's programming calculus to be used in mechanically supported program (algorithm) development as well as the type of tools that would best assist its application. The discussion centers around a prototype implementation that was undertaken to obtain insight into these issues, as well as a survey of results from other projects, as reported in the literature.

The first section details aspects of program development using Dijkstra's calculus that are amenable to automation and explains the main considerations and criteria that guided the form and content of the prototype implementation.

After this, different aspects of the mechanization of precondition calculations are discussed. First, a brief overview is given of the formal syntax adopted for annotated guarded command programs. A grammar for the language accepted by the prototype implementation appears in Appendix B. After this, requirements for the manipulation and representation of expressions during precondition calculations receive attention. Problems regarding the semantics of programs containing references to uninitialized variables are pointed out and a number of possible solutions are evaluated. We also discuss the formalized use of logical constants in program annotations. The treatment of data types, specifically arrays and records, in

Dijkstra's calculus and in the prototype implementation is considered, together with some suggestions for generalization and extension. This section is concluded with a discussion of problems surrounding multiple assignments to compound variables and the use of quantified expressions in program annotations.

The next section explores support mechanisms for generating and discharging the proof obligations arising from application of Dijkstra's calculus. The batch paradigm of verification condition generation is considered first. Here we pay special attention to the role played by program annotations and certain logical constants. This is followed by a discussion of some of the problems to be solved in trying to formalize the specification language used with guarded command programs. These include the lack of expressiveness of its underlying first-order logic as well as the informal treatment of undefined terms. After this, some perspectives are given on ways in which to support interactive program development. Finally, some desirable features are highlighted for the proof support component of a mechanical program development system based on Dijkstra's calculus. Special consideration is given to the simplification of logical formulae.

Conclusions are presented in section 4.4.

## **4.1 Considerations for a Prototype System**

We proceed by identifying and discussing various aspects of program development using Dijkstra's calculus that show potential for being partially or completely automated:

**calculation of (weakest) preconditions:** This activity is basic to the application of Dijkstra's programming discipline, in that further program development steps are guided by these conditions and they play an important role in the proof obligations that stem from the development. Following the rules of the weakest precondition calculus, calculating the weakest precondition of a program construct with respect to a given postcondition is a mechanical procedure in all but one case. In general, calculation of the weakest precondition of a repetitive construct is not as simple, because it is based on induction [146] and requires the formulation of a property that will carry an inductive proof.

In practical terms it also turns out to be more convenient to work with a suitable approximation of the weakest precondition itself. The responsibility of supplying an invariant condition and a bound function from which a proof of correctness of a repetitive construct can be constructed, traditionally rests with the programmer.

**generation of invariants and bound functions:** Supplying a suitable invariant and bound function for a loop under construction, requires a thorough understanding of its intended purpose and *modus operandi* as well as a certain amount of ingenuity. Since there is no “algorithm” for producing invariants and bound functions (see section 3.5), they cannot be generated solely by mechanical means and their semi-automated generation involves heuristics. A programmer is expected to use his/her deeper understanding of the principles involved to control the application of heuristic strategies and to salvage the situation when these fail to produce a solution. Some mechanical help may be beneficial, but the programmer remains the main protagonist in this operation.

**proof assistance:** Use of the weakest precondition calculus generates proof obligations to be fulfilled in showing consistency between programmer supplied annotations and developed code. Proof obligations take the form of a set of logical formulae such that proving the validity of these formulae in a suitable first order logic, is sufficient to prove that an implementation meets its specification. These formulae are called *verification conditions*. Generating verification conditions can be completely automated, provided that a specification and, in the case of a repetitive construct, an invariant and a bound function are supplied.

Mechanically generated verification conditions tend to be lengthy, obscuring useful facts for further program development. Certain automatic theorem proving techniques may be applied to simplify these conditions before proving takes place.

Incompleteness and undecidability results for the subject domain under consideration, indicate that proofs of verification conditions are not fully automatable. Nevertheless, partially automated theorem proving is very desirable in relieving programmers from detailed symbol manipulation and reducing the probability of human error.

**supporting development strategies:** Strategies for the development of guarded command programs and invariants, such as those mentioned in section 3.5, are characteristic

of the application of Dijkstra's calculus. Since the main program development strategies are guided by precondition calculations and theorem proving operations, meaningful automation of programming strategies must be built on good mechanizations for performing such functions and their successful integration into an interactive environment.

**general development support:** We classify administrative support such as storage and retrieval of programs and proofs, pretty-printing, editing, and support for user interaction under this heading.

In deciding on the exact form and extent of a prototype implementation, the following facts and criteria were used as the guidelines:

- The main purpose of the implementation is to gain insight and not to produce a highly sophisticated and generally "useful" system.
- The prototype should be suitable for incremental enhancements by future efforts exploiting the knowledge gained, or for connection to existing components from other sources, such as a powerful algebraic simplifier, theorem prover or proof editor where these seem appropriate.
- Due consideration should be given to whatever lessons can be learned from other similar projects.
- Mechanical tools are not meaningful unless they are powerful enough to be preferable to paper and pencil, yet convenient to use.
- Aspects of using the calculus that occupy considerable time, without having significant intellectual content are good candidates for automation.
- Unique aspects of the calculus are more likely to deliver interesting or new results when automated than features that show a high degree of similarity to those of other program verification or development systems.
- The fact that Dijkstra's discipline of programming is specifically suitable for the development of algorithms, suggests that mechanical support for the method could find

application in teaching environments. To make such a system accessible to a wide audience, it should have minimal resource requirements.

Stand-alone mechanical systems supporting most of the above aspects, such as the Gypsy Verification Environment [1, 3, 57] and the Stanford Pascal Verifier [69, 1, 2], are large systems developed over several years (in the case of Gypsy—more than a decade). Smaller implementations are often integrated with sophisticated theorem proving environments such as [61, 4] in HOL and [80] in the Karlsruhe Interactive Verifier, or used in conjunction with existing special-purpose tools such as arithmetic simplifiers [143]. For the current purposes, a simple prototype was constructed, focusing on the most fundamental and obvious candidates for automation, while providing a good foundation for future research, i.e. calculation of preconditions and generation of verification conditions, with limited attention to administrative support.

Research in automated theorem proving techniques and systems encompasses a vast spectrum [109, 107] and there are no clear-cut solutions to the intricate problems encountered in this field. Issues such as powerful hardware, productive user direction of mechanical provers as well as management of recorded proofs and provision for reusable theories to extend knowledge, form part of the kaleidoscope of factors that distinguish an invaluable tool from a constant source of frustration. This thesis is not intended as a contribution to mechanical theorem proving research. Therefore, I did not include a theorem proving component in the prototype. It could be used, however, to gather information about specific theorem proving requirements, imposed on an automated theorem prover by the use of Dijkstra's calculus.

The majority of existing mechanical verification systems, generate proof obligations (verification conditions) in batch mode. A programmer has to supply a specification as well as a suitably annotated program before verification conditions are generated. However, mechanical support for a calculational style of stepwise program development, guided by weakest precondition transformers (as embodied in Gries's program development strategies (see section 3.5)), should be interactive. In such an environment (weakest) precondition calculations, algebraic simplifications and the generation and discharging of verification conditions are interleaved with administrative functions, to guide and support a programmer in every step of the process of program construction.

It is important that interactive support for program development does not impose unnecessary restrictions on programmers. For example, a programmer should be able to undo undesirable development steps with as little impact on remaining work as possible and should have control over strategies used by the system and the order in which subgoals are tackled or proof obligations discharged where this is irrelevant to the correctness of the method. At the same time an implementation of program development strategies should be carefully constructed not to compromise soundness. In the light of the above, it seems that mechanical support for the use of development strategies (for programs, invariants, and/or bound functions) needs to be quite sophisticated in order to be meaningful.

Another obstacle to interactive support is the sheer bulk and complexity of the mechanically computed preconditions that are displayed during program development. Unless a highly effective mechanical simplifier, integrated with the system, is available to reduce the preconditions to such an extent that they may be readily grasped by a programmer, the fact that preconditions are computed interactively loses most of its attraction [143]. Because of these considerations, the prototype implementation does not provide direct support for formal program construction and follows the (admittedly less than ideal) batch paradigm of verification condition generation. It is used in conjunction with an existing programmable text editor, that has been customized with macros to help a programmer apply Gries's programming strategies.

The prototype system was implemented in Modula-2 on an IBM-compatible personal computer.

## **4.2 Generating (Weakest) Preconditions**

### **4.2.1 Formalizing the Syntax**

To facilitate calculation of (weakest) preconditions, a formal presentation of the syntax of guarded command language programs, annotated with first order predicate formulas (program assertions), is necessary. The syntax of guarded command statements, as well as suitable integer and Boolean expressions is given in BNF in [40]. Building on this foundation, a context

free grammar was developed for annotated guarded command programs. This grammar appears as Appendix B. A number of its features are listed here:

- Variable declarations follow the syntax used in section 3.2, and provides for the standard types INTEGER, BOOLEAN, and CHARACTER, as well as arrays and records.
- The domain of an array type must be an integer subrange, bounded by explicitly stated integer constants (see section 3.2). To cater for domains such as the well known  $[0..n-1]$ , domain bounds are allowed to take the form of simple expressions involving constants.
- Provision is made for global as well as local declarations of integer, Boolean, and character constants. Constants of compound data types are not catered for.
- Program statements include all the constructs listed in section 3.1.
- The requirement that an invariant as well as a bound function be supplied with each repetitive construct, is syntactically enforced. This is quite reasonable since no demonstration of correctness is possible without these and there are no facilities for mechanically generating them.
- Program statements may be interspersed with first order predicate formulas or assertions, the syntax of which differs from that of the Boolean expressions allowed in guarded commands in that the following are allowed only in assertions:
  - the implication operator,  $\Rightarrow$ ;
  - quantified expressions involving the quantifiers NUMBER, SUM, PRODUCT, FORALL, and EXISTS;
  - explicit reference to arrays and records that have been changed in one or more domain points, using the notation of section 3.2.
- Any program assertion (or bound function) may be optionally labelled with a name. This name may be used as an abbreviation for the formula itself in subsequent program annotations. In addition, one may introduce parameterized *predicate abbreviations* that may be used (with suitable arguments) in any program annotations. Predicate abbreviations are discussed in Chapter 5.

- Quantified expressions follow the syntactical pattern outlined in section 2, with the additional requirements:
  - that the type of the quantification variable be explicitly stated and must be one of INTEGER, CHARACTER, or BOOLEAN;
  - that no more than one dummy variable is allowed;
  - that the numerical quantifiers SUM, PRODUCT, and NUMBER be used only with finite ranges (see section 4.2.6 below), which is enforced by limitations on the syntax of the range predicate.

The grammar given in Appendix B was transformed into an equivalent LL(1) grammar to facilitate construction of a recursive descent (predicative) parser. The desire to obtain a grammar that is LL(1), required no noteworthy deviations from the syntax used in the literature.

In Appendix E, a solution for “Rubin’s problem” (see section 3.5) is shown in the syntax accepted by the parsing component of the prototype implementation.

#### 4.2.2 Representing and Manipulating Expressions

Generating preconditions and verification conditions require only very simple manipulations on the expressions involved. Apart from performing syntactic and basic semantic checking (such as type analysis), the only noteworthy manipulations are:

- renaming of bound variables in quantified expressions and local variables when calculating a precondition for a local block, e.g. in determining

$$wp([[\text{var } x : T_1, y : T_2 \mid S]], R(x, y));$$

- textual substitution as found in

$$R_{x+1, y \text{ div } 2, z=y}^{x, y, z};$$

- construction of an expression from subexpressions, e.g. when using the disjunction of the guards,  $GG$ , a bound function,  $t$ , and an invariant,  $P$ , of a loop to form the verification condition

$$(P \wedge GG) \Rightarrow (t > 0).$$

A brief discussion of suitable data structures for the representation of expressions subject to such operations, as well as the handling of variable renaming during textual substitution and precondition calculations for local blocks, follows.

### **Suitable data structures**

The choice of data structures for the representation of expressions is one of the most significant decisions to be made in constructing mechanical tools for program development using formal methods. The speed at which manipulations can be performed may be the determining factor for a user whether or not to abandon paper and pencil in favour of a mechanical system. Expression simplification and theorem proving operations typically have the greatest impact by far and data structures should be geared towards providing the greatest possible efficiency for these operations [137]. The prototype implementation has no theorem proving component and the design of its data structures does not contain any specific features for its support.

Handling of expressions in the prototype implementation dictated the following basic requirements:

- The data structures needed to represent expressions from program text and annotations, should be efficiently constructable during parsing.
- The data structures used for expressions should allow the basic manipulations listed above to be carried out as efficiently as possible.
- Data structures used for expressions should not make undue demands for space as this may become a critical resource in processing even relatively small programs:
  - even small programs sometimes require manipulation of long and complicated assertions; of which more are generated when calculating preconditions and generating proof obligations;
  - simplification and theorem proving activities typically generate large sets of logical formulae and consequently make significant demands on the available memory;
  - other operations such as parsing and editing will simultaneously require space for their data structures.

- Since no theorem proving or simplification currently takes place on the verification conditions generated, their end is simply to be recorded for the programmer's scrutiny. The data structures used for expressions should allow this information to be easily converted into a user appreciable format.

After considering these requirements, it was decided that representing expressions as binary tree structures presented a good solution. The operations called for, can be implemented by simple tree traversal, insertion, and deletion operations. Because one expression may be required for different purposes in a number of verification conditions, it was found that copying of expressions happens very frequently. This is a costly operation that will be well worth optimizing if it continues to play as important a role in simplification and theorem proving operations.

Instead of labeling individual nodes that represent variables with the name and type of the variable, expression trees reference a symbol table, that is constructed during program parsing, for information on variables. This makes renaming of bound variables very fast and because the symbol table also contains scope information, references to local and global variables with the same name are always easy to distinguish. The symbol table is currently implemented as a binary search tree. Its performance could be enhanced by changing this to a more sophisticated representation.

### Variable renaming

Calculation of the weakest precondition of an assignment construct involves textual substitution in the postcondition. If such a postcondition involves quantified expressions, correct substitution requires that variable capture within the scope of a quantifier be avoided by renaming bound variables. For example, in

$$(\forall k : R(k) : (\exists x : S(x) : P(k, m, n, x)))_{x+k, x \rightarrow j}^{m, n} \quad (10)$$

the bound variables  $k$  and  $x$  will have to be renamed, to prevent references to variables  $x$  and  $k$ , introduced into this scope by substitution, from being interpreted as references to the bound variables with the same names.

For the calculation of

$$wp([[\text{var } x : T_1, y : T_2 \mid S]], R(x, y)) \quad (11)$$

all references to the local variables  $x$  and  $y$  in the construct  $S$  have to be replaced by fresh variables, i.e. variables that do not occur free in  $R$  or in  $S$ .

Two useful options for the renaming of variables are open to a mechanical system:

1. Prompt the programmer for a suitable renaming.
2. Construct a new name by appending a special prefix or suffix to the name of an existing variable.

The first option is attractive in that a program developer retains complete control over the identifiers used in verification conditions associated with a program. Presumably, this makes verification conditions easier to understand. After implementing this capability, it was found to be very inefficient. The verification condition generator has no guarantee of the suitability of a replacement name supplied by a programmer for its intended purpose. If, for example, the bound variable  $k$ , in (10) is renamed to  $j$ , it will probably have to be renamed again. One may argue that programmers will be judicious in their choices when renaming variables, but a mechanical procedure will have to check whether this is indeed so. In the case of (11) this involves scanning both  $R$  and  $S$ , possibly repeatedly, to ensure that names supplied to resolve clashes do not already appear there.

The second alternative has the advantage that it may be efficiently implemented. Since variable names are altered only slightly, the connection with the program text remains apparent. Using a special character such as “#” to prefix or suffix names that have to be altered is impractical, because the same identifier may be used any number of times in different contexts. In [63] different uses of the same identifier are distinguished by using the program line number on which an identifier is introduced as a suffix for all occurrences of that identifier in verification conditions. I have implemented a similar scheme that renames an identifier  $x$  to  $n\textcircled{x}$ , where  $n$  is a unique number, in cases where ambiguity arises. This notation uses a “ $\textcircled{\phantom{x}}$ ” to separate the identifier and number, instead of the more common “.”, to prevent confusion with a record identifier qualified by a field name. Line numbers are not used, because

they need not be unique—nothing prevents the introduction of the same identifier as bound variable for different quantifiers on the same program line.

Since each program block and each quantified expression represent a new scope level, identifiers may be distinguished by expanding them to include a reference to the level at which they were introduced. As a program is parsed, the parser numbers the scope levels sequentially in the order that they are encountered and these unique numbers are used to distinguish identifiers where ambiguity arises. This is a similar arrangement to the use of module or procedure names to identify names declared in different contexts e.g. designating a variable `IOcheck` declared in a module `FIO` as `FIO.IOcheck` in Modula-2 or Ada (see [53]). If constructs such as procedures and modules are added to the guarded command language, the current scheme provides the scope to accommodate these as in the above example.

Because the representation chosen for expressions uses a symbol table when referring to variables, expanding all references to a certain variable to its full, unique form is accomplished by a very simple change to the symbol table entry for that variable.

Name substitutions within program constructs, as used for local blocks, require no additional or different procedures than those for renaming in logical formulae.

### **4.2.3 Variable Initialization**

What is the value of a variable after its declaration? Different approaches to this question include considering new variables to have a special “undefined” value upon their creation, or implicit initialization with a neutral value such as zero for all integer variables and “true” for all Boolean variables. None of these approaches is ideal. Our approach to this matter distinguishes between global and local variables. Where no treatment of input and output mechanisms is given, it is customary to assume that the initial state of a computation is represented by the values that global variables initially possess. Application of the weakest precondition calculus starting from a desired postcondition, will eventually produce a precondition stating a suitable initial state, and thus, suitable initial values for the global variables of the computation.

Variables introduced in local blocks present a different scenario. A practical way in which

to illustrate this, is to consider a typical situation that arises in proving the correctness of a local block constructed during a program development:

$$\{pre\}B : \llbracket \text{var } x : T \mid S \rrbracket \{post\}$$

We wish to show that the (weakest) precondition of the block,  $B$ , with respect to the postcondition,  $post$ , is a logical consequence of the precondition,  $pre$ , i.e.

$$\llbracket pre \Rightarrow wp(B, post) \rrbracket$$

It is clear that  $pre$  has to be independent of the local variable  $x$  and thus that we cannot prove the above unless  $wp(B, post)$  is also independent of  $x$ , i.e. for any condition  $R$ ,

$$\llbracket (\forall v :: wp(B, R)_v^x \equiv wp(B, R)) \rrbracket$$

or, in terms of our semantics, for any fresh variable  $y$ :

$$\llbracket (\forall v :: wp(S_y^x, R)_v^y \equiv wp(S_y^x, R)) \rrbracket \quad (12)$$

Unfortunately the weakest precondition semantics that we have assigned to local blocks does not guarantee that this condition will hold. In particular, it fails to hold iff  $x$  is not initialized by  $S$ . Various ways of addressing this problem were considered.

### An unsatisfactory solution

At a first glance, the simplest approach seems to be not requiring local variables to be initialized explicitly, thus assuming that newly declared local variables are of arbitrary value. The only explicit requirement is that the initial value of the variable be of the appropriate type. This sounds elegant enough; a view that is borne out by the simple change required to build this into the semantics of the local block:

$$\llbracket wp(\llbracket \text{var } x : T \mid S \rrbracket, R) \equiv (\forall y : y \in T : wp(S_y^x, R)) \rrbracket \quad (13)$$

Thus we are now requiring that  $S$  establish the postcondition  $R$  regardless of the initial value of  $x$ . This is similar to the approach adopted in the definition of the verification oriented

language, Euclid [158] and is in accordance with the semantics of the local block construct as given in [130].

Such semantics for the local block construct is however in violation of our semantic framework. Because we allow infinite data types, the predicate transformer thus defined is not or-continuous and would therefore allow unbounded nondeterminism, e.g.  $[[\text{var } y : T \mid x := y]]$  sets  $y$  to any integer—an unbounded number of choices!

As a result of the above, the requirement that all variables be initialized before they are referenced, is unavoidable.

### Dijkstra's regimen

In [38, Chapter 10] Dijkstra suggests that the guarded command language be restricted in order to ensure that variables are explicitly initialized before being referenced. His proposal includes syntactically recognizable initializing statements as well as a rigid discipline for their use.

Dijkstra's regimen requires that all private and inherited identifiers be listed upon block entry. The textual scope of a variable private to a block extends from the start of the block to its end, with the exception of nested blocks that do not inherit it. This textual scope is divided into the "passive scope" where it may not be referenced and it does not form part of the state space and the "active scope" where the variable may be referenced. The passive and active scopes of a variable are separated by an initializing statement that has to be placed in such a way that, independent of values of guards:

1. exactly one initializing statement for that variable will be executed inside its textual scope;
2. no statement from the active scope of the variable can be executed between block entry and the execution of the initializing statement.

Though the measures proposed by Dijkstra would accomplish our goal, they are quite complicated. The treatment of array variables is also less than ideal. Dijkstra does not propose

any program construct that can assign values to individual points in the domain of an array without explicitly listing them, i.e. for array variables  $ar$  and  $br$  we cannot use

$$ar := br$$

but only

$$ar[0], ar[1], ar[2], ar[3], ar[4] := br[0], br[1], br[2], br[3], br[4]$$

Neither can we use the loop

$$\begin{array}{l} i := 0 \\ \text{do } i \neq n \rightarrow ar[i] := br[i]; \\ \quad i := i + 1 \\ \text{od} \end{array}$$

to initialize  $ar$ , since no initializations are allowed to take place inside loops. If we consider the domain of an array variable as fixed (as we do here) this regimen, in effect, limits the array variables used in local blocks to those with domains that are small enough to list exhaustively.

### Hemerik's suggestions

In [81] explicit initialization of local variables is enforced by the imposition of the following syntactic condition:

$$x \in \text{INIT}(S) \vee x \notin \text{USE}(S)$$

for all variables  $x$  declared in a local block  $[[\text{var } x : T \mid S]]$ .

The set  $\text{USE}(S)$  can be informally described as the set of all variables occurring in an expression in  $S$  (or in its interspersing annotations), while  $\text{INIT}(S)$  can be seen as the set of all variables assigned to by every possible execution of  $S$  and not used in any expression before being assigned a value (see [81, Chapter 4] for definitions). Constructing these sets is a straightforward syntactic procedure.

An advantage of using Hemerik's proposal is that it does not require any syntactic extensions to the guarded command language. In particular, we do not need syntactically distinguishable initialization statements. Its main disadvantage is that it displays the same shortcomings as Dijkstra's discipline in the treatment of arrays with fixed domains.

### Final considerations

Our aim remains to ensure that the equivalence (12) above, holds for every local block in a guarded command program. Why not generate this as an additional verification condition to be satisfied by every local block? In the first place it is preferable to enforce such conditions without the need for theorem proving. Secondly, the fact that we will never explicitly generate the weakest precondition of a local block containing loop constructs, makes this condition practically impossible to check.

The condition (12) merely formalizes the notion that the weakest precondition of a local block is independent of any local variables declared in that block. Is it possible to ascertain this by a syntactic check on logical formulae? Because we will rarely generate the weakest precondition itself, we have to strengthen this question to: Can we establish that the weakest precondition is independent of local variables by some syntactic check on any given or generated precondition (formula)? The answer is no.

```

[[
var
  i    :    integer;
  ar   :    array[0..n] of integer
|
  i := 0;
  invar {( $\forall k : 0 \leq k \wedge k < i : ar[k] = br[k]$ )}
  bound { $n - i$ }
  do   $i \neq n \rightarrow ar[i] := br[i];$ 
       $i := i + 1$ 
  od
  R : {( $\forall j : 0 \leq j \wedge j < n : ar[j] = br[j]$ )}
  :
]]
{post}

```

Figure 5: Making a local copy of an array.

Using the given loop annotations in the process of calculating a precondition for the local block in figure 5, we obtain the following:

$$(\forall k : 0 \leq k \wedge k < 0 : ar[k] = br[k]).$$

This precondition is a tautology and thus independent of local variables, but without the application of the relevant simplification procedure, this cannot be deduced.

We are forced to conclude that the only practical way of ensuring proper initialization of local variables, is to augment our guarded command language in such a way that syntactic checks, such as those proposed by Dijkstra or Hemerik, are applicable. Hemerik's suggestions seem to be the most promising. A syntactically distinguishable initialization statement for arrays will have to be devised to allow the use of arrays with large, fixed domains.

Another solution to this problem would be to accept the local block semantics stated in equation (13). This requires that we disregard the or-continuity healthiness condition for weakest precondition predicate transformers. Chapters 6 and 7 describe a calculus where this situation exists.

The prototype implementation does not contain a solution to the problem of uninitialized variables. For each local block, a syntactic check is performed to ensure that the generated or given precondition does not contain any references to local variables. As indicated above though, such a check is in general not strong enough.

### Representation of initial values in program annotations

It is quite customary to introduce so-called *logical constants*, *ghost variables* or *specification variables* that do not form part of the state space to represent initial values of variables in program specifications and annotations. In [66] logical constant identifiers are distinguished from program identifiers by the convention that capital letters are reserved for the former and small letters for the latter, e.g.

$$x : [x = X, x > X].$$

In [125] 0-subscripted variables are reserved for logical constants that represent initial values of program variables.

As pointed out in [66] verification conditions have to be proven to be universally true (tautologies), e.g. a verification condition

$$(x = X) \Rightarrow wp(x := x + 1, x > X)$$

involving the logical constant  $X$ , should be understood as

$$[(\forall X :: (x = X) \Rightarrow wp(x := x + 1, x > X))].$$

Thus a verification condition is universally quantified over all its logical constants. In effect, the logical constants are a special type of local variable for use only in program annotations. In a formal environment, a mechanism for the declaration of logical constants should be made available to allow semantic checks on their usage and explicitly delimit their scope. Conventions such as the use of capital letters or 0-subscripts, that are inconvenient when one wants to refer to the value of a variable in more than one previous program state (see e.g. [121, page 51]), then become superfluous.

Apart from the exception mentioned below, the prototype implementation does not include a mechanism for the formal introduction of general logical constants and thus precludes their use. A good candidate for such a mechanism is given as part of the refinement calculus in [121] and is discussed in Chapter 7.

It is interesting to note that the omission of logical constants cause more than a mere inconvenience in the annotation of guarded command programs. In the Fundamental Invariance Theorem for Loops, a logical constant is used to represent the initial value of the bound function in the correctness proof. In annotating nested loops, one has no choice but to refer to this logical constant in stating the effect of executing an inner loop on the bound function of the outer loop. This has prompted the inclusion of a mechanism for introducing a specific logical constant, representing the initial value of a bound function, in guarded command program annotations. More details follow in section 4.3.1.

#### 4.2.4 A Meaningful Type System

At present the type system is very spartan. In order to provide more flexibility, a greater range of expression and better abstraction facilities, it seems desirable to extend the range of available data types, enhance the flexibility of types such as arrays and records and allow user-defined types.

The prototype implementation of array and record types is limited in a number of ways

(see below). These limitations will be accentuated if user-defined procedures and functions are included in the guarded command language. For complete flexibility, the suitability of polymorphic types, such as discussed in [117, 77], for inclusion in the type system of a guarded command language, is worth investigating. Since type inferencing is naturally associated with polymorphic types, this also provides scope for omitting type descriptors when introducing variables, e.g. in quantified expressions.

Using data types that are well suited to a particular problem domain is crucial in examples done by hand, but is difficult in systems that provide mechanical support, because of the need for complete formality. Including more standard data types in the guarded command language provides one way of addressing this issue. Some candidates for inclusion are character strings, integer subranges, rational numbers, real numbers, pointers, enumerated types, sets, and sequences. Some of these types, e.g. real numbers, are difficult to axiomatise. Pointers are contentious and some suggestions have been made for mechanisms allowing the creation and manipulation of dynamic variables without the explicit use of pointers, e.g. [142, 153]. Treatments of pointers in weakest precondition semantics may be found in [127, 126, 129, 17].

If mechanical proof support is to be offered, the inclusion of a wealth of data types complicates proof procedures immensely, because of the number and the extent of the theories involved. Extending the current type system to include facilities for user-defined types and subtypes, should also be able to contribute towards the use of types most suitable for an algorithm under development. Ultimately, the use of abstract data types seems ideal. This allows a high degree of flexibility without requiring excessive extensions to the standard type system. A programmer may use the exact level of data abstraction required and gains convenience of expression and the advantage of shorter, less complicated verification conditions. Data abstraction is investigated further in Chapters 5 and 7. Again, the implications for mechanical proof support are far reaching. Under such circumstances, a mechanical prover should support user building of (reusable) theories for new data types, ensuring that no logical inconsistencies result (see discussion in [70, page 1064–1066]), and provide mechanisms for their efficient inclusion in simplification and reasoning procedures.

Mechanical support for program development in the weakest precondition calculus will benefit from a thorough investigation into and probably significant extensions to the type system

used in guarded command programs. This is necessitated by the need for expressiveness and flexibility in the face of complete formality of syntax and semantics. The nature and spirit of Dijkstra's calculus suggests that we stand to gain the most by introducing suitable abstraction mechanisms—thus data abstraction facilities deserve special attention. Changes to the type system will have a definite effect on the requirements for a mechanical proof system to support program development.

### **Handling arrays and records**

The inclusion of arrays and records in Dijkstra's calculus allows the formulation of many non-trivial algorithms in a natural way. Their use also leads to a number of complications, including:

- program annotations and verification conditions involving arrays and records are frequently more complex and more difficult to read;
- special syntax has to be introduced to handle the initialization of arrays;
- textual substitution as well as rules for the correct use of multiple assignment (and procedure call) constructs require reformulation to take into account the possibility of aliasing;
- the underlying logic of the specification language is complicated by issues such as quantification over arrays and the treatment of undefined values;
- due to the above, proofs of verification conditions may also become more complicated.

The prototype implementation imposes certain limitations on the use of arrays and records. The following discussion highlights the most noteworthy, offering suggestions for future extensions:

**No structured constants:** It is useful to be able to introduce arrays and records whose values are not to be changed as constants. No specific problems are foreseen in lifting this restriction.

**Fixed, finite, integer domains:** The fact that array domains are limited to integers is not essential and expansion to include other domain types should pose no problems. Having fixed array domains is in accordance with most traditional treatments of arrays in programming languages, but in contrast to that of [38]. As mentioned in section 4.2.3 above, special arrangements are necessary for the initialization of arrays with large fixed domains, if correctness of local blocks is to be fully verified. Introducing suitable operators for domain manipulation and examination, such as those listed in section 3.2, the fixed domain requirement would be simple to remove. The requirement of finite domains is necessary to ensure that checking two arrays for equality, remains decidable.

**Explicit array domain bounds:** This limitation may be removed if primitive functions representing the lower and upper domain bounds are introduced (see section 3.2). These functions will have to be generalized to extract the domain bounds of higher order (multidimensional) arrays (see [33, page 209] for an example of how this can be done). Considering domain bounds as part of the value of an array data object (as in [33, 68]) will allow construction of more flexible programs and should be a worthwhile extension. Explicit bounds may then be considered as stating limits on the array domain, and thus as details of implementation.

**Few operators:** Most of the array operators introduced by Dijkstra change the domain of arrays. These are all candidates for future extensions. Another useful suggestion is found in [66, Chapter 5], where a notation is suggested for the restriction of an array to a section of its full domain. Given suitable integer expressions  $e_1$  and  $e_2$ , satisfying  $e_1 \leq e_2 + 1$ , the notation  $ar[e_1..e_2]$  denotes array  $ar$  restricted to a domain bounded by the values of  $e_1$  and  $e_2$ .

Dijkstra proposes only two operators, namely **swap** and **alt**, that change just the function values of an array. Of these, we use the operator **alt**, for changing the function value in one domain point. We adopt the well known notation  $ar[i] := x$ , instead of the more cumbersome  $ar : \text{alt}(i, x)$  (see section 3.2). Other useful operators, such as **swap**, could also be added, as necessary. We do not allow the assignment  $ar := br$  for array variables  $ar$  and  $br$ , because of its potential for misleading a programmer about its economics where the domains of  $ar$  and  $br$  are large (see [38, page 102]). Dijkstra

and Gries both allow the assignment of enumerated constants to array variables to simultaneously change the value of the array in more than one domain point. Though I feel that the multiple assignment construct is more flexible and sufficient for current purposes, the use of enumerated constants does allow a more compact notation.

In [66, Chapter 5] relational operators such as  $=$ ,  $\neq$ ,  $<$ ,  $>$ ,  $\leq$  and  $\geq$  are applied to arrays. They denote component-wise comparisons and are proposed as abbreviations for certain frequently used program assertions, e.g.

$$ar < x$$

instead of

$$(\forall i : 0 \leq i \wedge i < n : ar[i] < x).$$

Such abbreviations provide abstraction when constructing a program specification and may aid program understanding. On the other hand, these abbreviations can lead to confusion when performing manipulations, since  $\neg(ar < x)$  is in general not equivalent to  $ar \geq x$  (see [66, page 93] for more examples). If abbreviations of this kind are to be allowed, it seems desirable to expand them to their full form during program development manipulations—possibly with the option of contracting them again when the program has been completed. Additionally, it is strongly recommended that distinct relational operator symbols are introduced to avoid confusion. To allow mechanical manipulation and simplification of formulas containing such operators, the underlying calculus will have to be extended with rules for their manipulation.

New array operators may provide the means for more succinct, elegant formulation of program properties, but their introduction requires careful consideration. One has to ensure that the increased burden of providing effective measures to handle them in formal manipulations, is justified. Operators specific to a particular problem domain should be introduced by other means such as user-defined procedures and functions or as self contained libraries.

As for arrays, the only operators currently implemented on records are alteration of the record value in a single domain point, or in a number of specified domain points (multiple assignment).

**No array quantification:** From a logical point of view, quantification over arrays leads to the realm of weak second order logic. An example of such usage may be found in [33]. One of the consequences of allowing quantification over arrays is that primitive functions for extracting the upper and lower domain bound from an array value (such as `hib` and `lob` in section 3.2) become imperative. Without them the behaviour of quantifiers would be different. For example,

$$(\forall(n : \text{integer}) :: (\forall(ar : \text{array}[1..n] \text{ of integer}) :: P(ar, n))) \quad (14)$$

would differ from

$$(\forall(ar : \text{array}[1..n] \text{ of integer}) :: (\forall(n : \text{integer}) :: P(ar, n)))$$

Using functions `lob` and `hib`, the assertion (14) could be written as:

$$\begin{aligned} &(\forall(n : \text{integer}) :: \\ &(\forall(ar : \text{array of integer}) : ar.lob = 1 \wedge ar.hib = n : P(ar, n))) \end{aligned}$$

More details are available in [33, Chapter 7].

**Higher order arrays and records:** In [66, Chapter 5] Gries discusses arrays of arrays. This extends the notion of an array as an integer, Boolean or character valued function; allowing arrays themselves as array ranges. As an example consider the array variable declared as follows:

$$ax : \text{array}[0..10] \text{ of array}[1..5] \text{ of integer}$$

Based on this declaration, `ax` refers to the complete array, `ax[0]` refers to the array associated by `ax` with the domain element 0, while `ax[i][j]` (for suitable integer expressions `i` and `j`) refers to a single integer.

This view is not fully supported in the prototype implementation. Our view corresponds to the multidimensional arrays found in languages such as ALGOL, FORTRAN, and PL/1 as well as the PL/CV verification system [33]. An  $n$ -dimensional array is treated as a function from a finite set of consecutive integer  $n$ -tuples (according to the lexicographic ordering) to a set of simple values such as integers or characters. Assuming the array declaration as above, one may refer to individual function values using the notation

$ax[i][j]$ , while  $ax[i]$  has no meaning in this context. Multidimensional arrays provide a less complicated alternative to the “arrays of arrays” approach. The latter is, however, more flexible and provides suitable notation for describing the effect of some of the operators that change all the function values of an array.

For the other higher order objects such as arrays of records and records of arrays, the situation is similar.

#### 4.2.5 Multiple Assignment to Compound Variables

In section 3.1, the multiple assignment construct is described. This discussion focuses on assignment to simple, distinct variables and the notion of simultaneous textual substitution was found to describe its semantics adequately. For compound variables, such as arrays and records, the multiple assignment construct is generalized to the form

$$x_1 \circ s_1, \dots, x_n \circ s_n := e_1, \dots, e_n$$

where the  $x_i$  are variable identifiers,  $s_i$  are appropriate selectors and each  $e_i$  is an expression of a type compatible with that of  $x_i \circ s_i$ .

The semantics of this construct is complicated by the possibility of aliasing among the  $x_i \circ s_i$ . Two variable references  $x \circ s$  and  $y \circ t$ , comprising identifiers  $x$  and  $y$  and selectors  $s$  and  $t$ , are *aliased* if one is an initial segment of the other, i.e. the latter references part of the value of the former [67]. From the following examples, it should be clear that aliasing is in general not a syntactic property:

$rec.m$	and	$rec$	are aliased;
$rec.m$	and	$rec.n$	are not aliased;
$ar[i]$	and	$ar[j]$	are aliased only if $i = j$ ;
$ar[i]$	and	$ar[i - 1]$	are not aliased;
$i$	and	$ar[i]$	are not aliased

Compliance with the semantics of the original multiple assignment construct requires defining the weakest precondition of the general multiple assignment as the conjunction of the weakest

preconditions of all possible orderings of assignments [65]:

$$wp(ar[x_1], \dots, ar[x_n] := e_1, \dots, e_n, R) \equiv \bigwedge_{(i_1, \dots, i_n)} R_{ar(x_1:e_1, \dots, x_n:e_n)}^{ar}$$

where  $(i_1, \dots, i_n)$  represents a permutation of  $(1, \dots, n)$ .

This formidable definition is due to the requirement that the components of a multiple assignment be executed in any order. Since it is not particularly palatable, we simplify the definition by eliminating its nondeterminism. As in [66], we specify the semantics of the general multiple assignment construct as follows:

**Syntax:**

$$x_1 \circ s_1, \dots, x_n \circ s_n := e_1, \dots, e_n \quad (15)$$

where the  $x_i$  are variable identifiers,  $s_i$  are appropriate selectors and each  $e_i$  is an expression of a type compatible with that of  $x_i \circ s_i$ .

**Operational Semantics:** The variables specified by the  $x_i \circ s_i$  are determined and the expressions  $e_i$  are evaluated to yield values  $v_i$ . This is followed by the assignment of each  $v_i$  to the  $x_i \circ s_i$  of corresponding subscript in *left to right order*.

**Predicate Transformer Semantics:**

$$\begin{aligned} [wp(x_1 \circ s_1, \dots, x_n \circ s_n := e_1, \dots, e_n, R)] \equiv \\ (\forall i : i \geq 1 \wedge i \leq n : \text{domain}(x_i \circ s_i) \text{ and } \text{domain}(e_i)) \text{ and } R_{e_1, \dots, e_n}^{x_1 \circ s_1, \dots, x_n \circ s_n} \end{aligned} \quad (16)$$

where the predicates  $\text{domain}(x_i \circ s_i)$  and  $\text{domain}(e_i)$  characterize the set of states in which the expressions  $x_i \circ s_n$  and  $e_i$  can be evaluated without causing failure (see Appendix C for a definition).

To make the latter meaningful we assume the following definition [66, page 129] of  $R_{\bar{e}}^{\bar{x}}$ , where  $\bar{x}$  is a list of identifiers, each concatenated with a selector, and  $\bar{e}$  is a list of expressions of appropriate types, containing the same number of elements as  $\bar{x}$ .

1. If  $\bar{x}$  is a list of distinct identifiers (thus all selectors in  $\bar{x}$  are equal to  $\varepsilon$ ),  $R_{\bar{e}}^{\bar{x}}$  denotes conventional textual substitution.

2. If  $a$  and  $b$  are distinct identifiers,

$$[R_{\bar{e},f,g,\bar{h}}^{\bar{x},aos,bot,\bar{y}} \equiv R_{\bar{e},g,f,\bar{h}}^{\bar{x},bot,aos,\bar{y}}]$$

Thus, adjacent reference expression pairs may be exchanged, provided they begin with different identifiers.

3. If identifier  $a$  does not begin any of the  $x_i$ ,

$$[R_{e_1,\dots,e_n,\bar{f}}^{aos_1,\dots,aos_n,\bar{x}} \equiv R_{a(s_1:e_1,\dots,s_n:e_n),\bar{f}}^{a,\bar{x}}]$$

Thus, multiple assignments to subparts of a compound object  $a$  can be seen as a single assignment to  $a$ .

Thus we sacrifice the more abstract nondeterministic definition in favour of its more workable deterministic counterpart, shown above. Under these circumstances, one may well ask whether the use of the general multiple assignment is still justified. It is suggested, in [65], that its use can still aid program understanding as well as precondition calculations, by being more concise and straightforward than an equivalent sequence of simple assignments. The multiple assignment

$$p, b[p], b[i] := b[p], b[i], p$$

and equivalent assignment sequences [65]

$$\begin{array}{ll} t := p; & t := p; \\ p := b[t]; & t2 := b[i]; \\ b[t] := b[i]; \text{ and } & t3 := b[p]; \\ b[i] := t & b[i] := t; \\ & b[t] := t2; \\ & p := t3 \end{array}$$

illustrate this point.

In the prototype implementation, the general multiple assignment construct is treated deterministically, according to the semantics outlined here, while assignments to simple variables are still treated nondeterministically, according to (3). Thus, a multiple assignment construct

is not accepted if it contains more than one assignment to the same simple variable (identifier). This is a debatable decision. One may argue that it is unfortunate to distinguish between simple and compound variables here, after taking trouble to hide their differences with a functional view. Another valid objection is that if a construct has different semantics for different data objects, it would be less confusing to separate the distinct cases notationally as well—resulting in separate language constructs. I feel that different assignments to the same simple variable are clarified by expressing them as separate simple assignments. In the end the decision to view reassignment of a simple variable within a multiple assignment as an abuse of notation, is a personal one. It could easily be altered if a more pleasing alternative presents itself.

Using definition (15) or (16), the weakest precondition of an assignment construct involving arrays is simple to calculate. In contrast, it can be rather troublesome to simplify; typically involving case analysis as shown in the following:

$$\begin{aligned}
 wp(ar[i] := 7, ar[i] \geq ar[j] \wedge ar[j] \geq ar[k]) \\
 &\equiv (ar[i] \geq ar[j] \wedge ar[j] \geq ar[k])_{ar([i]:7)}^{ar} \\
 &\equiv ar([i]:7)[i] \geq ar([i]:7)[j] \wedge ar([i]:7)[j] \geq ar([i]:7)[k] \\
 &\equiv (i \neq j \wedge i \neq k \wedge 7 \geq ar[j] \wedge ar[j] \geq ar[k]) \vee \\
 &\quad (i = j \wedge i \neq k \wedge 7 \geq 7 \wedge 7 \geq ar[k]) \vee \\
 &\quad (i \neq j \wedge i = k \wedge 7 \geq ar[j] \wedge ar[j] \geq 7) \vee \\
 &\quad (i = j \wedge i = k \wedge 7 \geq 7 \wedge 7 \geq 7)
 \end{aligned}$$

In general, considering an assignment  $ar[j] := e$  and a postcondition containing  $n$  references  $ar[i_1], \dots, ar[i_n]$  involving array  $ar$ , the corresponding precondition will contain  $n$  references  $ar(j:e)[i_1], \dots, ar(j:e)[i_n]$ . Such a precondition could be broken down into  $2^n$  cases. Understanding the precondition may thus require effort exponential in the number of references in the postcondition involving the array being changed. This complexity seems to be unavoidable, as it is rooted in dynamic aliasing—two different variables  $i$  and  $j$ , used as array selectors, can yield two different names  $ar[i]$  and  $ar[j]$  for the same array value and these names change dynamically as  $i$  and  $j$  change. To control this complexity, Gries suggests that the number of references to the array be kept to a minimum and that enough restrictive

information be maintained in program assertions to allow the treatment of many subcases in a similar fashion.

#### 4.2.6 Quantified Expressions

As explained in section 4.2.1 and Appendix B, the prototype implementation allows quantified expressions of the form

$$(\text{quantifier } (\text{dummy} : \text{type}) : \text{range} : \text{term})$$

where

**quantifier** is one of NUMBER, SUM, PRODUCT, FORALL, and EXISTS;

**dummy** is a single variable;

**type** indicates the type of the dummy variable and is one of the primitive types INTEGER, CHARACTER, or BOOLEAN;

**range** is a(n) (optional) Boolean expression;

**term** is an expression of a type matching the specific quantifier.

We highlight some of the interesting issues surrounding quantified expressions as currently implemented:

**Type of the dummy variable:** As stated in section 2, the type of a dummy variable is traditionally omitted if it is obvious from the context. From the point of view of parsing and type analysis, the introduction of a dummy variable is similar to the declaration of a local variable. Allowing the freedom of omitting type information would require a type inferencing mechanism, which is not available currently.

**Multiple dummy variables:** The fact that multiple dummy variables are currently not allowed, is merely a syntactic restriction. It would be quite a simple matter to extend the prototype grammar and parser to accommodate this variation.

**Special ranges:** The requirement that the range must be a Boolean expression excludes abbreviations such as

$$0 \leq j < n$$

for a dummy variable  $j$ , that abounds in the literature. This extension should, however, be easy to incorporate.

For the numerical quantifiers SUM and PRODUCT we require additionally, following [40, page 141], that the equation

$$\text{range} \wedge \text{term} \neq 0$$

in the dummy variable has a finite number of solutions. This excludes expressions, such as

$$(\forall i : 0 \leq i : (-1)^i),$$

that do not denote proper integer expressions. This requirement is satisfied by limiting the dummy variable to a finite range. Syntactically, the range expression is limited to the conjunction of two Boolean expressions, one establishing the lower and the other the upper bound for the range of the dummy variable, e.g.

$$0 \leq j \wedge j < n$$

(also see Appendix B). For the quantifier NUMBER, the equation

$$\text{range} \wedge \text{term}$$

in the dummy variable is similarly required to have a finite number of solutions. This restriction is achieved as for SUM and PRODUCT.

## 4.3 Proof Obligations

### 4.3.1 Generating Verification Conditions

We now outline the well-known procedure for batch generation of verification conditions from an annotated guarded command program, as implemented in the prototype implementation.

The first step is to parse a guarded command program and its annotations, performing the necessary syntactic and semantic checks. At the same time data structures, containing the essential information for verification condition construction, are built. Most substantial programs consist of a sequence of program statements. Following the definition of the weakest precondition transformer for sequencing, verification conditions are generated in backwards order relative to the order of these statements. Thus for a simple program of the form  $S_1; S_2$ , annotated with a precondition and a postcondition as follows:

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

$R_1$ , such that  $R_1 \equiv wp(S_2, Q)$  (or merely  $R_1 \Rightarrow wp(S_2, Q)$ ) will be calculated first, followed by  $wp(S_1, R_1)$  (or  $R_2$ , such that  $R_2 \Rightarrow wp(S_1, R_1)$ ). One verification condition will be generated, i.e.

$$[P \Rightarrow wp(S_1, R_1)].$$

The universal quantification, shown explicitly here, is implicit in actual conditions.

Currently all verification conditions are recorded in a text file in the order that they are generated. No algebraic simplifications are carried out.

### Internal annotations

A programmer is allowed to place assertions between sequentially composed program statements. These assertions provide a way of decomposing a complex verification step into a number of more manageable steps. For a program of the form  $\{P\}S_1\{Q\}; S_2\{R\}$ , the following two verification conditions will be generated:

$$[Q \Rightarrow wp(S_2, R)]$$

and

$$[P \Rightarrow wp(S_1, Q)].$$

The prototype implementation does not allow annotations of the form:

$$\{P\}S_1\{Q_1\}\{Q_2\}; S_2\{R\}.$$

However for compound statements such as local blocks, alternative and repetitive constructs, the following situations may arise:

$$\{P_1\} \parallel [\text{var } \dots \mid \{P_2\} S \{Q_2\}] \parallel \{Q_1\}$$

$$\{P_1\} \text{ if } B \rightarrow \{P_2\} S \{Q_2\} \dots \text{ fi } \{Q_1\}$$

$$\{\text{invar } P_1\} \text{ do } B \rightarrow \{P_2\} S \{Q_2\} \dots \text{ od } \{Q_1\}$$

In each of these cases, the following two verification conditions will be generated (in addition to any others):

$$[P_1 \Rightarrow P_2]$$

and

$$[Q_2 \Rightarrow Q_1].$$

**Alternative constructs** Weakest precondition calculations are only conducted for alternative constructs that are not already annotated with a precondition. If a precondition is supplied with such a construct, Theorem 3.1.0.1 is applied, generating two verification conditions (see Theorem 3.1.0.1) which guarantee that the given precondition is at least as strong as the weakest precondition.

**Repetitive constructs** Weakest preconditions of loops are not calculated, instead all loops must be annotated with an invariant and a bound function that may be used for verifying its correctness. A typical loop,  $L$ , may be annotated as follows:

$$\begin{array}{l} \{P\} \\ \{\text{invar } X\} \\ \{\text{bound } t \text{ init } T\} \\ L \\ \{Q\} \end{array}$$

Using the Fundamental Invariance Theorem for Loops (Theorem 3.1.0.2), three verification conditions are generated, one of which refers to the logical constant  $T$ . The verification condition

$$[P \Rightarrow X]$$

is also generated.

As mentioned above, the general use of logical constants is not supported in the prototype implementation. A logical constant, representing the initial value of a bound function, does appear in the Fundamental Invariance Theorem for Loops, but it seems as though this constant could be generated by the system. Unfortunately, this is not always possible. Nested loops represent an exception.

```

:
{bound  $t_1$ }
 $L_1$  : do
    :
    {invar  $P_2$ }
     $L_2$  : do ... od
    :
    od

```

Figure 6: Nested loops.

Consider the schematic example in figure 6. To show termination of the outer loop,  $L_1$ , we will generate a verification condition stating that each iteration of  $L_1$  decreases the value of the bound function  $t_1$ . All we “know” about the effect of executing the inner loop,  $L_2$ , is contained in its annotations, particularly the invariant  $P_2$ . Thus,  $P_2$  must state the effects of executing  $L_2$  on the value of  $t_1$ . For this we need a logical constant representing the value of  $t_1$  before execution of the inner loop. We also need to ensure that the function of this logical constant is clear to a mechanical verification condition generator, so that it will use the same constant when generating verification conditions for  $L_1$ . The “Rubin’s problem” example of section 3.5 contains an example of correctly annotated nested loops.

In summary: not only does one need to use a logical constant in correctly annotating nested loops, but this constant must be the same as the constant found in one of the verification

conditions generated by using the Fundamental Invariance Theorem for Loops. The only way to accomplish this is to allow the programmer to choose the name for the constant in such a way that it will also be unambiguously identified to the verification system. In the prototype implementation, this is accomplished by stipulating that the phrase *init T*, where *T* is a suitable name for a logical constant representing the initial value of a bound function, must appear with the definition of each bound function.

Appendix E shows the verification conditions generated for the “Rubin’s problem” example of section 3.5.

### **4.3.2 A Suitable Specification Language**

The language used in the literature for expressing annotations and specifications of guarded command programs, is a first-order language allowing references to common data objects such as integers and characters. This language is defined only informally, giving it some flexibility to be extended to accommodate data types, operators or notation used in individual examples. To allow for mechanical calculation of (weakest) preconditions, generation of verification conditions and formal proofs of consistency between specifications and their guarded command implementations, the specification language has to be formalized. Within such a formal setting, a more critical assessment of the specification language is inevitable and some of its shortcomings also become apparent. The design of a specification language is a crucial issue with far reaching implications for many aspects of program development and consistency proofs, see e.g. [55, 72, 52]. Some basic requirements for such a language are as follows:

**Formality:** This requirement should be taken to mean both:

- that specifications should be formal—as opposed to informal. Although informal specifications can play an invaluable role in program development, they tend to be ambiguous, imprecise, and incomplete. They are also not amenable to automated processing or mathematical manipulation and cannot support the use of formal methods.
- that the specification language should be given a formal definition. Without a formal definition it is difficult to see what significance can be attached to a “proof”

of consistency between a specification and its implementation.

**Intuitiveness and Naturalness:** These characteristics take into account the fact that program specifications and annotations are also meant for human comprehension and appreciation. Because the construction and processing of specifications cannot be fully automated, such requirements remain important.

**Conciseness:** A common complaint about formal specifications is that the specification of a program is longer (and more complicated) than the program itself. A specification language should be geared towards clear and succinct formulation of program properties.

**Abstraction:** Just like for programs, abstraction mechanisms should be applicable to specifications to make them more manageable and allow separation of concerns.

The specification language as currently implemented in the prototype does not lend itself readily to conciseness of expression and contains no abstraction mechanisms apart from simple predicate abbreviations (see section 5.3). This leads to rather unmanageable logical formulas having to be manipulated and used in deriving further program steps. It is difficult to say to what degree these deficiencies could be rectified, especially as it seems to be quite a pervasive problem within the “standard paradigm” of program verification [54]. One may experiment with some additions to the language in an attempt to alleviate the situation, but this may jeopardize the formal basis of the language and bring little relief at that. Another aspect of the specification language that may have a bearing on these shortcomings, should also be considered, i.e. its underlying logic. The following list of desirable properties for logics of specification languages appears in [72]:

**Soundness:** This property is the very least that should be expected of a logical system for conducting proofs about program properties.

**Completeness:** Although desirable, negative results such as Gödel's Incompleteness Theorem, suggest that it may be incompatible with other crucial aspects, such as soundness, in logics of interest.

**Entrenchment:** The involvement of social processes in proofs suggest that a widely known and accepted logic is needed to serve as a standard. It is also desirable to have a large

“library” of reasonable model and proof theoretic results to call upon when necessary. Only first-order logic measures up to this requirement.

**Expressiveness:** This property is needed to characterize a specification in the intended way. First-order logic lacks expressiveness as there are nonstandard models of the first-order Peano axioms. In addition one may also demand that key concepts (e.g. well-foundedness) be expressible in a “natural” manner.

**Freeness:** Quantifiers should not range over error objects, e.g. in arithmetic

$$0 * 5 \text{ div } 0 = 0$$

and

$$0 + 5 \text{ div } 0 = 5 \text{ div } 0$$

should not be logical consequences of the laws

$$(\forall x :: 0 * x = 0)$$

and

$$(\forall x :: 0 + x = x)$$

Only free logics have the property of being free of existential pre-suppositions.

**Constructiveness:** Because programs construct solutions, constructive proofs best capture the nature of programming logic. Only intuitionistic logics (and perhaps their S4 counterparts under the Tarski-McKinsey translation) have this property.

**Deduction Property:** The Deduction Theorem allows the rule of conditional proof to be incorporated in a natural deduction system, making it easy to use. A number of many-valued logics lack this property.

**Hintikka Property:** A reasonable definition of a Hintikka model set (or system) must exist, such that it follows from this definition that any model set is satisfiable. This property is necessary for the development of reasonable Smullyan tree, Beth tableau and Gentzen sequenzen systems. It may also be useful for showing satisfiability or establishing the Craig property (see below). Relevance logics lack this property.

**Prenex Property:** Resolution and various other theorem proving techniques require that a prenex conjunctive normal form theorem must hold. Some modal and tense logics lack this property.

**Craig Property:** The Craig Interpolation Lemma must hold for a logic to have a reasonable model theory. The correctness of Nelson and Oppen's cooperating decision procedures [134] also depends upon this property. Some modal and tense logics lack this property.

Apart from completeness, expressiveness, freeness, and constructiveness, a standard first-order logical system has all the above properties, making it an obvious candidate for the foundation of a specification language. Many variations of standard first-order logic are possible and some variants may be more appropriate as the basis for reasoning about and describing program properties than others. In [33], for example, the authors use a classical many-sorted applied predicate calculus with equality and definitions, in which a constructive subsystem is distinguished.

On the other hand, first-order logic is now considered by many as not being rich enough for the formulation and proof of many interesting properties of programs. A purely first-order framework, for example, is not conducive to the use of induction in proving properties of programs and their data types. This is a serious drawback, as inductive arguments have proven to be a very natural way of expressing such proofs [54]. One reaction to this deficiency has been the use of higher-order logics [61, 4].

If it is important that reasoning about program correctness takes place within a formal system, the lack of expressiveness of the first-order logic underlying our specification language also becomes problematic. As informally shown in section 3.4 and formally in [11], the set of first-order formulae over the state space and operators available in the guarded command language, is not rich enough to express the weakest preconditions of loops. An extension of ordinary first-order logic that allows conjunctions and disjunctions over a countably infinite number of formulae is sufficient for this purpose and is used to this effect by authors such as Back [9, 11] and Hemerik [81]. The logic,  $L_{\omega_1\omega}$ , obtained in this way, is essentially stronger than ordinary first-order logic. One can, for example, give a categorical characterization of the standard model of arithmetic by a single sentence of  $L_{\omega_1\omega}$ . Although  $L_{\omega_1\omega}$  is "close" to ordinary

first-order logic in many respects, it does include inference rules that may require an infinite number of premises to be proved and thus proofs are allowed to be of infinite length. Instead of giving a completely formal proof by showing the sequence of formulae that constitute the proof, one has to resort to induction in such cases to show the existence of a certain proof sequence. Another approach is found in [112], where Manna shows that if one is allowed to use predicate variables, then the weakest preconditions of nondeterministic programs can be expressed in (ordinary) first-order logic. In comparison to Dijkstra's formulation of weakest preconditions, however, the use of predicate variables makes Manna's formulation complicated and difficult to use in reasoning about program properties.

Another problem encountered in expressing properties of guarded command programs in a standard first-order language, is the treatment of undefined terms. As Dijkstra's calculus includes certain mechanisms for dealing with undefinedness, we discuss this topic in more detail.

### Dealing with undefinedness

Partial functions frequently occur in algorithms. The following examples present some sources of such partiality:

- finite machine arithmetic can make  $x + y$  unrepresentable for large  $x$  and  $y$ ;
- the natural number expression  $x - y$  can be undefined;
- $\text{mod}(x, 0)$  and  $\text{div}(x, 0)$ ;
- $ar[x]$  is undefined for array  $ar$ , if  $x$  is outside its domain;
- the use of user-defined functions such as  $\text{fact}(-1)$ , where

$$\begin{array}{l} \text{fact}(x) = \text{ if } x = 0 \rightarrow \text{fact} := 0 \\ \quad \square \ x \neq 0 \rightarrow \text{fact} := x * \text{fact}(x - 1) \\ \text{ fi} \end{array}$$

One of our primary assumptions has been that predicates are total functions on the state space. How should we then interpret  $\text{mod}(x, 0) = \text{mod}(y, 0)$ ?

The problem of what truth value should be given to a sentence involving a term that “has no denotation”, has been studied by many mathematicians, philosophers and computer scientists. Various solutions have been proposed, many of which give rise to non-classical logics (see [15] for references).

In order to give a formal treatment, predicates may be considered as representing either partial functions with values in the set  $Bool = \{T, F\}$  or total functions with values in  $Bool_3 = \{T, F, \perp_{Bool}\}$ . The first option leads to a strict treatment of undefinedness, i.e. the value of a logical formula is undefined whenever at least one of its arguments is undefined. The latter alternative allows a lazy, non-strict treatment of undefinedness, that is more convenient for the purpose of algorithm specification and verification (see [22, page 247–249]). The formalization of three-valued predicates demands the solution of two distinct problems:

1. A calculus of three-valued predicates has to be constructed.
2. A logic suitable for proving facts expressed in terms of three-valued predicates has to be established.

**Three-valued logical calculi** There are various ways of extending the classical two-valued predicate calculus to cater for three logical values. Two well known three-valued calculi proposed in the literature are a calculus described by Kleene [103] and one described by McCarthy [115]. These calculi have the following noteworthy properties [104]:

1. The propositional connectives of both Kleene ( $\neg_K, \vee_K, \wedge_K, \Rightarrow_K$ ) and McCarthy ( $\neg_M, \vee_M, \wedge_M, \Rightarrow_M$ ) extend the classical connectives, i.e. they coincide with them on the values  $T$  and  $F$ . In addition, certain classical relationships are still satisfied, e.g.

$$a \Rightarrow_M b \equiv (\neg_M a) \vee_M b$$

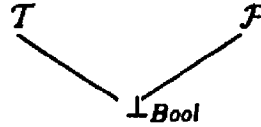
$$a \wedge_K b \equiv \neg_K((\neg_K a) \vee_K (\neg_K b))$$

There are, however, classical laws such as the law of the excluded middle:

$$a \vee (\neg a) \equiv T$$

that do not hold in either of these calculi.

2. Both calculi allow for lazy evaluation of expressions. This means that in evaluating  $a \vee_M b$  (or  $a \vee_K b$ ), if  $a$  evaluates to  $\mathcal{T}$ , then  $b$  need not be evaluated, since the value of the whole expression is  $\mathcal{T}$  regardless of the value of  $b$ .
3. The connectives of both calculi are monotone in the usual cpo over  $Bool_3$ :



Every monotone function in  $Bool_3$  has the property that if one of its arguments is replaced by  $\perp_{Bool}$ , then the value of the function either stays unchanged or becomes  $\perp_{Bool}$ . In addition, the former case is only possible if the function is constant in that argument, so that its evaluation may be lazily omitted. Monotonicity is important, since it is a prerequisite for application of the fixed point theorem as used in section 3.1. In addition, all functions that are monotone in  $Bool_3$  are implementable even when  $\perp_{Bool}$  represents a non-terminating computation, while non-monotone functions are not.

4. McCarthy's connectives are left-strict, i.e. they assume the value  $\perp_{Bool}$  whenever their first (leftmost) argument is  $\perp_{Bool}$ . This property makes McCarthy's connectives implementable on a sequential machine. They are not right-strict and hence  $\vee_M$  and  $\wedge_M$  are not commutative. Kleene's connectives are right-strict, but not left-strict. Thus, Kleene's calculus requires parallelism to implement (see [22] for more details).
5. The quantifiers  $\forall_M, \exists_M$  and  $\forall_K, \exists_K$  extend the classical  $\forall$  and  $\exists$  to the three-valued case and generalize conjunction and disjunction in the calculi of McCarthy and Kleene, respectively. Definitions and a comparative discussion may be found in [22].

The propositional connectives of McCarthy seem more widely acceptable for software specification and verification [22] and have been used in this context in [20, 21]. The connectives  $\vee_M$  and  $\wedge_M$  are found in Euclid, ALGOL-W, C, and Ada (as *and then* and *and or else*). They appear in Dijkstra's weakest precondition calculus as *cand* and *cor*. Kleene's calculus is used for software specification and development in [96, 97].

**Logics incorporating three-valued predicates** One approach is to construct a three-valued logic corresponding to a chosen calculus of three-valued predicates. This approach is explored in [15, 96, 97, 90, 104]. Since some proof rules and techniques from two valued logic are not applicable in such a framework, proving theorems sometimes require more than usual skill and different styles of reasoning [15, 22].

A number of approaches are based on the desire to retain a two-valued logic for reasoning about program assertions. This exploits the existing body of “two-valued intuition” in reasoning and allows mechanical support by a variety of existing systems. In [20, 21, 22] two-valued predicates, called *superpredicates*, are added to a three-valued predicate calculus. These superpredicates are binary relations between predicates and provide a bridge between a calculus of three-valued predicates and a classical two-valued logic. The enriched calculus is shown to be convenient for formulating propositions concerning program correctness and program transformations, with proofs carried out in classical logic. In LCF [59] two levels of truth values are used. Terms, representing values that can be computed, can be undefined, thus allowing three truth values for Boolean expressions, while assertions about terms are two-valued. These two levels are connected through a non-monotone, strong equality predicate. A similar approach is advocated by Tennent in [152]. The PL/CV verification system also uses two-valued logic for reasoning about program correctness. In this framework, two approaches to the problem of undefined terms are proposed; one requiring changes to proof rules of the predicate calculus, while the other does not. A detailed exposition is available in [33].

**Undefinedness in Dijkstra's calculus** Dijkstra's calculus does not include a formal treatment of undefinedness, but only some measures for preventing the evaluation of undefined terms wherever these occur. A brief discussion of the main sources of partiality and how they may be dealt with, from the perspective of the prototype implementation, follows.

**Bounded machines:** One source of partiality in programs is the finite bounds of computing machines. Real machines perform finite arithmetic and can accommodate only a finite state space—leading to undefined values if such bounds are exceeded.

In [38, Chapter 26] Dijkstra points out that having integer variables implies an unbounded state space for guarded command programs. Such programs are implicitly intended for the hypothetical “Unbounded Machine” (UM). Since for all initial states satisfying  $wp(S, T)$ , the number of computation steps as well as the nondeterminism exhibited by a mechanism  $S$  are bounded, a “Sufficiently Large Machine” (SLM) that can simulate the behaviour of the UM on its computation, is generally realizable. Finally Dijkstra suggests that since it is troublesome to determine a priori the size of the SLM suitable for a particular computation, the machine we use should have the additional feature of checking, as computation proceeds, whether it is large enough for that computation. If this is not the case, the machine should explicitly refuse to continue the computation. The SLM with such a feature is called a “Hopefully Sufficiently Large Machine” and the notion of a weakest liberal precondition (see Chapter 2) may be used to describe its boundedness.

In keeping with these views, we do not consider it the duty of a programmer to incorporate implementation dependent parameters into arguments about program correctness. In effect, we are targeting our programs for the UM, thus eliminating machine bounds as a potential source of undefinedness. This is in the spirit of separation of concerns, as advocated by Dijkstra. On the other extreme it is possible to consider the boundedness of real machines very explicitly in the development of correct programs, including machine bounds in program specifications and ensuring that these bounds are nowhere exceeded by suitably extending program assertions. This may add considerable complexity to the already complex process of formal program development. Ensuring the correctness of Pascal programs on bounded machines is explored in some detail in [32], while [158] discusses correctness of Euclid programs in the same context.

**Arrays and other partial functions:** In the prototype implementation arrays as well as the two standard functions `mod` and `div` are the primary sources of undefinedness. Dijkstra shows that in preconditions generated by application of his calculus, undefinedness is avoided by expanding any predicate  $P$  involving a program expression  $e$  to

$$\text{domain}(e) \text{ cand } P.$$

Here, the predicate  $\text{domain}(e)$  characterizes the set of states in which the expression  $e$  can be evaluated without causing failure. Such domain predicates may be mechanically generated (see Appendix C for a definition that may be used for this purpose)<sup>1</sup>. Furthermore a program developer should make use of the operators **cand** and **cor** in the formulation of guards and other Boolean expressions to avoid undefinedness. Ironically, injudicious use of these operators may also lead to undefinedness, e.g. for  $ar[0..n-1]$  the expression

$$i \leq n \text{ cand } ar[i-1] > 0$$

may be undefined. This necessitates the generation of a domain predicate wherever program expressions containing **cand** or **cor** appear in preconditions. For the above expression, the following domain predicate would be generated

$$\begin{aligned} & \text{domain}(i \leq n) \text{ cand } (i \leq n \Rightarrow \text{domain}(ar[i-1] > 0)) \\ & \equiv i \leq n \Rightarrow (ar.\text{lob} \leq i-1 \wedge i-1 \leq ar.\text{hib}) \\ & \equiv i \leq n \Rightarrow 0 \leq i-1 \end{aligned}$$

In formulating program specifications and annotations, bounded quantifiers should also be employed for this purpose [95, page 57–61] e.g. use

$$i < n-1 \text{ cand } a[i] < a[i+1]$$

instead of

$$i < n-1 \wedge a[i] < a[i+1]$$

and

$$(\forall i : i \geq 0 \wedge i \leq n-1 : a[i] > 5)$$

instead of

$$(\forall i :: (i \geq 0 \wedge i \leq n-1) \Rightarrow a[i] > 5)$$

Notice that the equivalence

$$(\forall i : R : P) \equiv (\forall i :: R \Rightarrow P)$$

from [14, page 239] does not apply if  $P$  may become undefined.

<sup>1</sup>To avoid additional complexity in preconditions the prototype implementation does not include any treatment of undefinedness—domain predicates are not automatically generated.

A disadvantage of not having a formal treatment of undefinedness is that the responsibility for avoiding undefined terms (and predicates) ultimately rests with the programmer. In providing mechanical support for reasoning about program assertions, a formal proof theory has to be established. In the framework of Dijkstra's calculus, a two-tiered approach such as followed in LCF seems a suitable candidate as it obviates the need for acquiring reasoning prowess in an exotic logic. However, further investigation is necessary.

**Note:** Two other potential sources of partiality that we do not have to contend with in the prototype are user-defined functions and subrange types. It should be possible to accommodate these in a similar fashion to the above.

### **Concluding remarks on the specification language**

In conclusion, it is obvious that the specification language for guarded command programs, as implemented in the prototype (see Appendix B for a formal description), is far from ideal. Apart from certain cosmetic changes, it is also not clear how the situation may be improved satisfactorily. Most of the shortcomings pointed out, seem to be directly attributable to its underlying standard first-order logical formalism. If this logic is to be abandoned in favour of a richer and more expressive system, which one do we choose? A higher order logic would allow inductive arguments and more concise formulations of certain properties, the logic  $L_{\omega_1\omega}$  would give expressiveness, while a logic allowing reasoning about partial functions would remove problems in dealing with undefined terms. Each of these alternatives also has complications of its own and we may well find that there is more to lose than to gain in changing to a more exotic logic. A deeper investigation than that conducted here will be necessary to fully explore all the options. If we consider (as we do, in Chapters 5 and 7) extending the guarded command language to allow for various mechanisms of procedural and data abstraction, it becomes more important that abstraction mechanisms also be applicable to specifications. It may then be worthwhile to consider the use of an established specification language, such as Z [150] or Larch [76, 75], with guarded command programs.

### **4.3.3 Perspectives on Program Development Support**

When verification conditions are generated in batch mode as done by the prototype implementation, a programmer does not receive any help in constructing a correct program, but only in showing that it is correct after its construction. Some macros were implemented in a programmable text editor to support application of Gries strategies for developing loops and alternative commands, but because no semantic checks or theorem proving is performed, these provide very limited help. Even for small programs, verification conditions generated under these circumstances may be extremely complex and detailed (see Appendix E for some examples), as they represent all the correctness arguments that should have been considered during the development.

If incremental program development support can be provided, smaller proof obligations may be generated and discharged on a continual basis, allowing a factorization of the proof burden. Additionally, the programmer is in a better situation to apply domain-specific knowledge at an early stage to simplify propositions to be proven and to find mistakes. The philosophy of “verifying the correctness of a constructed program”, is exchanged for “maintaining the correctness of a program under construction”. In what follows, some perspectives on providing interactive mechanical program development support are presented.

#### **Initial requirements**

As a starting point we refer back to section 4.1 to obtain the following list of basic functions to be supported by a mechanical program construction assistant:

1. interactive precondition calculations, with a recalculation facility to allow incremental changes to programs and annotations;
2. automated simplifications on preconditions and verification conditions as they are generated;
3. interactive generation (and proving) of verification conditions;

4. facilities for displaying and editing programs (possibly annotated with generated preconditions) and assertions;
5. the capability of interleaving the above operations as necessary during program construction.

Verification conditions may be recorded for proof at a later stage or, preferably, an interactive theorem prover may be provided as part of the system (see section 4.3.4 for suggestions on proof support). Because proofs of program properties play an important part in program development strategies, a formal reasoning component is necessary if support for application of such strategies is envisioned. A program development system may also offer automated application of techniques for discovering invariants and bound functions, e.g. manipulations for weakening a predicate and for combining predicates may be provided.

### **Development strategies**

Apart from variations in the sophistication and extent of basic features which a program development system may offer, some variation in the amount of guidance that a programmer will receive, is also possible.

One approach is to provide the features listed above and allow a user to apply them at will during program construction. This approach is exemplified in a project undertaken by Odyssey Research Associates [143, 70] to assess the feasibility of constructing formally verified Ada programs:

Their prototype implementation centers around a syntax-directed (structure) editor, Penelope, that can be used to create and edit abstract syntax trees that represent parts of an annotated Ada program. Penelope computes (approximated) weakest preconditions as attributes of the nodes of a syntax tree, incrementally recomputing and propagating changed conditions through the tree as a program is developed. A simplifier, based on the Nelson-Oppen algorithm for cooperating decision procedures [134], as well as facilities for using lemmas and axioms of theories as rewrite rules are available in Penelope. Verification conditions are generated incrementally, with automatic updating as a program or its annotations change. To

support proofs of verification conditions, a sub-editor is available, enabling users to construct proofs using a sequent calculus. Penelope's theorem proving facilities are not extensive, but are useful in that proofs of verification conditions that have changed are "replayed", with any parts that are no longer valid, indicated to the user. The Penelope editor was produced from a suitable attribute grammar using the Cornell Synthesizer Generator [144, 145]. The predicate transformers implemented in the editor were derived from a denotational semantics of Ada.

Syntax directed editing [110] is a well-known concept in integrated programming environments that exploit program structure in supporting interactive program construction. If such an editor can be generated from a formal description, such as an attribute grammar in the case of Penelope, it provides an excellent vehicle for experimenting with extensions to the syntax and semantics of the programming language.

A higher level of support is possible by integrating the basic functions given above into a number of program development strategies, such as those proposed by Gries [66] and Dromey [43]. Such a system guides a user through the process of program construction by proposing subgoals for the development, according to some built-in strategy, while prompting the user for decisions and information that fall outside its jurisdiction. An implementation of Gries's program development strategies within the framework of the Karlsruhe Interactive Verifier (KIV) [80], is described in [80]:

This implementation involved a formalization of Gries's development method using a sequent calculus for a dynamic logic DL, which is the underlying logic of KIV. The implemented program development strategies construct and manipulate proof trees, representing program developments that are provably consistent with a specification. The goal of program development is expressed as a logical formula, which is placed in the root of such a proof tree. Subprograms and guards still to be developed, are represented by metavariables in tree nodes. The programming strategies guide the user through an expansion of the proof tree in a top-down fashion, during which metavariables are systematically instantiated. Program development is completed when the root contains no more metavariables and the leaves contain only predicate logic formulas. A user may cancel undesirable development steps by invoking a "backtracking" function, which removes all developments up to the last recursive invocation

of the main development strategy.

One advantage of implementing Dijkstra's calculus in a sophisticated theorem prover such as KIV [80] or HOL [60] (see [61, 4] for descriptions of such projects) is the formal assurance of soundness that may be obtained. The programming language semantics is formalized in the logic supported by the prover. Then soundness preserving methods may be employed to formally derive the strategies for generation of verification conditions and for program development from the semantics. This guarantees that results obtained from application of such strategies are logical consequences of the underlying program semantics. This kind of assurance is, for instance, not possible for the traditional verification system consisting of a verification condition generator and a separate theorem proving subsystem or even for systems such as [143, 70]. Another advantage is that it is simple to extend the guarded command language under consideration to include new constructs such as procedures. Systems such as KIV and HOL also present excellent opportunities for experimenting with different methods of simplification and proof of verification conditions (see [4]).

### **Final considerations**

There is no evidence to suggest that a programmer using Dijkstra's calculus will apply only certain development strategies (such as those of [66]). If anything, the informal nature of paper and pencil developments seem to indicate otherwise. While novices may benefit most from step-by-step guidance, it is more important to build a mechanical support system for interactive programming in such a way that it allows a great deal of flexibility for the user:

- A programmer should be allowed to apply any development technique that produces a program that is consistent with the specification.
- It is very important that a programmer should be able to revise or undo previous development steps at any stage as more knowledge is gained. Any development or proof steps influenced by such a change should be indicated.
- A programmer should have the freedom of determining the order in which subgoals are tackled. This allows development to proceed bottom-up, top-down or indeed in any other sequence.

- A programmer should be allowed to select the degree of annotation displayed with program code during development.

Thus one should aim for a system that is able to:

1. allow a programmer to store, retrieve, view, and edit completed and partially completed developments of guarded command programs;
2. calculate preconditions of programs at request to aid in development steps or on its own initiative as part of proof obligations;
3. make known any proof obligations arising from a development step as soon as they become applicable;
4. assist a programmer in discharging proof obligations at request.

It would be useful if such a system could provide a way of experimenting with automation of development strategies and other heuristics such as those for developing loop invariants. One suggestion is to allow a user to capture frequently used strategies and heuristics as “programs” in a metalanguage which may then be “executed” by the system. This principle is applied to theorem proving in systems such as LCF, KIV, and HOL, where the system may be extended with useful patterns of inferencing discovered by users by “coding” these in the form of *tactics* and *tacticals* [118]. Other approaches to automation of development strategies such as found in expert-system based or transformation-based systems, e.g. [36, 50, 91, 139, 148, 151] should also be investigated.

Another factor that is of vital importance in supporting interactive application of Dijkstra's calculus is the control of complexity. On the one hand this means that every effort should be made to simplify the logical formulae displayed and manipulated during program development and on the other, that during the development of programs, a programmer should be able to “hide” details that are considered unnecessary at that stage. Informal program annotations are also useful in this regard and their use should be allowed.

The successes of projects such as [143, 70], [4] and [80] show that there are merits to using either a structure editor generator or a sophisticated theorem proving environment for implementing

experimental support systems for programming in Dijkstra's calculus. Both alternatives offer facilities for experimenting with various aspects of the calculus as well as the type of support provided to programmers. A thorough comparative study will be necessary to recommend one of these alternatives over the other. Meaningful interactive support, including theorem proving facilities, will only be possible on hardware such as powerful workstations.

### **4.3.4 Perspectives on Proof Support**

#### **General**

Though the prototype implementation offers no proof support mechanisms, it is clear that such a component would be crucial even if elementary mechanical program development support is to be provided. The topic is therefore pursued briefly here.

The logical formulae generated as proof obligations through the application of Dijkstra's calculus, are very similar to those resulting from standard program verification methods for sequential programs, such as Hoare logic or the inductive assertion method. (Compare for example the verification conditions generated for "Rubin's problem" by the prototype (see Appendix E), to the verification condition examples in a paper such as [92].) A mechanical theorem prover or proof assistant suitable for the latter, should thus be equally suitable for supporting the use of Dijkstra's calculus.

As mentioned in section 4.1 much research has been done in the area of mechanical support for formal reasoning. A good survey of computer support for the type of formal reasoning resulting from software engineering applications may be found in [107]. Although some program verification environments with integrated reasoning facilities, such as Gypsy and the Stanford Pascal Verifier, exist, dedicated formal reasoning systems generally offer better features. The latter may be roughly divided into two classes:

1. highly automated systems, such as resolution provers and the Boyer-Moore prover [26] (see [109] for a survey of such systems)
2. interactive proof editors and proof checkers such as LCF and HOL.

Highly automated theorem provers typically conduct systematic searches for proofs. Because of the size of the search space, complex heuristics and/or user advice are often used to constrain and direct search efforts. They are limited by undecidability results for most theories of interest to software engineering applications and by requiring a user to have an intimate knowledge of the system's complicated search routines and heuristics in order to guide it down the right paths in case of failure.

Most interactive formal reasoning assistants concentrate on a goal-directed approach to proof construction. The user extracts subgoals for proof from a goal, and new subgoals from these in turn, until the prover is able to invoke its routines for automated proof to deduce that the subgoals are true. Some features that have been identified as useful and desirable for proof assistants for software engineering applications are (also see [107] and [98, 99]):

1. Although completely automated theorem proving is not practical in this area, a powerful automatic proving component may be used to good effect as part of interactive theorem proving. Such a component serves to eliminate much of the tedium of obvious reasoning steps. Good examples of techniques that have been used with success in automatic provers are implementations of (combinations of) decision procedures for decidable theories, term rewriting and congruence closure algorithms for equational reasoning and different variations of resolution.
2. Another way of combating the tedium of proof construction is to derive new rules of inference and to allow frequently used patterns of inference to be "coded into" a theorem proving system, thus building up a repertoire of proof strategies suitable to different types of problems. Such patterns of reasoning are called *tactics* (and *tacticals*) [118]. The LCF, KIV, and HOL systems, among others, support tactical reasoning. Imaginative matching algorithms must be used to help a user in locating lemmas, inference rules, and tactics that may be applied in specific situations.
3. To allow for flexibility during proof construction, the user should have access to different proof styles: natural deduction, backwards as well as forwards reasoning, constructing a proof by sketching a proof outline and refining it to an appropriate level of detail. Wherever possible, the choice of which subgoal to prove next should rest with the user.

Backtracking to various stages of proof construction should be possible.

4. Instead of supporting only “formal” proofs, for which the expected gains may not justify the effort, “rigorous” proofs may be allowed. One step towards this is to permit a user to assert the truth of certain conjectures without formally justifying them. The system should keep track of all such lemmas on which a proof depends and prevent circularities.
5. All possible measures should be taken to ensure soundness of reasoning tools. In particular the soundness of derived inference rules and tactics must be formally ensured. If a user is allowed to add axioms to the system, the best assurance that can be given is soundness relative to a given logical calculus and a set of user-supplied axioms.
6. To manage the explosion of detail that occurs in proofs of reasonable size, a definition mechanism for the folding and unfolding of abbreviations is useful. A specific discipline for hiding irrelevant details of proofs should be applied during their construction.
7. To accommodate proofs about the data types used in programs, it is important that a prover be able to handle theories of data types containing axioms and some logical consequences thereof that are useful in constructing proofs. The prover should provide built-in theories for common data types such as integers, Booleans and characters. To handle compound data types such as arrays, parameterization of theories may be employed. Facilities for constructing new theories should be provided to handle proof obligations arising from treatments of programs containing definitions of abstract data types (see section 5.2). Reuse of theories may be encouraged by supporting various ways of constructing new theories from existing ones. It is important that the information in theories is used effectively, thus it is conceivable that information such as tactics, decision procedures, simplification methods, normal forms, etc. could form part of theories. This aspect is not yet considered as state of the art in theorem proving systems [107]. Various kinds of induction are among the proof techniques that have proven to be useful for propositions involving popular data types and this should be supported by a prover.
8. Because proofs may take long to complete, it is desirable to be able to store and retrieve partially completed proofs.
9. The structure of a proof is frequently expressed as a tree, called a *proof tree* or, more

generally, as an acyclic graph (see [99]). The user interface of a theorem proving system should allow a user to navigate, extend and edit such structures in a natural way. Structure editor-like interfaces have much to offer in this regard.

Verification conditions are typically long and complex logical formulae (see Appendix E for examples). They are usually implications of the form:

$$A_1 \wedge A_2 \wedge \dots \wedge A_n \Rightarrow C_1 \wedge C_2 \wedge \dots \wedge C_m$$

and in many cases the consequent is a minor modification of the antecedent. In [4, Chapter 8] a collection of tactics is given which help to automate some of the proof effort required to prove such verification conditions in the HOL prover.

### **Simplification of logical formulae**

For interactive program development support, the single most important facility that should be provided is the simplification of preconditions and other logical formulae generated during program construction.

There are a number of (sometimes conflicting) reasons for simplifying verification conditions:

- to make them easier to read and understand;
- as a means of proving (or disproving) them;
- to reduce them to a form that allows effective use of other proof procedures.

We also find that many of the preconditions that would be generated during program development are so long and complicated that, unless significant simplification can be performed on them, a programmer may overlook vital information to drive further program developments. This fact is borne out by implementations such as [143].

This type of simplification is very difficult. The main problem is that there are so many ways in which to simplify any given logical formula—no general canonical form exists. When a verification condition is in a form that allows efficient processing by automated proving

facilities, it may indeed be particularly obscure to a human reader and vice versa. Here are some of the simplifications that may be performed:

A minority of the candidates for simplification are trivial to recognize and simplify, e.g. propositional terms such as  $P \wedge T$ ,  $P \wedge P$ ,  $P \vee P$  and  $P \vee \mathcal{F}$  may be rewritten to  $P$ , while others such as  $P \vee T$  and  $P \wedge \mathcal{F}$  may be reduced to  $T$  and  $\mathcal{F}$ , respectively. In addition arithmetic terms may be simplified by performing arithmetic on integer literals.

Many obvious simplification steps may be performed automatically by implementing decision procedures or term rewriting algorithms for decidable theories. Such theories include propositional logic, the theory of equality with uninterpreted function symbols, Pressburger arithmetic, the theory of real numbers under  $+$  and  $*$ , the theory of arrays under **store** and **select** and the theory of lists under **car**, **cdr**, **cons** and **atom**. (These theories are all quantifier-free.) There are also simplification techniques for arithmetic terms that deduce integer inequalities by keeping track of the bounds imposed on integer variables in logical formulae, in terms of other variables and constants.

All the theories involved, provide equivalences that may be used for “rewriting” logical formulae e.g.

$$[A \Rightarrow (B \Rightarrow (C \Rightarrow D))] \equiv (A \wedge B) \Rightarrow (C \Rightarrow D) \equiv (A \wedge B \wedge C) \Rightarrow D]$$

and

$$[x < y \vee x = y \equiv x \leq y]$$

However, for most of these one cannot say offhand which form is to be preferred. The context in which a formula appears may have to be used as a guide in such situations. Another good example is to be found in the manipulation of the range and term expressions of quantified formulae, that may be combined or split in various ways (see [14] or [40] for a list of equivalences). e.g.

$$\begin{aligned} &[(\forall i : 0 < i \wedge i \leq k + n : a[i] > 0) \equiv \\ &(\forall i : 0 < i \wedge i \leq k : a[i] > 0) \wedge (\forall i : k < i \wedge i \leq k + n : a[i] > 0)] \end{aligned}$$

The best way to control mechanical simplifications of this kind, is to allow as much user guidance as possible and attempt to build up a set of usable heuristics through experience.

Much experimentation may be necessary to find a good simplification method for the formulae manipulated during use of Dijkstra's calculus. The safest route to follow, would be to start with an automatic simplifier that only simplifies terms belonging to theories or subtheories with well understood normal forms. It should rather do too little than too much. More simplification could be done under direct control of the user. Provided efficient and effective ways can be found to incorporate additional simplifications into the automated simplifier, it may be expanded as more knowledge is gained.

## 4.4 Conclusion

Formalization of Dijkstra's calculus is necessary to allow meaningful mechanical support for its use.

On the one hand, loss of the ability to mix formal and informal arguments and notation limits the applicability of the calculus. To combat this effect, special care should be applied in providing a powerful and flexible type system for guarded command programs as well as an expressive and powerful specification language. At the same time, it is important not to disregard the effects of choices made in these matters on the construction of proofs.

Another negative effect of complete formality is additional complexity in applying Dijkstra's calculus, e.g. preconditions and verification conditions generated during mechanical application of the calculus are often prohibitively complicated, while the effort of constructing formal proofs of obvious results may not justify the effort expended. (This is aptly illustrated by the verification conditions generated from the "Rubin's problem" example of section 3.5 (see Appendix E)). This is a decidedly unattractive feature for potential users and puts the usefulness of mechanical support in question. Such complexity does not appear in textbooks such as [38, 66, 14, 40, 43] because a large degree of informality is allowed. It is possible to allow some informality in mechanically supported program development, e.g. by supporting "rigorous" instead of "formal" proofs of program properties, allowing informal as well as formal program annotations and not unnecessarily restricting the development strategies allowed.

Another factor that limits complexity in examples done by hand is the conscious use of

abstraction. Providing formal mechanisms of abstraction in program development will provide a way of factoring program development to control complexity during mechanical program construction. Abstraction mechanisms may also provide a basis for the construction and recording of a meaningful and informative development "history" of a program such as results from paper and pencil developments. These considerations prompted further investigation. In Part 2 of this thesis a survey of possible abstraction mechanisms for Dijkstra's calculus is presented.

## **Part II**

# **Providing for the Formal Use of Abstraction**

## Chapter 5

# Abstraction Mechanisms for Dijkstra's Calculus

In this chapter we investigate facilities for three kinds of abstraction in Dijkstra's weakest precondition calculus: procedural, data, and specification abstraction.

Procedural and data abstraction are related concepts. A procedural abstraction represents a mapping from a set of input values to a set of output values (usually implemented as a procedure or subroutine). Thus we are allowed to ignore the details of the mapping and consider only its effect. The domain and range of a procedural abstraction consist of data abstractions. A data abstraction provides a set of data values and a set of operations to manipulate these values. The behaviour of a procedural abstraction is defined solely in terms of the abstract data objects thus created, ignoring details of how such objects are concretely represented for implementation.

To assist in the development and management of complex specifications, we may similarly introduce abstractions that reduce the amount of detail under consideration at any one time.

## 5.1 Procedural Abstraction

Program development is frequently characterized by the identification and subsequent individual development of coherent subtasks within the main development process. This is the well known process of procedural abstraction at work. Alternatively, if we view the construction of a program according to a specification as a constructive proof, procedural abstraction embodies the formulation and proof of lemmas. Eventually all components are integrated, typically by placing parameterized procedure calls at the points of abstraction. Procedural abstraction plays an integral role in stepwise program development and formalizing a mechanism for its application in the weakest precondition calculus, deserves attention. Axiomatic treatments of the semantics of procedure constructs are complicated by the influence of various traditions as well as the fact that so many issues are involved: local variables, abstraction, parameterization, procedures, recursion.

Recent treatments such as [81, 120] attempt to simplify matters by considering some of these concerns separately.

Staying within the semantic framework of Chapter 2, we follow the same progression as in [81]. The local block construct can be seen as a first step in the investigation of the semantics of a procedure construct, in that it may be used to consider the introduction of local nomenclature in isolation from other aspects. We also consider the effects of parameterization before combining these features into a general procedure construct that makes provision for procedural abstraction.

### 5.1.1 Local Blocks Revisited

In Chapter 3 the following two alternatives for the semantics of local blocks were considered:

$$[wp(|[\text{var } x : T \mid S]|, R) = wp(S_y^x, R)] \quad (17)$$

where  $y$  is a fresh variable and  $S$  initializes  $x$ ; or

$$[wp(|[\text{var } x : T \mid S]|, R) = (wp(S, R_y^x))_x^y] \quad (18)$$

where  $y$  does not occur free in  $R$  or  $S$ , and  $x$  is initialized by  $S$

The first alternative requires renaming of local variables in program text, while the second uses temporary renaming of global identifiers in certain program assertions. In [6] it is pointed out that when we consider the introduction of procedures, a more subtle difference between these two semantics also appears.

Consider a parameterless procedure  $P$  with body:

$$x := y$$

It is our desire to interpret all procedure calls according to the ALGOL-60 copy rule, which states that a program that calls a procedure is equivalent to one in which the procedure name is replaced by the text of the procedure body, with suitable renaming of local variables to avoid name clashes. Using this rule and (17), we find:

$$\begin{aligned} & wp((y := 0; |[var\ y : integer \mid y := 5; P]|), x = 0) \\ & \equiv wp((y := 0, wp([var\ y : integer \mid y := 5; P]|), x = 0)) \\ & \equiv wp(y := 0, wp((z := 5; P), x = 0)) \\ & \equiv wp(y := 0, wp((z := 5; x := y), x = 0)) \\ & \equiv \mathcal{T} \end{aligned}$$

while (18) gives:

$$\begin{aligned} & wp((y := 0; |[var\ y : integer \mid y := 5; P]|), x = 0) \\ & \equiv wp((y := 0, wp((y := 5; P), x = 0)) \\ & \equiv wp((y := 0, wp((y := 5; x := y), x = 0)) \\ & \equiv \mathcal{F} \end{aligned}$$

Thus (17) leads to a situation where each procedure call is evaluated in the environment in which the procedure has been declared, i.e. *static scope*, while (18) reflects the case where procedure calls are evaluated in the environment of the call, or *dynamic scope*. Our choice of (17) for the semantics of local blocks is in accordance with the fact that all ALGOL-like languages use a static scope assumption.

### 5.1.2 Parameterized Statements

To study parameterization, we introduce parameterized statements resembling Hemerik's "abstractions" and "applications", [81, page 121].

**Syntax:**

$$[[\text{value } x : T_1; \text{ value result } y : T_2; \text{ result } z : T_3 \mid S]](e, v, w)$$

**Comments:**

- $x, y$ , and  $z$  are the *formal parameters* and their types are given by  $T_1, T_2$ , and  $T_3$ , respectively;
- the attributes **value**, **value result**, and **result** determine the details of parameter treatment (see [66] for details);
- there may be any number of **value**, **value result**, and **result** formal parameters;
- $S$  is a guarded command program construct;
- $e, v$ , and  $w$  are the *actual parameters* with  $e$  an expression matching the type of  $x$ , and  $v$  and  $w$  variables matching the types of  $y$  and  $z$ , respectively.

**Restrictions:**

1.  $S$  has no access to global variables;
2. **actual value result** and **result** parameters must be mutually different;
3.  $wp(S, R)$  must be independent of  $z$  for any condition  $R$ . This means that  $S$  initializes  $z$ , formally:  
for any condition  $R$ ,  $[(\forall v :: wp(S, R)_v^z \equiv wp(S, R))]$ ;
4.  $S$  must be *transparent* to  $x$  [114, 18]. Transparency of  $S$  to  $x$  means that  $S$  contains no assignment to  $x$  and no parameterized statement with  $x$  as **actual value result** or **result** parameter, formally:  
for any constant  $k$  of suitable type,  $[x = k \Rightarrow wlp(S, x = k)]$ .

**Semantics:** We consider the statement

$$[[\text{value } x : T_1; \text{ value result } y : T_2; \text{ result } z : T_3 \mid S]](e, v, w)$$

to be equivalent to the local block:

$$[[\text{const } x = e \text{ var } y : T_2; z : T_3 \mid y := v; S; v, w := y, z]]$$

Under the assumption that the formal parameters are not redeclarations of global identifiers, we obtain the following weakest precondition semantics:

$$\begin{aligned} & [wp([[\text{value } x : T_1; \text{ value result } y : T_2; \text{ result } z : T_3 \mid S]](e, v, w), R) \\ & \equiv (wp(S, R_{y,z}^{v,w}))_{e,v}^{x,y} \end{aligned} \quad (19)$$

### Discussion of assumptions and restrictions

**No global identifiers in  $S$ :** In [114, page 308–309] Martin shows how this restriction may be lifted. Essentially global variables are treated as additional **value** and **value result** parameters depending on how they are used in  $S$ . Martin points out that structured global variables present some difficulty in deciding whether the whole structured variable or only some component should be declared as the global entity. A treatment of globals restricted to simple identifiers may be found in [67, page 568].

**Mutually different actual result and value result parameters:** This restriction addresses the issue of aliasing. Considering (19) we find that if the actual **value result** and **result** parameters are aliased, the innermost substitution has to be understood in terms of the definition of multiple substitutions in section 4.2.4. Thus, a left-to-right order is imposed on multiple substitutions involving the same variable. This approach is used in [67, 66] and (in essence) [28]. Other approaches to handling aliasing frequently involve distinguishing between variables (locations) and their values (e.g. in [24]), or in the case of [5] between program identifiers, locations, and values; allowing aliased identifiers to be recognized as such by their being mapped to the same location identifier.

Most reference works, however, forbid aliasing between **value result** and **result** parameters. As pointed out in [120], aliasing may cause parameterization to lose the important property of monotonicity, i.e. in general:

$$(\forall Q :: wp(S_1, Q) \Rightarrow wp(S_2, Q)) \not\Rightarrow (\forall R :: wp(S_1^{\text{par}}, R) \Rightarrow wp(S_2^{\text{par}}, R))$$

where  $S_1^{par}$  and  $S_2^{par}$  denote program statements  $S_1$  and  $S_2$ , respectively, parameterized in the same way.

Even equality may be lost:

$$\begin{aligned} & wp([result\ x, y : integer \mid x := 0; y := 1])(a, a), a = 0) \\ & \neq wp([result\ y, x : integer \mid x := 0; y := 1])(a, a), a = 0) \end{aligned}$$

Thus, aliasing inhibits the use of statement parameterization for stepwise refinement (see Chapter 7).

If it is forbidden, the absence of aliasing among the appropriate actual parameters must be formally established. A simple solution is to have a verification condition generator examine actual parameters for aliasing as far as possible. Where freedom from aliasing cannot be determined, an appropriate disjointness condition must be generated and conjoined to the weakest precondition of the parameterized statement. A more elaborate treatment is found in [147]. Here program specifications are expressed in an augmented specification logic wherein program identifiers are distinguished from their meanings, allowing “non-interference” conditions to be formulated explicitly as part of a specification.

**$S$  must initialize the result parameters:** Requiring that  $wp(S, R)$  be independent of  $z$ , is the same as the restriction imposed on local blocks in Chapter 4 to ensure that all local variables are properly initialized. For practical reasons, this condition should be reformulated so that it is syntactically enforceable during parsing. A full discussion of the requirements for performing such a check can be found in section 4.2.3.

**No redeclaration of global identifiers:** The restriction that prohibits formal parameters from being redeclarations of global identifiers, can be lifted if a suitable renaming is applied to all formal parameters, i.e.

$$\begin{aligned} & [wp([value\ x : T_1; value\ result\ y : T_2; result\ z : T_3 \mid S])(e, v, w), R) \\ & \equiv (wp(S_{p,q,r}^{x,y,z}, R_{q,r}^{v,w}))_{e,v}^{p,q} \end{aligned}$$

where  $p, q$ , and  $r$  are fresh variables and  $S$  initializes  $z$ .

**Transparency of  $S$  to the formal value parameters:** This requirement is a formal enforcement of the concept of a value parameter. Although it is not strictly necessary

to enforce transparency on parameterized statements, it allows a simpler treatment of procedural abstraction below. Transparency is enforceable by a simple syntactic check on  $S$ : no assignments to the formal value parameters and no parameterized statements using the formal value parameters as actual value result or result parameters are allowed.

**Reference parameters:** Reference parameters allow implementations that are more efficient in terms of time and space, but cause additional complications. The reference works [67] and [66] include weakest precondition semantics for this parameter passing mechanism.

### 5.1.3 A Procedure Construct

So far we have considered a local block construct that makes provision for the introduction of local nomenclature and a mechanism for statement parameterization that adds flexibility to program text through the use of parameters. To make provision for procedural abstraction, procedure declarations and calls are introduced.

#### Procedure declaration

The following syntax for procedure declarations bears a strong resemblance to the parameterized statements above and is also similar to that of [66]:

$$\text{proc } p(\text{value } x : T1; \text{ value result } y : T2; \text{ result } z : T3) : \llbracket \{P\}B\{Q\} \rrbracket$$

Comments:

- $x, y, z$  are the *formal parameters* of procedure  $p$ —all the above comments about the formal parameters of parameterized statements apply also to the formal parameters of procedures;
- $B$  is a guarded command program construct—called the *body of the procedure*

- $P$  and  $Q$  are assertions embodying the specification of the procedure  $p$ , with  $P$  a function of  $x$  and  $y$  only, and  $Q$  a function of  $x, y$ , and  $z$  only;
- restrictions and assumptions concerning the formal parameters and the body of the procedure declaration are as discussed above for parameterized statements;
- each procedure declaration leads to the generation of a verification condition stating that  $B$  is consistent with its specification, i.e. :

$$\lceil P \Rightarrow wp(B, Q) \rceil,$$

which is assumed to have been proven when writing procedure calls based on this declaration.

### Procedure call

**Syntax:**  $p(e, v, w)$

**Comments:**

- This call assumes a declaration for procedure  $p$  as above;
- $e, v$ , and  $w$  are the *actual parameters* with  $e$  an expression matching the type of the formal value parameter  $x$ , and the variables  $v$  and  $w$  matching the types of the formal value result parameter  $y$  and the formal result parameter  $z$ , respectively;
- assumptions and restrictions for actual parameters are as discussed above for parameterized statements;
- recursive procedure calls are excluded here (see page 121).

**Semantics:** As for parameterized statements, we consider the procedure call

$$p(e, v, w)$$

to be equivalent to the local block:

$$\llbracket \text{const } x = e \text{ var } y : T_2; z : T_3 \mid y := v; B; v, w := y, z \rrbracket$$

Assuming that the formal parameters are not redeclarations of global identifiers, the weakest precondition semantics is as follows:

$$\lceil wp(p(e, v, w), R) \equiv (wp(B, R_{y,z}^{v,w}))_{e,v}^{x,y} \rceil \quad (20)$$

### Proof rules

To compute the weakest precondition of a procedure call, detailed knowledge of the procedure body is required. Using a procedure call as an abstraction mechanism, one desires to concentrate on what the procedure does, i.e. its specification, as opposed to a detailed description of how it accomplishes this task. Thus, from the specification  $[P, Q]$  of a procedure, i.e. predicates  $P$  and  $Q$  such that

$$[P \Rightarrow wp(B, Q)],$$

we want a rule enabling us to construct a predicate  $U$  such that:

$$[U \Rightarrow wp(p(e, v, w), R)]$$

for a given predicate  $R$ .

Several proof rules for the correctness of procedure calls have been published, including the traditional treatments [85, 92, 45, 47, 67] and [66, 114, 17] which apply specifically to guarded command programs under weakest precondition semantics. In [17], Bijlsma et al. prove the following formula

$$[wp(B, V)_{e,v}^{x,y} \wedge A_e^x \Rightarrow wp(p(e, v, w), R)] \quad (21)$$

where  $V$  and  $A$  are any two predicates such that  $A$  is independent of  $y$  and  $z$ , and

$$[V \wedge A \Rightarrow R_{y,z}^{v,w}]. \quad (22)$$

Formula (21) can be interpreted as a “skeleton” proof rule. From it, the three most prominent procedure call proof rules for weakest precondition semantics may be obtained by suitable choices for the predicates  $V$  and  $A$ .

- Choosing  $Q$  for  $V$  and  $(\forall y, z :: V \Rightarrow R_{y,z}^{v,w})$  for  $A$ , we obtain a rule due to Gries, [66, page 153] (here formulated in terms of a logical constant  $m$  used in the procedure specification,  $[P, Q]$ )

$$[(\exists m :: P_{e,v}^{x,y} \wedge (\forall y, z :: Q_e^x \Rightarrow R_{y,z}^{v,w})) \Rightarrow wp(p(e, v, w), R)]$$

- Choosing  $Q$  for  $V$  and letting  $A$  be determined by (22), delivers Martin's rule, [114] (shown here with a logical constant  $m$ )

$$[(\exists m :: P_{e,v}^{x,y} \wedge A_e^x \Rightarrow wp(p(e, v, w), R))]$$

- choosing  $(\forall m :: P_v^y \Rightarrow Q)$  for  $V$  and  $(\forall y, z :: V \Rightarrow R_{y,x}^{v,w})$  for  $A$ , gives the rule formulated by Bijlsma et al. [17] (where  $m$  is a logical constant used in the procedure specification)

$$[((\exists m :: P_{e,v}^{x,y}) \wedge (\forall y, z :: (\forall m :: P_{e,v}^{x,y} \Rightarrow Q_e^x) \Rightarrow R_{y,x}^{v,w})) \Rightarrow wp(p(e, v, w), R)]$$

In [18], Bijlsma et al. prove that the proof rules of Martin and Gries are equivalent, in the sense that the predicate ( $U$ ) in Gries's rule is the weakest predicate obtainable from Martin's rule. Martin's rule does not require  $A$  to be the weakest solution of (22), arguing that a stronger predicate may be easier to establish and sufficient for a particular situation. In [17] it is shown that the predicate  $U$  delivered by Gries's rule, is at least as strong as that delivered by the rule of Bijlsma et al. These two rules coincide when there is at most one value of  $m$  satisfying  $P_{e,v}^{x,y}$ , and when  $m$  does not actually occur in  $P$  or  $Q$ .

In [67, 66] Gries proposes another proof rule that may be easier to apply by hand as it eliminates the complicated conjunct  $(\forall y, z :: Q_e^x \Rightarrow R_{y,x}^{v,w})$ . This rule is only applicable under certain circumstances and requires that the postcondition of a procedure call be of the form  $Q_{v,w}^{y,x} \wedge I$  where the free variables of predicate  $I$  are disjoint from  $v$  and  $w$ .

From the point of view of mechanical generation of verification conditions, a proof rule should be simple to apply. This excludes candidates such as the above where the postcondition is not a simple identifier, or as in Martin's rule, where the predicate  $A$  is not mechanically constructed. Mechanical application of a procedure call proof rule also makes it important that the predicate  $U$  it delivers is as weak as possible. In this respect the rule of Bijlsma et al. is the most attractive one, since it satisfies the following "sharpness" theorem, [17]

**Theorem 5.1.3.1 (Sharpness Theorem)** *Let  $U$  be the predicate*

$$(\exists m :: P_{e,v}^{x,y}) \wedge (\forall y, z :: (\forall m :: P_{e,v}^{x,y} \Rightarrow Q_e^x) \Rightarrow R_{y,x}^{v,w}).$$

If a predicate  $X$  satisfies

$$[X \Rightarrow wp(p(e, v, w), R)]$$

for any procedure  $p$  with procedure body  $B$  satisfying  $[P \Rightarrow wp(B, Q)]$ , then

$$[X \Rightarrow U].$$

Another issue of vital importance for procedure call proof rules is soundness. The following soundness theorem is shown to hold for the rule of Bijlsma et al. [17]

**Theorem 5.1.3.2 (Soundness Theorem)** *Let  $U$  be the predicate*

$$(\exists m :: P_{e,v}^{x,y}) \wedge (\forall y, z :: (\forall m :: P_{e,v}^{x,y} \Rightarrow Q_e^x) \Rightarrow R_{y,z}^{v,w}).$$

For any procedure  $p$  with procedure body  $B$  satisfying  $[P \Rightarrow wp(B, Q)]$ , we have

$$[U \Rightarrow wp(p(e, v, w), R)].$$

The soundness of Gries's rule is proven in [136], while an informal argument to this effect is presented for Martin's rule in [114].

### Incorporating procedures into Dijkstra's calculus

A means of referring to the initial values of variables, specifically parameters, becomes indispensable in a specification language for guarded command programs using procedures. A formal mechanism for the introduction of logical constants, such as proposed in [121], is preferable in a mechanical verification environment (see Chapter 7).

In the above proof rules we placed no restriction on reference to the formal value parameters in the predicate  $Q$ . As pointed out in [66], the substitution  $Q_e^x$  used in these proof rules, only makes sense if  $x$  still has the initial value of  $e$  upon termination of the procedure body. This is partly achieved by our requirement that  $B$  be transparent to  $x$ . However, the use of structured data objects as actual parameters necessitates an additional requirement, namely that actual value parameters not be affected by assignments to other actual parameters during the execution of the procedure body. This kind of potential for interference between

actual parameters generally requires the generation of a verification condition to ensure that it does not realize, making it difficult and expensive to enforce mechanically. A pragmatic alternative is to eliminate potential problems by confining references to the value parameters in annotations of a procedure to the precondition, using logical constants elsewhere. Apart from being syntactically enforceable, the use of logical constants provide a formal, elegant, and sound way of adapting the procedure specification to use the same identifiers as the postcondition of a call (see [66, page 155] for an example).

A discussion in the previous chapter pointed out the need for a flexible type system for a guarded command language used with mechanical program development support. Procedures provide more evidence to this effect. To make procedures as generally applicable as possible and keep the use of expensive proof procedures to a minimum, it is in our interest to make the types of formal parameters as general as possible. Thus it is advantageous, for example, to consider the bounds of an array as part of the value of an array object and not as part of the type. General polymorphic types seem to offer the most valuable direction for further investigation on this matter [117].

#### **5.1.4 Functions**

User-defined functions are traditionally treated as special instances of procedures, because of the close correspondence between the syntax of their definitions. Their semantic status is, however, quite different. Procedure definitions create new program constructs, the meaning of which is defined in terms of weakest precondition predicate transformers, while a function definition creates a new function that may be applied to program variables in expressions within program constructs and annotations. The standard references for Dijkstra's weakest precondition calculus do not include a treatment of user-defined functions.

We may learn some lessons by considering user-defined functions in traditional references such as [30, 92, 47] using the partial correctness paradigm, but not all their results are applicable here. The total correctness semantic model as well as our nondeterminism assumption have effects on the formulation of proof rules for user-defined functions:

- From the references [7, 135] we find that the original function proof rule proposed by

Clint and Hoare [30] introduces unsoundness into Hoare logics, but not so under a total correctness interpretation.

- In [47] it is pointed out that standard algebraic simplification rules may fail for expressions involving user-defined “functions” in the presence of nondeterminism, since  $f(x) = f(x)$  is not necessarily true if  $f$  has a nondeterministic definition.

Mistakes in some published proof rules have highlighted the importance of substantiating all new rules with at least a proof of soundness. This falls beyond the scope of this treatise, thus I will not do more than make some suggestions for the contents of a suitable proof rule for user-defined functions in the weakest precondition calculus:

1. There are some options for the type of functions users are to be allowed to define. We may, for example, allow only function definitions consisting of a single expression, or we may allow functions to be defined by an arbitrary set of guarded commands that assign a value to a special “return” variable, representing  $f(x)$ . Functions that simply act as abbreviations for arbitrary complex expressions, are not as powerful as the more general “procedure-like” functions and their semantics will also be simpler, e.g. : they do not introduce nondeterminism and do not require a proof that the function declaration conforms to some given specification, as necessary for procedures.
2. To allow algebraic simplification of expressions to proceed as for ordinary mathematical functions, it may be worthwhile to impose a condition ensuring single-valuedness on the definitions of user-defined functions. An example of such a condition is found as one of the premises of the proof rule for Euclid function declarations [47, page 15]. To ensure that the order of evaluation of functions within an expression does not matter, user-defined functions should also not be allowed to have side effects.
3. Since function application can be used freely as part of expressions in guarded command programs, further possibilities for undefinedness are introduced and the domain predicate calculated as part of the (weakest) precondition of each program construct has to include the function precondition. For detailed discussions of this topic, see section 4.3.2 and specifically [33, Chapter 6].

4. Assume that function definitions have the following syntax:

$$\text{function } f \text{ (value } x : T) : \llbracket \{P\}B\{Q\} \rrbracket$$

similar to and with the same restrictions as for procedure declarations as explained above. Also assume that the function name is used as a variable to store the result of the function to be returned from a function call. Just as for procedures, it will be necessary to prove that such a declaration conforms to its given specification, i.e.

$$[P \Rightarrow wp(B, Q)].$$

We want to formulate a proof rule to help us verify properties of expressions containing an application  $f(e)$  of this function to a suitable expression  $e$ . As a basis, one may start with a stripped-down version of the Euclid function rule [47] (which corresponds to the proof rule of Clint and Hoare [30]), i.e.

$$\frac{P \Rightarrow wp(B, Q)}{P_e^x \Rightarrow Q_{f(e)}^f}$$

The Euclid rule contains various other premises, including those for recursive functions and single-valuedness. The remaining task is to determine which other premises or restrictions, if any, are required in the setting of the weakest precondition calculus.

### 5.1.5 Recursion

The omission of a programming concept as powerful and “fundamental” as recursion from Dijkstra’s calculus, has led to some criticism [78] and various proposals for its inclusion, e.g. [34, 114, 81, 133, 130, 8, 82, 83, 125, 121]. In [38] Dijkstra remarks that while the semantics of a repetitive construct can be defined in terms of a recurrence relation between predicates, general recursion requires a recurrence relation between predicate transformers. Recalling that repetition is theoretically and practically the most complicated construct of the basic calculus, it is clear that recursion will present new challenges to both our formal semantic machinery and its application to program developments. We elaborate on some of the issues.

**Semantics of recursive procedure calls**

In previous sections we have shown the development of a mechanism for procedural abstraction, including facilities for local variables and different kinds of parameters. Allowing recursive procedure calls will make our procedure constructs more powerful, but their already complicated semantics will become even more so. To make the transition more manageable we may first study the effects of recursion in isolation by considering only parameterless recursive procedures without local variables.

**Direct recursion**

Consider a procedure  $p$  declared as follows:

$$\begin{array}{l} \text{proc } p : \\ \{P\} B : \llbracket \dots; p; \dots \rrbracket \{Q\} \end{array}$$

We want to define  $wp(p, R)$  for a call  $p$  and a postcondition  $R$ . Using equation (20), we obtain:

$$\llbracket wp(p, R) \equiv wp(B, R) \rrbracket.$$

Because  $B$  contains another call  $p$ , we obtain a semantic equation that assumes the form:

$$\llbracket Y \equiv F(Y) \rrbracket, \tag{23}$$

for a predicate transformer  $Y$ .

This situation calls to mind our treatment of the semantics of iteration in Chapter 3. There we found a similar semantic equation, (7), of the form

$$\llbracket X \equiv f(X) \rrbracket, \tag{24}$$

for a predicate  $X$ .

We wish to follow the same procedure here, as we did for iteration, but there is a major difference: (23) is an equation in predicate transformers. In order to argue that  $F$  has a least fixed point and to obtain an expression for it, we will apply Kleene's Limit Theorem A.5.0.1

(see Appendix A.5). This theorem requires that  $F$  be a chain continuous function over a complete lattice. For this purpose we need to impose a lattice structure on predicate transformers. Our current semantic framework does not provide such a structure, but may easily be extended to this effect. This is done in Chapter 6. We also need to prove that  $F$  is indeed chain continuous. Examples of such proofs appear as Theorems 4.25 and 4.27 in [81]. From the Limit Theorem we then obtain the result that  $F$  has a least fixed point with

$$[\mu x.F(x) \equiv \bigsqcup \{F^n(\text{abort})\}_{n \geq 0}].$$

There are practical situations in which equation (23) reduces to the form (24). This occurs if  $wp(B, R)$  can be written as a function of  $wp(p, R)$ , since we then obtain:

$$[wp(p, R) \equiv wp(B, R) \equiv f(wp(p, R))]$$

for some predicate transformer  $f$ . Such recursions are called *tail recursions*. They can be implemented by procedures in which each recursive call is a dynamically last call.

### Indirect recursion

Consider procedures  $p$  and  $q$ , declared as follows:

**proc**  $p$  :  
 $\{P_p\} B_p : \llbracket \dots; q; \dots \rrbracket \{Q_p\};$

**proc**  $q$  :  
 $\{P_q\} B_q : \llbracket \dots; p; \dots \rrbracket \{Q_q\};$

For these we will obtain two mutually dependent semantic equations of the form:

$$\begin{aligned} [X &\equiv F(X, Y)] \\ &\text{and} \\ [Y &\equiv G(X, Y)] \end{aligned} \tag{25}$$

for predicate transformers  $X$  and  $Y$ .

These equations must be solved simultaneously. Following [130] we now define

$$(F, G)(x, y) \equiv (F(x, y), G(x, y))$$

for predicate transformers  $x$  and  $y$ .

Using results from Chapter 6 and Appendix A.3, it would be easy to show that  $(F, G)$  is a function on a complete lattice of predicate transformer pairs. We could then rewrite (25) as

$$[(X, Y) \equiv (F, G)(X, Y)].$$

As above, one has to prove that  $(F, G)$  is chain continuous (see [81]), before characterizing its least fixed point by:

$$\mu x.(F, G)(x) \equiv \bigcup \{(F, G)^n(\text{abort}, \text{abort})\}_{n \geq 0}.$$

This result can also be generalized to more than two mutually recursive procedures [81].

### **Introducing parameters**

To achieve full generality for recursive procedures, the above treatment can be extended to include parameters. The most comprehensive coverage of this topic is to be found in [81, Chapter 4]. This material combines results obtained for parameterized statements with those for parameterless recursive procedures. A definition of the relevant weakest precondition predicate transformer appears on page 155 of [81]. As expected, this definition is complex and offers no direct guidance in development of individual programs.

### **Proof rules**

As for non-recursive procedures, we want to use calls of recursive procedures as an abstraction mechanism. To achieve the desired abstraction, we formulate a proof rule that will enable us to prove procedure calls correct using only the information contained in the procedure specification. At the same time we will obtain a more practical method of doing correctness proofs than by appealing directly to the complicated semantic equation for the weakest precondition itself.

Proof rules for recursive procedures are inductive in nature. In essence we prove that the procedure body possesses a certain property on the hypothesis that every recursive call possesses it. Termination of a sequence of recursive calls is established, as for loops, by showing that each call decreases a non-negative integer-valued function (a bound function). An example of such a proof appears on pages 310–311 of [114]. Proof rules for recursion in the weakest precondition calculus are given in [114, 81, 130].

In [130], theorems are presented to show how the concept of a variant/bound function can be employed to simplify correctness proofs of recursive procedures. Mutually recursive procedures are also discussed in outline, but parameterization is not included. These results apply to a more general setting than the one under consideration here and are not directly applicable to our situation.

Recursive procedures, as we have approached them, are covered in [114, 81]. In [114], Martin explains how to go about the inductive correctness proof and gives a corresponding extension to his proof rule for non-recursive procedures (see section 5.1.3 above) to be used in the induction step. Mutual recursion is not addressed. A comprehensive, formal treatment of the semantics of recursive procedures is given in [81], including the formulation of proof rules for procedures with and without parameters. With some notational adaptations, the most general proof rule presented there is as follows:

$$\begin{array}{c}
 [(P_{e_1, v_1}^{x, y} \wedge 0 \leq t_{e_1, v_1}^{x, y} < x) \Rightarrow wp(p(e_1, v_1, w_1), Q_{e_1, v_1, w_1}^{x, y, z})], \\
 \vdots \\
 [(P_{e_n, v_n}^{x, y} \wedge 0 \leq t_{e_n, v_n}^{x, y} < x) \Rightarrow wp(p(e_n, v_n, w_n), Q_{e_n, v_n, w_n}^{x, y, z})] \\
 \vdash [(P \wedge t = x) \Rightarrow wp(B, Q)] \\
 \hline
 [(P_{e_0, v_0}^{x, y} \wedge 0 \leq t_{e_0, v_0}^{x, y}) \Rightarrow wp(p(e_0, v_0, w_0), Q_{e_0, v_0, w_0}^{x, y, z})]
 \end{array}$$

where  $x$  is an arbitrary non-negative integer and  $t$  represents the integer-valued bound function.

To handle calls for which a postcondition other than an instance of  $Q$  is to be used, this rule should be used in conjunction with a version of Hoare's Rule of Adaptation [85]:

$$\frac{[P \Rightarrow wp(p(e, v, w), Q)]}{[(\exists m :: P \wedge (\forall v, w :: Q \Rightarrow R)) \Rightarrow wp(p(e, v, w), R)]}$$

It is possible to combine these rules, but the result is quite unwieldy! The preconditions that they generate will be just as unwieldy, but that seems inescapable in the light of the inherently complex semantics of recursive constructs. Apart from this complexity, there are no obvious obstacles to the mechanical application of these proof rules. We would require that recursive procedures be syntactically recognizable as such (as is the case in languages like Pascal and Modula-2) so that a verification condition generator will know when to use the inductive proof rule instead of the ordinary one. Just as for loops, each recursive procedure should also be annotated with a bound function, as well as a logical constant that can be used to represent its initial value.

The reader should note that the above proof rule only covers the case of a single recursive procedure. Hemerik [81] mentions that it may be extended to cover mutually recursive procedures. Such an extension will bring about an additional increase in complexity; we will, for example, no longer be able to argue termination of a recursive call in isolation, but will require a bound function formulated in terms of all the procedure bodies involved.

### **5.1.6 Procedural Refinement**

In [120] an extra dimension is given to the discussion of procedural abstraction. As above, procedure call and parameterization are also introduced and studied separately. Additionally, the concept of procedural abstraction is formally introduced in its own right.

Procedure calls are still understood via the ALGOL-60 copy rule and parameterization is treated as an independent substitution mechanism. Parameters may be used to adapt any program fragment, including procedure calls and specifications, to specific circumstances. Procedural abstraction is described by a predicate pair, consisting of a precondition and a postcondition, that specify the effect of a mechanism upon the program state. Such specification statements are treated as “first-class citizens” of the language by giving their weakest precondition semantics as for any program construct. In addition, a notion of correctness-preserving procedural refinement is defined. This extends the domain of applicability of the copy rule to accommodate calls to procedures for which executable code has not been developed yet and, by formalizing the process of stepwise refinement, takes us into the territory of

a refinement calculus.

Formal support for procedural refinement is a natural next step in the process of introducing more powerful abstraction mechanisms into Dijkstra's calculus. It would be interesting to see to what extent the complexity of program development and correctness proofs may be reduced by such a measure as well as its demands on a mechanical support system. A refinement calculus with provision for the calculation of procedural refinements, is described in Chapter 7.

## **5.2 Data Abstraction**

Abstract programs (algorithms) should manipulate suitably abstract data. Though abstract data types may need to be represented in terms of more primitive types for implementation purposes, this need should have no effect on usage of the abstract objects. Data abstraction reduces complexity by separating the description of essential behavioural properties of data at a particular level from the representational detail of lower levels, organized to exhibit these properties. Thus we consider an abstract data type to be a class of data objects whereof the behaviour is completely characterized by a finite set of (abstract) operations that act upon members of the class.

Providing a formal mechanism for data abstraction allows factoring of program development into two (or more) stages:

1. Use the properties of an abstract data type, as given in its specification, during formal development of algorithms operating on objects of that type.
2. Develop a representation for the abstract data type such that the implementations of the abstract operations are consistent with their specifications.

There are two main approaches to the specification of data abstractions, sometimes referred to as "operational" and "definitional". Surveys of these approaches may be found in [73, 108]. The definitional approach involves specifying abstract data types by stating their desired properties instead of giving a method for constructing them.

Two prominent examples of definitional specification techniques are algebraic specifications [56, 44, 23, 74] and an axiomatic approach suggested by Hoare [86]. The algebraic approach uses equational axioms to specify properties of abstract data types. Data types are defined to be heterogeneous algebras. Hoare's method involves the use of predicate logic pre- and postconditions for describing the behaviour of each operation of an abstract data type and requires minimal adaptation to be incorporated into Dijkstra's calculus. A comparison of these two approaches appears in [73]. Here it is pointed out that problems requiring a data abstraction closely related to a data type available in the underlying specification language, are best handled using Hoare's method. Conversely if a data abstraction not readily represented by a data type in the specification language is required, algebraic techniques are superior. These conclusions emphasize the importance of very carefully selecting the data types to be included in the specification language if we are to use *specification techniques based on Hoare's suggestions*. The set of basic data types should be as rich as possible to allow the widest possible range of applicability. It is safe to assume that we will need such basic types as sets, bags, and sequences, but further study is necessary to determine other suitable candidates.

### 5.2.1 Hoare's Method

The following is a summary of Hoare's treatment of abstract data types as found in [86]:

#### Concepts

We consider a program that manipulates an abstract variable  $t$  of type  $T$ . For implementation purposes,  $t$  is represented by the concrete variables  $c_1, c_2, \dots, c_n$  whose types are more directly or efficiently implementable. The primitive operations on  $t$  are represented by procedures (and/or functions)  $p_1, p_2, \dots, p_m$  whose bodies operate on the concrete variables. Hoare uses the notation of the SIMULA 67 class declaration to express the concrete representation of a type  $T$  (see figure 7).

An abstract variable  $t$  of type  $T$  is declared using the notation:

```

class  $T$  :
  ||
    var
       $c_1 : \dots;$ 
       $c_2 : \dots;$ 
       $\vdots$ 
       $c_n : \dots;$ 
      proc  $p_1$  ("formal parameters") :  $||\{P_1\}B_1\{Q_1\}||;$ 
      proc  $p_2$  ("formal parameters") :  $||\{P_2\}B_2\{Q_2\}||;$ 
       $\vdots$ 
      proc  $p_m$  ("formal parameters") :  $||\{P_m\}B_m\{Q_m\}||;$ 
       $B$ 
    ||

```

where  $B$  represents an optional program fragment to initialize the concrete variables.

Figure 7: Concrete representation of a typical type  $T$ .

$\text{var } (T)t$

while operations on  $t$  are denoted by compound identifiers as follows:

$t.p_j(\text{"actual parameters"})$

The meaning of class declarations and procedure calls as above can be given in terms of textual substitution. For  $T$  of the form given in figure 7,  $\text{var } (T)t$  is equivalent to figure 8.

For an example illustrating these concepts, see [86] or [47].

### Correctness proofs

Consider the program

$||\text{var } (T)t; S||$

for which we wish to prove

```

var
     $t.c_1 : \dots;$ 
     $t.c_2 : \dots;$ 
     $\vdots$ 
     $t.c_n : \dots;$ 
proc  $t.p_1$  ("formal parameters") :  $[[\{P'_1\}B'_1\{Q'_1\}]]$ ;
proc  $t.p_2$  ("formal parameters") :  $[[\{P'_2\}B'_2\{Q'_2\}]]$ ;
     $\vdots$ 
proc  $t.p_m$  ("formal parameters") :  $[[\{P'_m\}B'_m\{Q'_m\}]]$ ;
     $\vdots$ 
 $B'$ 

```

where each of  $B'$  and the  $P'_i, B'_i$ , and  $Q'_i$  is obtained from the corresponding  $B, P_i, B_i$ , and  $Q_i$  by prefixing all occurrences of the concrete variable and procedure identifiers with " $t$ ".

Figure 8: Meaning of the abstract declaration  $\text{var } (T)t$ .

$$P\{[[\text{var } (T)t; S]]\}Q.$$

According to [86], this may be accomplished by showing:

1. "that the concrete representation given for the type  $T$  exhibits all the properties expected of it by the program  $S$ ";
2. that the abstract program,  $S$ , is consistent with its specification.

**Establishing correctness of the concrete representation** This involves showing that each operation of the form

$$t.p_j(a_1, a_2, \dots, a_k)$$

is equivalent to an assignment

$$t := f_j(t, a_1, a_2, \dots, a_k)$$

where  $f_j$  is a primitive operation on  $t$ , required by the abstract program.

Similarly, the initialization program  $B$  must be shown to be equivalent to an assignment of a suitable initial value,  $t_0$ , from the abstract space, to  $t$ .

To perform these proofs we need an expression of the relationship between the abstract and concrete spaces involved. This is given in the form of a representation function  $\mathcal{A}(c_1, c_2, \dots, c_n)$ , that maps the concrete variables to the abstract object represented by them. In many cases, the representation function is many-one.

Many practical situations also require use of an invariant  $\mathcal{I}(c_1, c_2, \dots, c_n)$ , that defines some relationship between the concrete variables. Initialization of  $T$  must be shown to establish  $\mathcal{I}$  and each abstract operation to maintain it.

Thus the following  $m + 1$  theorems have to be proven:

$$T\{B\}I \wedge \mathcal{A} = t_0$$

and

$$\mathcal{A} = t \wedge I \wedge P_j\{B_j\}Q_j \wedge I \wedge \mathcal{A} = f_j(t) \quad (26)$$

for all procedure bodies  $B_j$ .

**Establishing correctness of the abstract program** In proving  $P\{S\}Q$ , one will have to prove conditions of the form:

$$U\{t.p_j(a_1, a_2, \dots, a_k)\}V$$

To allow the desired abstraction in this proof, we may use theorems (26) above to formulate a formal proof rule that is applicable here. An example of such a proof rule for the Euclid language, may be found in [47].

## **Discussion**

There are many possible extensions to the class concept as described above, e.g. :

- Allow classes to have formal parameters that may be replaced with different actual parameters in declarations. This gives class definitions a wider range of applicability. Euclid modules include this facility and [47] may be referenced for the proof obligations resulting therefrom.
- A number of programming languages include a construct that allows support for both data abstraction and general encapsulation. Extending our view of the class formalism in this way, would allow the declaration of some concrete variables of a class as “private” and others as “public”. Euclid modules [47] allow both the declaration of local variables, that are not visible outside the scope of the module definition, and variables that are visible to the abstract program. A module's interface with programs that make use of it is stated explicitly through an “import” and an “export” list.
- The ability to define generic classes that may be used as the basis for constructing other classes would be very useful. One way to achieve this is to allow classes that define polymorphic types. A suggestion for modules with this property is described by Gries and Prins [68] (also see below).
- The class mechanism may be extended to allow treatment of recursive type definitions. This would allow construction and manipulation of types such as lists and trees in a natural manner.

The proof rule for Euclid modules as presented in [47] may be seen as an example of the formalized application of Hoare's suggestions concerning data abstraction. This rule is very complicated as it contains all the intricacies associated with correctness proofs for procedures, functions, and recursion. It has the form:

$$\frac{1, 2, 3, 4, 5, 6, 7, (8.1, 8.2, 8.3, 8.4) \vdash 8.5}{P\{\text{var } t : T(a); S\} R \wedge Q_1}$$

where premises 1–6 are properties required of the module definition, that need to be verified only once. Premise 7 states that the instantiation precondition is met and must be proven every time an abstract variable of the type concerned, is declared. Premise 8 uses the verified module definition (premises 1–6) to verify the uses of abstract variables of this type. These premises are described in more detail in [47] and [73].

Hoare's method does not provide any support for the development of data abstractions consistent with given specifications, but only for proving such consistency. Thus no guidance is provided in the formulation of a proposed data refinement. Only once it has been proposed, can its validity be verified. Guaranteed termination and nondeterminism in data types is also not considered. These issues are addressed by developing a weakest precondition semantics for data abstractions.

### 5.2.2 Gries-Prins Modules

In [68] Gries and Prins make some suggestions for extending the weakest precondition calculus with a module construct that may be used to describe data abstractions. This theory is developed further in [64].

A flexible type system, offering the following features, is assumed:

- A facility for type definitions: This should give the name of the type as well as the syntax of operations on values of the type. Figure 9 shows an example from [64].

```

type RATIONAL
  ||
    "+" : RATIONAL  $\mapsto$  RATIONAL prefix;
    "+" : RATIONAL  $\times$  RATIONAL  $\mapsto$  RATIONAL infix;
    "*" : RATIONAL  $\times$  RATIONAL  $\mapsto$  RATIONAL infix;
    "/" : RATIONAL  $\times$  RATIONAL  $\mapsto$  RATIONAL infix;
    IntToRat : integer  $\mapsto$  RATIONAL prefix;
    :
  ||

```

Figure 9: A type definition.

- Polymorphism: This allows the use of parameters for modules and procedures whereof the types may vary from one situation to another. The notation  $x : ?T$  is used to indicate that the type of identifier  $x$  can vary from use to use.
- Type inferencing: If the type of an identifier can be determined from the program, it need not be explicitly stated.

A Gries-Prins module is used to define a restricted implementation of (the value of) an abstract type. Different variables of the same type in an abstract program may be implemented using different implementation modules. This is indicated by the use of implementation directives in an abstract program:

```

var  $x, y$  : integer seq;
implement  $x$  by STACKARRAY(100);
implement  $y$  by LINKLIST;
in  $B$  ni

```

To allow a more flexible interface between an abstract program and implementation modules for the types used therein, procedures and functions are no longer used to implement data type operations. Parameterized statements and expressions are used instead.

A typical module has the form shown in figure 10. Here  $t$  is an abstract and  $c$  a concrete variable; more than one of each of which may be present in a module. The body of the module consists of pairs of abstract operations and their concrete implementations, of which one representative statement,  $S_t$  and one expression,  $f_t$ , with their respective implementations, are shown in this example.

```

module  $M$  ("parameters") :
[[
  var
     $t : T_t$  repr  $c : T_c$ ;
     $\vdots$ 
    {invar  $I(t, c)$ }
     $\square \{P_S\} S_t(t, \text{var } - x : T)$  into  $S_c(c, x')$ 
     $\vdots$ 
     $\square \{P_f\} f_t(t)$  into  $f_c(c)$ 
]]

```

Figure 10: A typical Gries-Prins module.

Patterns of the form:

```

var -  $x : T_x$ 
exp -  $e : T_e$ 
stmt -  $s$ 

```

are used in the definitions of abstract operations to allow simultaneous description of a class of operations. Such patterns specify a type or form restriction that must be obeyed by the text of the variable portions of these statements or expressions. The example in figure 10 shows that the second argument of  $S_t$  can be any variable  $x$  of type  $T$ . The corresponding variable in the implementation is  $x'$ .

### Developing correct modules

Abstract programs containing implementation directives may be developed with the aid of the weakest precondition calculus as any other guarded command program. No explicit weakest precondition semantics for such programs is given in [68], but the following proof rule for their correctness, is formulated:

$$\frac{[P \Rightarrow wp(B, Q)], M \text{ correct, } M \text{ and } B \text{ compatible}}{[P \Rightarrow wp(\text{implement } t \text{ by } M \text{ in } B \text{ ni}, Q)]}$$

A module  $M$  must be developed and shown correct relative to a representation invariant,  $I(t, c)$ , which is a predicate relating abstract and concrete values. This invariant may specify a function (compare to Hoare's representation function above), or in general, a relation.

- To be correct, each pair  $S_t, S_c$  must maintain (or establish, in the case of initializing operations) the representation invariant, thus ensuring that the values of  $t$  and  $c$  are always related by  $I$ . Other values produced by  $S_t$  and  $S_c$  (such as  $x$  and  $x'$  in figure 10) must always be equivalent. Thus development of a correct module involves finding suitable preconditions  $P_S$  such that

$$[P_S \Rightarrow wp((S_c(c, x'); S_t(t, x)), I \wedge x = x')]$$

It is pointed out in [68] that this definition is not general enough for nondeterministic operations. The following generalized rule can be used to determine  $P_S$  for any implementation that is at least as nondeterministic as the abstract operation. A full justification for its contents is to be found in [68].

$$[P_S \Rightarrow wp(S_c(c, x'), \neg wp(S_t(t, x), \neg(I \wedge x = x')))]$$

- Expression pairs  $f_t, f_c$  are simpler to find suitable preconditions for, since no state changes are involved. We use the rule:

$$[P_f \Rightarrow f_t(t) = f_c(c)]$$

A module  $M$  that implements abstract object  $t$ , using representation invariant  $I$ , is compatible with an abstract program  $\{P\}B\{Q\}$  if the following three conditions are met [68]:

- Every operation on the abstract variable  $t$  appearing in  $B$  must be matched within module  $M$ .
- An operation in  $B$  with precondition  $U$ , that is matched by a template in  $M$  with precondition  $P_S$ , must be shown to satisfy:
  - $U \Rightarrow P_S$ , if  $t$  is not free in  $U$ , or
  - $U \wedge I \Rightarrow P_S$ , if  $t$  is free in  $U$ .
- $t$  may not be free in  $P$  or  $Q$ .

Examples of developing modules in this style are to be found in [68] and [64].

Mechanical support for manipulation of Gries-Prins modules will require a high level of sophistication in order to be useful. A good structure editor is recommended for this purpose in [68]. Some areas that provide problems are:

1. Sophisticated pattern matching algorithms may be necessary to ensure flexibility in matching patterns in an abstract program with implementation templates in a module (during substitution of implementations for abstract operations).
2. Because both the type and implementation of variables and expressions have to match in a given context, one will have to provide both a type inference mechanism and an implementation inferencer. These must ensure that the type and implementation of variables and expressions are deduced from the text wherever possible.
3. Mechanisms for module interaction and interconnection are not defined (see [23] for examples of basic module interconnections and their (algebraic) semantics). It is therefore

not clear how to handle situations that require one module to reference representations given in other modules. Flexible measures for module connection provide scope for reuse of existing modules in new applications.

4. Aspects surrounding the type system need further clarification, e.g.

- which types, if any, should be considered as concrete types of the programming language;
- exactly what kind of type definitions will be allowed and
- how will recursively defined types be handled?

5. The use of abstract data types in programs present additional requirements for the specification language used with guarded command programs [143]. A useful “basic” set of abstract data types, together with suitable new notation for them, must be incorporated into the specification language and extension facilities for the language should be provided, to allow formulation of properties in terms of user-defined abstract types.

6. Mechanical proof support is significantly complicated by the use of abstract data types. In order to simplify and prove logical formulae that refer to abstract data types, a mechanical prover must provide facilities for constructing and effectively using theories of data types [94]. To stimulate reuse of theories it is also important to provide powerful facilities for building new theories from existing ones, e.g.

- extending a theory by adding items (sorts, constants, axioms, etc.);
- merging theories, possibly containing common elements;
- instantiating abstract parameters of a parameterized theory;
- generalizing (abstracting) from existing theories (so that the latter become instances of the new theory)

These requirements are non-trivial to accomplish (also see section 4.3.4 and [107, 98]).

7. The interface between a module and its environment must be rigorously defined. This may have an effect on the validity of data type induction, which is a valuable proof

technique for proving properties of abstract data types. How may we ensure that the principle of data type induction holds for a particular module? In Euclid, for example, this may be attained through careful use of the module import and export lists: if no “var” globals are imported and no portion of the internal data structures is exported as “var”, data type induction is valid over the type in question (see [73] and [47] for more details).

### 5.2.3 Data Refinement

Because the difference between an abstract type and the concrete types available in a programming language may be quite significant, a number of successive data refinements may be necessary before a suitable implementation for an abstract data type is obtained. Unless the “in-between” stages of data refinement have formal status, mechanical support for such developments can at best be ad hoc and proof obligations may not be factored over the development. A formal relation of data refinement, which allows development of an implementation by piecewise data refinement, is presented as part of the refinement calculus in Chapter 7. In [128] and [123] this refinement relation is used as the basis for a calculational style of program construction, which emphasizes the progressive *calculation* of data refinements as opposed to proposing them and then attempting to prove their correctness.

## 5.3 Abstraction in specifications

Due to various factors, guarded command program specifications are sometimes complex and lengthy. Here we want to consider specification language features that allow abstraction from some of the detail that is necessary for a formal proof of consistency between specification and program code.

The use of *abbreviations* for complicated or recurring predicates in constructing program specifications and annotations is quite well known. A first step towards this goal was taken in the prototype implementation by allowing each program annotation to be optionally labelled with a name that may be used instead of the actual assertion formula in other program

annotations (see section 4.2.1). An obvious extension to this practice is to generalize the abbreviations allowed so that not only program assertions themselves may be abbreviated, but also any other predicate to be used in program annotations. The following shortcomings of such abbreviations were observed:

- The predicate formula that is being abbreviated, is formulated in terms of the state space existing at the point where the abbreviation is defined. This causes problems in two situations:

1. Where an abbreviation is used in the scope of a quantifier, variable occurrences in the abbreviated formula may become bound, e.g. using the abbreviation

$$less \equiv (\forall i : 0 \leq i \wedge i < n : A[i] < B[i])$$

in the context:

$$(\forall (n : \text{character}) :: \dots less \dots).$$

2. Where an abbreviation is used in a local block, renaming of global variables appearing in the abbreviated formula cannot be allowed, e.g. using the abbreviation for *less* above, in the local block

$$[[\text{var } n : \text{character} \mid S]]$$

would be illegal.

Under these circumstances, the abbreviated formulas have to be scanned to locate any potential problems.

- When textual substitution is being performed on an assertion that uses abbreviations, abbreviations for formulas involving the variable(s) being substituted for have to be expanded to their full form. For example, in calculating  $P_{n+1}^n$ , where  $P$  contains a reference to the predicate *less*, as defined above, *less* will have to be expanded to allow substitution of  $n + 1$  for  $n$ . This means that all abbreviated formulas have to either be scanned beforehand to determine whether they contain references to such variables, or expanded automatically.

To solve these problems and provide more functionality for predicate abbreviations, parameterized abbreviations were introduced. Since no variables apart from the parameters are allowed in abbreviated formulas, the need to perform substitutions can be detected by examining the list of actual parameters and not the abbreviated formula itself. Clashes with quantification variables or local variables as mentioned above, cannot occur. The other advantage is that parameterized abbreviations are much more widely applicable. As with procedures, a more flexible type system, ideally incorporating polymorphism, will make parameterized abbreviations even more useful.

It is not clear during which stage of simplification or theorem proving, if at all, abbreviations in logical formulae should be expanded to their full form. One extreme is to expand all abbreviations before any simplification starts as this may allow more opportunities for simplification. This approach is simple to implement, but suffers from the obvious disadvantage that formulae will become longer and less readable. This is especially bad for human interaction. Mechanical simplifications on expanded formulae may leave them even less comprehensible to a human. The other extreme approach is to incorporate manipulation “rules” for abbreviations into the theorem proving process where they may be used for rewriting purposes. Such a process may allow “bigger” simplification steps and should be easier for a human to guide. On the other hand it would require a very sophisticated theorem prover. Allowing a user to formulate rules of manipulation for abbreviations will also raise concerns for their soundness, which may be difficult to establish. In [4, Chapter 8] some examples are given to illustrate how properties of abbreviated formulae may be used in mechanical simplification of verification conditions.

It is difficult to envision other abstraction facilities for specifications in the current setting. To adequately manage specifications of complicated problems we may need to decompose them into simpler “pieces”, while operations that allow the construction of complex specifications from simpler ones may also be useful. There is an obvious connection between decomposition of complex specifications and procedural and data refinement, which may be exploited during program construction within a unified framework such as the refinement calculus of Chapter 7. A specification language allowing such refinements should also have a rich set of suitably abstract data types to allow concise and natural formulation of abstract specifications. An

interesting example of powerful abstraction mechanisms in a specification language, is the schema calculus of Z [150]. In considering changes or extensions to the specification language used with guarded command programs, some consideration should be given to these or other abstraction facilities.

## **5.4 Conclusion**

Procedural abstraction, as provided by proof rules for procedure calls, provides a valuable way of decomposing program construction. The complexity of procedure proof rules stems from the inclusion of parameterization and recursion, but these may also be treated separately [120].

Another source of complexity, i.e. the couching of specifications and algorithms in terms of low level data structures, may be addressed by mechanisms for data abstraction, such as classes or modules. The accommodation of data abstraction is a more difficult problem than that of providing procedural abstraction and some details of approaches such as [68] have yet to be formalized before mechanical support can be considered.

The use of procedural and data abstraction facilities dictates the use of a more powerful specification language. A richer (and extendible) set of data types and facilities for composing and decomposing specifications seem to be required. Mechanical provers used for proofs of properties of data abstractions should have well developed mechanisms for the handling of theories of user-defined data types. Advantages of using procedural and data abstractions in mechanically supported program development include:

- they form a natural basis for a (rudimentary) “record” of program development;
- they may be used in limiting the proof obligations arising from changes to a program development.

*The use of procedural and data abstraction suggests a top-down method of program construction. In reality program development would proceed from an abstract specification through*

a suitable implementation is reached. The framework of Dijkstra's calculus does not accommodate formal applications of procedural and data refinements, although it may be used to establish the correctness of the results. The main obstacle is that the result of a refinement step that does not immediately produce program code, has no formal meaning and is therefore not amenable to formal correctness arguments. Thus we cannot use Dijkstra's calculus to answer crucial questions such as:

- Is a development step indeed a refinement, i.e. does it produce a result that is in some sense "closer to program code" than what we had before its application? This would require a formal notion of refinement between specification/program hybrids.
- Does a refinement step "preserve correctness"? Proof obligations may be factored over the development process if each refinement step can be shown to maintain correctness, instead of proving correctness only of the resulting code.
- Which refinement steps would be appropriate in a particular situation? This would imply a calculus that may be employed to "calculate" correct refinements.

Support for formal program development using procedural and data refinement, requires support for arbitrary levels of abstraction within the development calculus. This may be achieved by extending Dijkstra's calculus to a *calculus of refinement*.

## Chapter 6

# An Extended Semantic Framework

In Chapter 2 a lattice theoretical semantic framework for Dijkstra's programming calculus is presented. This framework rests on the lattice  $Pred_{Var}$  of predicates, which in turn is built on the lattice  $Bool$  of truth values.

As described in Chapters 4 and 5, some of the complexity imposed by the formality of mechanical application of the calculus may be countered by introducing more powerful mechanisms of abstraction. To accommodate program development by refinement of abstractions we have to add a further formal layer to our current framework, by constructing a lattice of predicate transformers based on  $Pred_{Var}$ . This extension allows an elegant formal treatment of general recursion and simplifies procedural and data abstraction by presenting a unified view of programs and specifications. The ordering imposed on predicate transformers, expresses the concept of "correctness preserving refinement". It offers a basis for the formulation of a *refinement calculus* in which programs may be developed from specifications by a series of formal applications of the stepwise refinement technique. Each construct in this progression retains formal status as part of the calculus.

### 6.1 A Lattice of Predicate Transformers

According to Chapter 2 a predicate transformer is a function that associates one predicate with another, i.e. assuming a state space  $\Sigma_{Var}$  based on a countable set  $Var$  of program

variables, a *predicate transformer on Var* is a function

$$P : Pred_{Var} \mapsto Pred_{Var}$$

We denote the set of all predicate transformers on  $Var$  by  $Ptran_{Var}$ . By results from Appendix A.3,  $(Ptran_{Var}, \sqsubseteq)$  is a complete lattice for the pointwise extended partial order on  $Pred_{Var}$ , i.e. for predicate transformers  $P_1$  and  $P_2$ :

$$[P_1 \sqsubseteq P_2 \equiv (\forall R : R \in Pred_{Var} : P_1(R) \sqsubseteq P_2(R))] \quad (27)$$

The top element of  $Ptran_{Var}$  is the predicate transformer **magic** which maps every predicate to the predicate  $\mathcal{T}$ , while the bottom element is the predicate transformer **abort** that maps every predicate to the predicate  $\mathcal{F}$ .

## 6.2 Healthiness Conditions Reconsidered

In shifting our attention from executable programs to abstract programs/specifications, we no longer require that all mechanisms be executable. This allows us to question Dijkstra's healthiness conditions for weakest precondition predicate transformers in order to gain more expressive power. A similar process has been followed by a number of authors, e.g. in [9, 8, 12, 25, 124, 122, 123, 130, 133], dropping some of the healthiness conditions has allowed treatments of specifications, logical constants, parallel programs, procedural and data refinement. We consider three such generalizations:

**Miracles:** In [133] Nelson extends Dijkstra's guarded command calculus by removing the Law of the Excluded Miracle. He calls the resulting semantic model the *general model*, as it is a generalization of the total correctness model that also includes so-called partial commands or "miracles", i.e. mechanisms that relate some initial state to no final state. Miracles are useful in that they simplify the program development process [124, 122] by allowing simpler applicability conditions for program refinements. They also allow proof of certain data refinements that are not provable in the total correctness model

[119]. Finally they lead to a simplification of the theory [88, 133, 130]. Unlike [133] we provide no operational interpretation for miracles—though they form part of the refinement calculus, they must be removed during program development, as executable programs are required to map every initial state to some final state(s).

**Nondeterminism:** Dijkstra's calculus encompasses mechanisms that display only bounded nondeterminism. This is a result of the requirement that the weakest precondition predicate transformers of all mechanisms under consideration be or-continuous [38]. Nondeterminism is a useful abstraction device in program development and may be used, in particular, where further refinement of an abstract program is to take place. In this situation it is somewhat artificial to allow only mechanisms that lead to choice from among a finite number of alternatives. To accommodate the needs of a program refinement calculus, we therefore drop the healthiness condition relating to or-continuity. Nevertheless, unbounded nondeterminism is not implementable and subsequent refinements must introduce suitable bounds in order to obtain an executable program.

**Conjunctivity:** To add expressive power, we will also allow programs  $S$  for which the knowledge that an initial state satisfies both  $wp(S, Q)$  and  $wp(S, R)$  is not a sufficient basis for concluding that a final state satisfying  $Q \wedge R$  will be established. Such programs violate Dijkstra's conjunctivity healthiness condition. This generalization is useful in that it allows a formal treatment of logical constants in terms of weakest precondition semantics (see Chapter 7).

Thus we shall limit our calculus to mechanisms such that their weakest precondition predicate transformers display the monotonicity and disjunctivity healthiness properties as set out in Chapter 2.

These weakest precondition predicate transformers are embedded in the lattice  $Ptran_{var}$ , or more specifically, in the lattice  $[Pred_{var} \rightarrow Pred_{var}]_M$  of monotonic predicate transformers. As shown in Appendix A.3,  $[Pred_{var} \rightarrow Pred_{var}]_M$  is a complete sublattice of  $Ptran_{var}$  and has the same bottom and top elements.

### 6.3 A Relation of Refinement

The ordering relation on (weakest precondition) predicate transformers, as defined in (27) above, is called the *refinement relation*. This relation is used in a number of reference works, including [9, 8, 12, 124, 122, 121, 130], and is essentially the same as the Smyth ordering [149]. To simplify notation, we formulate the definition of  $P_1 \sqsubseteq P_2$  for mechanisms (programs)  $P_1$  and  $P_2$  as follows:

$$[P_1 \sqsubseteq P_2 \equiv wp_{P_1} \sqsubseteq wp_{P_2} (\equiv (\forall R : R \in Pred_{Var} : wp_{P_1}(R) \Rightarrow wp_{P_2}(R)))] ,$$

thereby effectively identifying mechanisms with their weakest precondition predicate transformers. Informally we may give the meaning of  $P_1 \sqsubseteq P_2$  as:

**Mechanism  $P_2$  satisfies at least all specifications satisfied by  $P_1$ .**

This implies that  $P_2$  may be used as a replacement for  $P_1$  in every context where  $wp_{P_1}(R)$  has been shown to hold for a particular predicate  $R$ . We say  $P_2$  “refines”  $P_1$ .

Operationally,  $P_1 \sqsubseteq P_2$ , whenever  $P_2$  resolves nondeterminism in  $P_1$ , or terminates when  $P_1$  might not.

## Chapter 7

# A Program Refinement Calculus

Program development by stepwise refinement is a familiar concept in programming methodology [37, 156], during which a program is derived from an initial specification by carrying out a sequence of refinement steps. Both procedural and data refinements may be involved, ideally proceeding in parallel. The sequence of refinements used to construct a program form a detailed history of development that is useful when the resulting program must be changed or design decisions revised.

There are definite advantages to the formalization of this process, such that only formal, provably semantics-preserving refinements are carried out:

1. the resulting program is *correct by construction* (Thus an argument for the correctness of a program is decomposed according to the sequence of abstractions comprising the program development.);
2. program development may be supported by mechanical tools;
3. refinements of general importance may be recorded and reused in other developments;
4. general “patterns” of program development (spanning more than one refinement) may be constructed for reuse.

These are the basic ideas behind *transformational programming* [138, 48]. In this paradigm, programs are constructed by successive applications of correctness-preserving *transformation*

*rules*, starting with a formal specification and ending with a(n) (executable) program. Here a program transformation rule is a partial mapping from one program scheme (class of related programs) to another, such that a certain semantic relation holds between them. Predicates, called *enabling conditions*, are used to restrict the domains of partial transformation rules, i.e. they state proof obligations to be discharged when applying the rule.

The refinement calculus described here (and its theoretical framework in Chapter 6) is a generalization of Dijkstra's weakest precondition calculus, that embodies a formal notion of correctness-preserving stepwise refinement, with transformation rules for formal construction of programs from specifications. Pioneering work on this kind of calculus was performed by R.J.R. Back [9, 10, 8], with independent contributions by other researchers, most notably C. Morgan [124, 122, 121, 123] and J. Morris [130, 128]. As the notation of [9, 10, 8] differs substantially from the other two sources, most of the material presented here is drawn from the work of Carroll Morgan and Joseph Morris.

The first section introduces the concept of abstract programs and gives examples of how the guarded command language of Chapter 3 may be extended to express such programs. These extensions include generalizations of the semantics of local blocks and iteration, and new constructs such as specification statements, program conjunction, and modules. Data types suitable for use in abstract programs are also discussed.

Section two presents stepwise refinement as a formal manipulation on abstract programs. Both procedural and data refinement are considered. Laws of refinement are introduced as a mechanism for showing (and calculating) correct refinements of programs without reverting to definitions in terms of weakest preconditions. The constructive use of type information and general program invariants during program refinement, is also discussed.

The last section gives some perspectives on mechanical support for use of a refinement calculus.

## 7.1 Abstract Programs

During the process of constructing a program by stepwise refinement, a specification is systematically transformed into a program. To give formal status to the in-between stages of

program development, a unified view of programs and specifications is adopted. The notion of a program is generalized to encompass specifications. Thus we may view the stepwise refinement process as that of transforming an abstract program into a concrete/executable program, by forming a series of “increasingly concrete” programs in which abstract and concrete constructs are freely mixed. We will sometimes refer to executable programs as *code*.

The language used for expressing (abstract) programs must encompass both executable programming language constructs and high-level specification constructs. Specification languages are typically richer than programming languages, containing more expressive, not necessarily implementable, statements and a selection of richer data types. The language used for abstract programs in the refinement calculus, is built around the guarded command language of Dijkstra’s calculus, with more general semantics for some constructs (e.g. local blocks and iteration) and augmented with more powerful constructs, some of which are not implementable. A wider selection of data types is also allowed to encompass descriptions of abstract and concrete data. There is however, no formal standard for this language. In the sections to follow we describe typical features of such a language as found in the literature.

### 7.1.1 Language Extensions

#### A specification statement

Specification statements are obtained by giving a formal status to the “specifications” of section 3.4. They allow a uniform treatment of specifications and executable programs, which is the basis of the refinement calculus. The syntax of a typical specification statement is [124]:

$$w_1, \dots, w_n : [pre, post]$$

where  $w_1, \dots, w_n$  is a list of changing variables—called the *frame*, and *pre* and *post* are predicate formulas, respectively called the *precondition* and the *postcondition*.

Informally, this specification statement denotes a program that, if the initial state satisfies *pre*, will change only the variables  $w_1, \dots, w_n$  to attain a final state satisfying *post*.

To formalize the above, we give corresponding predicate transformer semantics [124]:

$$[wp(w : [pre, post], R) \equiv pre \wedge (\forall w :: post \Rightarrow R)] \quad (28)$$

As an informal justification of this choice of semantics, consider that *pre* describes the states in which termination of the above specification statement is guaranteed, and thus appears as the first conjunct of the weakest precondition. To ensure that *R* holds on termination, the second conjunct requires that, in all states in which *[pre, post]* terminates (described by *post*), *R* also holds. A formal justification of this semantics is given in [124].

Considering some extreme examples of specifications, it is clear that this construct allows us to write programs that violate some of the healthiness conditions of Dijkstra's calculus:

- A program called *magic*, that always terminates, establishing the impossible:

$$w : [T, \mathcal{F}]$$

*Magic* violates the Law of the Excluded Miracle. We adopt the convention of referring to all programs that violate this law as *miracles*.

- A program called *choose w*, that always terminates, but assigns any value of its choice to *w*:

$$w : [T, T]$$

*Choose w* violates the or-continuity healthiness requirement, for any *w* belonging to a type comprising infinitely many values.

Various abbreviations for special specification statements have been formulated, e.g. [121]

- A formula appearing as a conjunct of both the pre- and postconditions of a specification, may be written once, in between the two, i.e.

$$w : [pre, inv, post] \text{ means } w : [pre \wedge inv, post \wedge inv]$$

The formula *inv* above, is called a *specification invariant*.

- If the frame is empty and the postcondition is *T*, both may be omitted. We use *{pre}* instead of *: [pre, T]*

These commands are called *assumptions*. The assumption  $\{pre\}$  behaves like **skip** if  $pre$  is true; otherwise like **abort**.

- If the frame is empty and the precondition is  $T$ , both may be omitted, i.e.

$[post]$  means :  $[T, post]$

Such commands are called *coercions*. The above coercion behaves like **skip** if  $post$  is true; otherwise like the program magic.

### Local blocks

Without the need for or-continuity and the Law of the Excluded Miracle, we may generalize the semantics of the local block construct to a form that can be used to describe formal data refinements. We first consider:

$$[wp([[\text{var } x : T \mid S]], R) \equiv (\forall y :: wp(S_y^x, R))]$$

This semantics solves previous problems with uninitialized variables, but displays unbounded nondeterminism if the type  $T$  is infinite. For the treatment of data refinement to follow 7.2.2, it is also convenient to have explicit initialization of local variables. Adding this, we obtain the syntax [128, 123]:

$$[[\text{var } x : T \mid I \bullet S]]$$

where  $I$  is a predicate that must hold after initialization of the variable(s)  $x$ , and  $\bullet$  is used to separate the initialization predicate from the block body,  $S$ .

The corresponding semantics is as follows:

$$[wp([[\text{var } x : T \mid I \bullet S]], R) \equiv (\forall y :: I_y^x \Rightarrow wp(S_y^x, R))] \quad (29)$$

where  $y$  is a fresh variable.

Local blocks can also be miraculous, e.g.

$$[wp([[\text{var } x : T \mid \mathcal{F} \bullet S]], \mathcal{F}) \equiv T]$$

### Program conjunction

To formalize the use of logical constants, Morgan introduces the concept of *program conjunction* [123]. The generalized program conjunction of program  $S$  over a variable  $i$  of type  $T$ , is written:

$$[[\text{con } i : T \mid S]]$$

Informally, this construct chooses a value for  $i$  that makes subsequent preconditions true wherever possible, whereas a local block construct randomly chooses a value for a local variable. Consider, for example, the following program to increase the value of  $x$ :

$$[[\text{con } \textit{initval} : T \mid x : [x = \textit{initval}, x > \textit{initval}]]]$$

The weakest precondition semantics of program conjunction reflects this duality with (29):

$$[wp([[\text{con } i : T \mid S]], R) \equiv (\exists j :: wp(S_j^i, R))] \quad (30)$$

Logical constants are most often used to represent initial values of variables. Program conjunction eliminates the need for conventions such as capital letters or 0-subscripts to distinguish initial values of variables from the variables themselves and avoids the limitations of such an approach. Naming conventions for logical constants representing initial values may still be desirable as abbreviations, but these can now be defined formally (see e.g. [121, page 52]). Supplying type information with the introduction of each logical constant may be made compulsory, to allow simple mechanical type checking. Otherwise type inferencing would have to be applied when mechanical processing takes place.

Programs involving program conjunction may not obey Dijkstra's conjunctivity property for "healthy" predicate transformers.

### Procedures, parameters, and modules

The extended semantics of the refinement calculus gives more flexibility than before, in that procedural abstraction is now available through the use of specification statements. However,

procedure declarations are still important as a mechanism for avoiding repeated refinements of the same specification and for clearly exhibiting the structure of a program. Procedures and parameters in the context of the refinement calculus, are discussed in [120] and [121]. The same notation used in [66] and Chapter 5 may be adopted [121, Chapter 12]. The semantics of procedure calls is still based on the ALGOL-60 copy rule.

To allow parameterization of any command, systematic substitution of an expression (or variable) for another variable in a command is formally defined [120]. Substitution by *value*, *value result*, and *result* is included. The semantics is the same as for the parameterized statements in Chapter 5, except that references to global entities are no longer prohibited. A parameterized statement of Chapter 5

$$[[\text{value } x : T_1; \text{value result } y : T_2; \text{result } z : T_3 \mid S]](e, v, w)$$

would now simply appear in the form

$$S[\text{value } x : T_1, \text{value result } y : T_2, \text{result } z : T_3 \setminus e, v, w]$$

Modules are introduced into the refinement calculus in [121, Chapter 16]. They act as mechanisms of encapsulation and data abstraction and may be formulated to be largely independent of their environment, thus creating potential for their reuse in other contexts. Syntactically, these modules are close to those found in Euclid [47]. They consist of:

1. an *export list*, stating the names of all local variables and procedures accessible outside the module;
2. an *import list*, stating the names of all non-local variables and procedures accessible to the module;
3. a list of declarations of local entities, such as variables and procedures;
4. an *initialization*, that is a formula constraining the initial values of local variables.

No weakest precondition semantics is given for modules, but [121] shows how formal stepwise refinements may be used to transform an abstract *definition module* into a more concrete *implementation module* in order to obtain a correct implementation of a specification.

**Iteration and recursion**

Because Dijkstra's or-continuity requirement for predicate transformers does not apply to the refinement calculus, Kleene's Limit Theorem can no longer be used to show the existence of least fixed points for the semantic equations arising from loops and other tail recursive constructs. Instead, a generalized theorem due to Hitchcock and Park may be used (see Theorem A.5.0.2 in Appendix A.5). This theorem only requires monotonicity of the semantic functions, but characterizes the desired least fixed point in terms of iterated function composition over ordinals instead of the natural numbers which previously sufficed.

This necessitates a reformulation of the Fundamental Invariance Theorem for Loops 3.1.0.2. The integer-valued bound functions of Dijkstra's calculus are no longer general enough, so that we must now have a bound function  $t$  such that either

- $t : \Sigma_{Var} \mapsto (D, \sqsubseteq)$  for some poset  $(D, \sqsubseteq)$  and  $[(P \wedge GG) \Rightarrow (t \in C)]$  where  $C \subseteq D$  and  $(C, \sqsubseteq)$  is well-founded (see [132, 42] or [130]); or
- $t$  is an ordinal-valued function on the state space and  $[(P \wedge GG) \Rightarrow (t \in \lambda)]$  for some ordinal  $\lambda$  (see [25] or [130]).

At a glance, the first alternative seems the easier of the two for practical proofs, but only experimentation will show if this is actually the case. It is clear that ordinary first order logic is not general enough for the construction of such termination proofs as one cannot express the property of well-foundedness in it.

In [121], Morgan uses the notation

$$\text{re } p \mid B \text{ er}$$

for a recursive program  $p$ . Here  $p$  is a program name and  $B$  is a program scheme probably containing the command  $p$ . Viewing  $B$  as a function from  $Ptran_{Var}$  to  $Ptran_{Var}$ , the meaning of such a construct is given as  $\mu x.B(x)$ . The existence of this fixed point may be shown using Theorem A.5.0.2 (see Appendix A.5).

A method for constructing recursive procedures, with or without parameters, by formal step-wise refinement is given in [121]. In [130], theorems illustrating the validity of this approach,

are proven. The method involves showing termination of recursive procedures by finding a bound function of exactly the same nature as that for a loop (see above). Such a function must be shown to deliver a sequence of decreasing values in some well-founded set (possibly an ordinal) corresponding to the sequence of recursive calls. Mutual (indirect) recursion is also discussed (also see page 121).

### **7.1.2 Data Types**

A rich type system is required to allow formulation and refinement of suitably abstract specifications in the refinement calculus. The basis of the type discipline used in [121], is that any set is allowed to act as a data type.

Standard types may include truth values, characters, and various sets of numbers e.g. natural numbers, integers, rational numbers, reals, and complex numbers. New types may be constructed from existing ones by various set operators. Other compound data types used in [121] include bags, powersets, and sequences, while Cartesian products and maps are also used in [96]. Gries's work regarding data abstraction and program refinement [68, 64] suggests that it would also be useful to have a flexible mechanism for the definition of new abstract types, specifically allowing recursive type definitions and the use of type variables (for type polymorphism).

A clear delineation must be given as to which types may appear in code.

## **7.2 Formalized Stepwise Refinement**

### **7.2.1 Procedural (Algorithmic) Refinement**

The goal of procedural refinement is to reduce the expressive statements of abstract programs to (efficiently) executable ones. Using the refinement relation presented in Chapter 6, one may develop executable programs from their specifications through a sequence of formal, correctness preserving refinements.

According to this ordering, a program  $P_2$  refines a program  $P_1$  iff every specification satisfied

by  $P_1$  is also satisfied by  $P_2$ . Formally:

$$[P_1 \sqsubseteq P_2 \equiv (\forall R : R \in \text{Pred}_{\text{Var}} : \text{wp}_{P_1}(R) \Rightarrow \text{wp}_{P_2}(R))].$$

When we develop a program  $P$  from a specification  $X$  through successive correctness preserving refinements, we construct a sequence of programs related by the refinement relation as follows:

$$X \sqsubseteq P_1 \sqsubseteq \dots \sqsubseteq P_k \sqsubseteq P.$$

For this procedure to be sound, the refinement relation must be transitive. This property follows from the definition of the refinement relation (see Chapter 6).

We now consider two crucial strategies to be applied during stepwise program development based on this refinement relation.

### Top-down development

One powerful and well-known strategy that may be applied here, is top-down development, which combines the use of abstract program fragments with piecewise refinement.

We illustrate the technique on a typical step in the above development: Given an (abstract) program  $P_m$  ( $1 \leq m \leq k$ ), construct a program  $P_{m+1}$  such that  $P_m \sqsubseteq P_{m+1}$ . Assuming that

$$P_m = P_m[S_1, S_2, \dots, S_n]$$

contains the abstract programs  $S_1, S_2, \dots, S_n$ , we now use these as subgoals for the development process and construct programs  $T_1, T_2, \dots, T_n$  such that

$$(\forall i : 1 \leq i \leq n : S_i \sqsubseteq T_i).$$

Afterwards, piecewise refinement is applied, replacing each  $S_i$  in  $P_m$  by its corresponding  $T_i$ . This gives a new program

$$P_{m+1} = P_m[T_1, T_2, \dots, T_n]$$

which may be further refined if necessary. Continuing in this manner results in a “tree-shaped” development history.

To be sound, the above procedure must guarantee that  $P_m \sqsubseteq P_{m+1}$  holds. Thus we require that the refinement relation possess the following property: The set of programs (predicate transformers) under consideration admits piecewise refinement, i.e.

$$P[P_1] \sqsubseteq P[P_2]$$

holds whenever  $P_1 \sqsubseteq P_2$  holds. A proof of this property uses the fact that programs are built up from basic constructs such as assignment and skip statements, using program constructors such as sequential composition, conditional composition, and iteration. The proof consists of showing that each program constructor is monotone with respect to the refinement relation. An example of such a proof for a guarded command language may be found in [9].

Transitivity is also necessary to ensure that  $X \sqsubseteq P$  holds eventually.

### Refinement laws

If program development proceeds by proposing new refinements and proving them to follow from their ancestors using first principles, the burden of proof is unnecessarily high. To help in finding suitable refinements and reducing proof obligations, *refinement laws* are formulated. Such a law can be seen as a (partial) function:

$$RL : Ptran_{Var} \mapsto Ptran_{Var}$$

such that  $(\forall P : P \in Ptran_{Var} : P \sqsubseteq RL(P))$ . A refinement law is usually not defined for all possible programs, but applies only to programs conforming to given syntactic and/or semantic restraints.

It is convenient to present these laws in the form [124]

$$\frac{\text{before} - \text{refinement}}{\text{after} - \text{refinement}} \text{side} - \text{condition}$$

where *before-refinement* and *after-refinement* are “program templates” showing the form of programs to which the rule may be applied and that result from its application, respectively, while *side – condition* is a predicate that must be shown to hold for application of the rule to be valid. Thus, the rule means: “if *side – condition* is universally valid, then *before-refinement*  $\sqsubseteq$  *after-refinement* holds”.

A large selection of useful refinement laws is given as an appendix in [121]. These laws range from general ones, such as:

$$\frac{w : [pre, post]}{w : [pre', post]pre \Rightarrow pre'}$$

for weakening a precondition and

$$\frac{w, x : [pre, post]}{w : [pre, post]}$$

for contracting the frame, to laws for introducing specific executable constructs, e.g.

$$\frac{w, x : [pre, post]}{w := E} pre \Rightarrow post_E^w$$

$$\frac{w : [pre, post]}{[[\text{var } x : T \mid w, x : [pre, post]]]} w \text{ and } x \text{ are disjoint}$$

Use of refinement laws allows formal program development to take place at a higher level than when calculation of predicate transformers is used directly. The program development style is analogous to natural deduction proofs. For the sake of clarity, development steps based on application of a refinement law should be annotated with a reference to that law. Examples of program development using the refinement calculus may be found in [124, 125, 121].

Refinement generally strengthens a specification. Though over-strengthened specifications can never be refined to code (as they are miraculous), refinement laws provide no check against strengthening a specification too much. This keeps the laws as simple as possible, but allows one to embark on a sequence of unproductive refinements without noticing it. A separate feasibility test may be applied at a programmers discretion whenever such a situation is suspected. This test is based on Dijkstra's Law of the Excluded Miracle, which states

$$wp(P, \mathcal{F}) = \mathcal{F}$$

for all executable programs  $P$ .

If a specification is miraculous (and thus can never be refined to code) it will therefore fail the following feasibility test:

The specification  $w : [pre, post]$  is feasible iff

$$pre \Rightarrow (\exists w :: post)$$

### 7.2.2 Data Refinement

Data refinement is a special case of procedural refinement, that concentrates on replacing the abstract data types found in an abstract program with simpler or more efficiently implementable types, suitable for refined code. As in the foundational papers [128, 123], we view data refinement as a special case of refining a local block, i.e. a refinement transforming an abstract block

$$|[\text{var } t : T_t \mid I \bullet S]|$$

to a concrete block

$$|[\text{var } c : T_c \mid I' \bullet S']|$$

where  $I$  and  $S$  do not refer to the concrete variable(s)  $c$ , and  $I'$  and  $S'$  do not refer to the abstract variable(s)  $t$ . The essence of such a transformation is that the abstract variable  $t$  and operations on it are replaced with the concrete variable  $c$  and corresponding operations on  $c$ . Apart from  $t$  and  $c$  the local blocks may share variables—that are referred to as *common variables*.

A data refinement of the form shown above has two important characteristics:

1. The concrete block is a procedural refinement of the abstract block:

$$|[\text{var } t : T_t \mid I \bullet S]| \sqsubseteq |[\text{var } c : T_c \mid I' \bullet S']|$$

2. The concrete program  $S'$  retains the algorithmic structure of its abstract counterpart  $S$ . In simple terms, this means that data refinement will transform sequential composition in an abstract program to sequential composition in the concrete program, and similarly for local blocks, alternation, iteration, and recursion. Proofs to this effect may be found in [128] and [123]. This property is very important, because it ensures that piecewise data refinement (similar to piecewise procedural refinement) is a valid *modus operandi*.

A data refinement is facilitated by:

1. expressing the relationship between the abstract and concrete variables using a suitable relation,  $AI$ , called an *abstraction invariant*.  $AI$  may not refer to common variables that are assigned to by  $S$  or  $S'$ ;

2. choosing a concrete initialization  $I'$  such that [123]:

$$[I' \Rightarrow (\exists t :: AI \wedge I)].$$

The *data refinement relation* is formally defined as follows [123]:

A program  $S$  on abstract variable  $t$  is data refined by a program  $S'$  on concrete variable  $c$ , under the abstraction invariant  $AI$  (written  $S \preceq_{AI,t,c} S'$  or just  $S \preceq S'$ ) whenever

$$[(\exists t :: AI \wedge wp(S, X)) \Rightarrow wp(S', (\exists t :: AI \wedge X))]$$

for all  $X \in Pred_{Var}$  not containing free occurrences of  $c$ .

Two alternative formulations of data refinement were proposed by Gries and Prins [68] and Chen and Udding [29], respectively. These definitions and the Morris/Morgan/Gardiner version given above, are shown to be equivalent in [29]. Practical application of the different definitions with a view to comparing their ease of use may be found in [64] and [111]. Though the Gries/Prins and Chen/Udding data refinement rules are presented in a form that allows one to assume the truth of the abstraction invariant and prove a simpler formulation, they are not necessarily easier to apply in all situations.<sup>1</sup>

To avoid having to apply the, rather complicated, definition of data refinement to every refinement in practical situations, the same procedure is followed as for procedural refinement—laws of data refinement are derived. Examples of such laws, as well as their application in program development, may be found in [128, 123] (among others).

Another factor that considerably eases program development by the application of data refinement, is the trend to give data refinement laws a calculational style. The idea of calculating data refinements, rather than propose them and then prove them correct, originated in [89], where a relational setting is used instead of the predicate transformers under consideration here (also see [88]). A data refinement law is called a *calculator* [100, 123] if it delivers the weakest (most general) concrete program from the abstract program, under the given abstraction invariant. The following examples of data refinement calculators are from [123]:

<sup>1</sup>A similar simplification also applies to the Morris/Morgan/Gardiner form if the abstraction invariant is functional (see [128] or [123]).

*Initialization calculator: If  $S \preceq S'$  then*

$$[[\text{var } t : T_t \mid I \bullet S]] \sqsubseteq [[\text{var } c : T_c \mid (\exists t :: AI \wedge I) \bullet S']]$$

*Specification calculator: For all programs  $P$ ,*

$$t, x : [pre, post] \preceq P$$

*if and only if*

$$c, x : [(\exists t :: AI \wedge pre), (\exists t :: AI \wedge post)] \sqsubseteq P$$

As in procedural refinement, miracles again play an important role in data refinement. By not restricting data refinements only to those that do not introduce miracles, refinement calculators may be formulated without proof obligations (compare the alternation calculators of [128, Theorem 4] and [123, Lemma 6]). In [119] it is also shown that miracles allow proof of certain data refinements that were not otherwise provable. A detailed discussion of positive effects of miracles in the refinement calculus is presented in [122].

### 7.2.3 Types and Invariants

An extension to the refinement calculus, regarding the use of type information in program derivation, is presented in [125]. With this extension, typed local variable declarations affect the meaning of commands within their scope and program development may be formally aided by this information.

In [125], type information is treated as a special kind of *local invariant*. A programmer may declare any predicate formula to be a local invariant, using the notation:

$$[[\text{inv } I \mid S]].$$

This is taken to mean that the invariant  $I$  is assumed initially, automatically maintained by every command in  $S$ , and thus also established finally. While this is an example of an *explicit* invariant, declaration of local variables gives rise to *implicit* invariants, e.g. the local declaration

$$[[\text{var } x : T \mid \dots]]$$

implicitly introduces the invariant  $x \in T$ , preserved everywhere in its scope.

To allow typing information as well as any additional invariants stated by a programmer to have a formal effect on program development, the meaning of a guarded command program is now given relative to an invariant, called the *context*. The semantics of the usual program constructs is given in terms of  $wp_I$ : the weakest precondition in context  $I$  (see [125, page 282]). Thus we have for instance:

$$[wp_I(x := e, R) \equiv I \wedge (I \Rightarrow R)_e^x]$$

and

$$[wp_I([\text{inv } J \mid S], R) \equiv wp_{I \wedge J}(S, R)]$$

Taking the context  $I$  to be  $\mathcal{T}$ , results in the usual weakest precondition semantics for the guarded command language.

Procedural refinement in context,  $\sqsubseteq_I$ , is now also defined [125]:

$$[(P \sqsubseteq_I Q) \equiv [[\text{inv } I \mid P]] \sqsubseteq [[\text{inv } I \mid Q]]]$$

where  $\sqsubseteq$  is defined as follows [125]: For programs  $P$  and  $Q$  we have  $P \sqsubseteq Q$  iff

$$wp_I(P, R) \Rightarrow wp_I(Q, R)$$

for all postconditions  $R$  and all contexts  $I$ .

This is a not the refinement relation of Chapter 6 and section 7.2.1 above, although it shares many of the features of the latter and most refinement laws remain valid for the new relation (see [125] for details).

The advantage of introducing a local invariant  $J$ , is that within its scope the context may be strengthened, allowing one to use the refinement relation  $\sqsubseteq_{I \wedge J}$ , which is easier to establish than  $\sqsubseteq_I$  (see [125, Theorem 5.5]). At first it seems surprising that the imposition of additional invariants do not necessarily increase the proof obligations of program development. Morgan and Vickers show that invariants are automatically maintained during program development, making it unnecessary to prove this, e.g. calculating:

$$wp_{\mathcal{T}}([\text{inv } x \in \mathcal{N} \mid x := -5], x \in \mathcal{N})$$

$$\begin{aligned}
&\equiv wp_{x \in \mathcal{N}}(x := -5, x \in \mathcal{N}) \\
&\equiv x \in \mathcal{N} \wedge (-5 \in \mathcal{N} \Rightarrow -5 \in \mathcal{N}) \\
&\equiv x \in \mathcal{N}
\end{aligned}$$

we see that the invariant is maintained even though it appears to have been broken. Of course this is a miracle:

$$\begin{aligned}
&wp_T([[\text{inv } x \in \mathcal{N} \mid x := -5]], \mathcal{F}) \\
&\equiv wp_{x \in \mathcal{N}}(x := -5, \mathcal{F}) \\
&\equiv x \in \mathcal{N} \wedge (-5 \in \mathcal{N} \Rightarrow \mathcal{F}) \\
&\equiv x \in \mathcal{N}
\end{aligned}$$

A check should be applied to exclude any such miracles from the final program. In [125, Section 6] laws are given for the removal of explicit local invariants in order to obtain code. Part of this process is the discharging of certain proof obligations e.g. in the law [125]:

$$[[\text{inv } J \mid x := e]] \sqsubseteq_I x := e$$

for any context  $I$ , provided  $(I \wedge J) \Rightarrow J_e^x$ .

Thus removal of a local invariant will fail if the program is miraculous, e.g. for the program

$$[[\text{inv } x \in \mathcal{N} \mid x := -5]]$$

we cannot prove  $x \in \mathcal{N} \Rightarrow -5 \in \mathcal{N}$  and therefore cannot apply the above law to eliminate the invariant. This is a kind of type checking, although it encompasses more—compliance with arbitrary invariants may be established.

If only implicit invariants, due to local variable declarations, are present, mechanical type checking is enough to exclude miracles. Of course we require that for all types allowed in program code the type of any expression be mechanically deducible from the types of its constituent terms.

The formal treatment of types advocated by Morgan and Vickers not only allows typing information to play a constructive role during program development, but gives the ability to factor details, such as type (or feasibility) checking. This is a practical way of lightening the burden of proof associated with formal program development.

### 7.3 Some Perspectives on Mechanical Support

As with Dijkstra's calculus, mechanical support for application of the refinement calculus has to start with formalization of the calculus. The language used for expressing abstract programs is not formally defined in the literature and this needs to be addressed. Formalizations and implementations of the refinement calculus have been undertaken in HOL [13, 4], but both are quite limited and do not provide interactive mechanical tools for program development. Specific attention to the following aspects is necessary:

- The use of unbounded nondeterminism dictates that induction over either the ordinals or well-founded sets be used to prove termination of loops and recursions. Either or both of these should be accommodated in the language used for predicates.
- The type system must be formalized. A rich, flexible type system is necessary to allow both expressive, concise specifications and efficient implementations. To support stepwise refinement, abstract and concrete types should be allowed to coexist in programs. This approach is in the spirit of the *wide-spectrum specification/programming languages*, used in some program transformation systems, e.g. CIP-L [46] in CIP and *PA<sup>ada</sup>* [105] in PROSPECTRA. These languages incorporate a variety of constructs, ranging from high-level specification constructs to low-level, implementation-oriented ones and allow the construction of abstract programs that contain a mixture of these. Another interesting characteristic of CIP-L is that instead of a fixed set of specific data types, it has a general mechanism for introducing data types by algebraic specifications.

Another option worth investigating is the connection of a guarded command programming language with the specification language Z [157]. Such an approach will draw benefit from the fact that Z is a well established specification language and a number of mechanical tools for supporting its use have been developed.

As with Dijkstra's calculus, the refinement calculus is not intended as a set of rules that may simply be applied to a specification in a mechanical manner in order to produce a correct implementation. Thus tools to support its use should concentrate on efficient manipulation and management of details, without restricting the ability of the user to guide and control

the development process. Once again the degree of mechanical support provided may range from batch oriented systems that accept a “script” of program development and generate lemmas to validate its correctness (e.g. the tool set for VDM described in [102]) to interactive systems which allow a user to select and apply pre-proven refinement laws (and strategies), with the system generating lemmas from the side-conditions. Systems of the latter kind will face problems common to general interactive program transformation systems. Typical components of an interactive refinement support system would be (also see [138]):

**A component for storing, managing, and applying refinement laws:** Predefined refinement laws must be rapidly accessible and it is preferable to have facilities for extending the set of laws maintained by the system and possibly for combining existing laws into refinement strategies. Depending on the size of the law set, it may be necessary to divide it into smaller sets by purpose, e.g. separate procedural and data refinement laws and group together laws relating to data refinement using particular data types. Support for automated selection and application of laws is difficult and it is probably better to make the user responsible for selecting each individual law to apply. Different matching techniques may be implemented to allow the system to identify concrete instances of a law.

**A component for documenting the development process:** Depending on the sophistication of this facility, it may also be used to minimize the amount of reprocessing required to reflect changes to a program development.

**A component for generating lemmas and for proof support:** The generation of lemmas stems from side-conditions associated with refinement laws and should not be difficult to automate. Issues regarding proof support remain basically the same as for Dijkstra’s calculus. Meaningful interactive support for data refinement requires the ability to use a variety of data type theories efficiently in simplification of formulae (also see section 5.2.2).

**A component to support user interaction during program construction:** A structure editor-like interface seems suitable for the type of interaction required.

## Chapter 8

# Conclusion

The prototype implementation which was conducted for this investigation is too limited to be a practical tool for the construction of guarded command programs conforming to their specifications. Nevertheless, some valuable lessons have been learnt from its implementation and use:

Although Dijkstra's calculus has a formal foundation, it is used in an informal manner in the literature. To allow efficient and sound mechanical processing and formal reasoning about correctness, various aspects of the calculus must be formalized. These include the specification language, the set of data types used in program construction, initialization of variables, the use of logical constants and the treatment of undefined terms.

Apart from finding suitable formalizations for the above, meaningful mechanical support depends mainly on two interrelated aspects:

- The ability to counter the limitations imposed by complete formality.
- The ability to control complexity in formal program developments.

Measures to accomplish these goals include:

- Implementation of a rich set of data types and a powerful specification language is required.

- We also need efficient symbol manipulation procedures to calculate preconditions, generate verification conditions and simplify logical formulae. One of the strongest requirements for mechanical simplification and proof support is the ability of a prover/simplifier to effectively employ domain-specific knowledge arising from the use of various data types.
- A flexible approach to program development support is necessary. Interactive support is both feasible and desirable, but further investigation will be necessary to determine the extent and exact form such support should take. Provision of a repertoire of development strategies, as well as mechanisms for formulating and storing new strategies should be provided. A structure editor-like environment seems to be a suitable interface for such program development.
- The (strictly) controlled use of informality, e.g. in proofs of verification conditions will also help to counter excessive complexity.

Another way of combating complexity is to enhance the calculus itself by introducing suitable mechanisms for procedural and data abstraction. Some candidates were discussed in this thesis. However, a suitable return on such an investment will probably be achieved only if the extensions go as far as supporting arbitrary levels of abstraction during program development. This leads quite naturally to a refinement calculus such as those developed by J. Back, C. Morgan and J. Morris.

The refinement calculus allows a transformational approach to program development, which lends itself well to mechanical support. A number of general program transformation systems are in existence and have been used with some success. Because of the abstraction facilities provided in the refinement calculus and because proof obligations can be effectively factored over the development, it is hoped that program development will take place more smoothly and that fewer proofs, at a higher level, will be necessary. Some formalization of the calculus is necessary before interactive tools may be developed to support its use, but it is a worthy direction for future research efforts.

## Appendix A

# Basic Concepts, Notation and Terminology

**‘When I use a word,’ Humpty Dumpty said in rather a scornful tone, ‘it means just what I choose it to mean—neither more nor less.’**

**‘The question is,’ said Alice, ‘whether you *can* make words mean so many different things.’**

**‘The question is,’ said Humpty Dumpty, ‘which is to be master—that’s all.’**

*Lewis Carroll, Through the Looking Glass*

The definitions and properties of relations stated here, may be found in most introductory texts on algebra or discrete mathematics for computer science such as [16, 27, 154]. Posets and lattices are treated thoroughly in the works [19, 62]. The origins of the fixed point theorems used here, are discussed in [106] and the section dealing with ordinals is based on [116] and [101].

### A.1 Relations

A *binary relation*  $R$  on a set  $S$  is a subset of the Cartesian product  $S \times S$ . Given two elements in  $S$ , say  $x$  and  $y$ , we write  $xRy$  when  $(x, y) \in R$ . For simplicity we sometimes refer to a

binary relation simply as a relation.

Let  $x, y, z$  be any elements of a set  $S$  and  $R$  a binary relation on  $S$ . We say that the relation  $R$  is

reflexive	iff	$(\forall x : x \in S : xRx)$
irreflexive	iff	$(\forall x : x \in S : (x, x) \notin R)$
symmetric	iff	$(\forall x, y : x \in S \wedge y \in S : xRy \Rightarrow yRx) \cap$
antisymmetric	iff	$(\forall x, y : x \in S \wedge y \in S : (xRy \wedge yRx) \Rightarrow (x = y))$
transitive	iff	$(\forall x, y, z : x \in S \wedge y \in S \wedge z \in S : (xRy \wedge yRz) \Rightarrow xRz)$
linear	iff	$(\forall x, y : x \in S \wedge y \in S : xRy \vee yRx)$
well-founded	iff	there does not exist an infinite sequence $(x_i)_{i \in \mathcal{N}}$ of
(noetherian)		elements $x_i$ in $S$ such that $(\forall i : i \in \mathcal{N} : x_{i+1}Rx_i)$

A relation  $R$  on a set  $S$  is an *equivalence relation* if it is reflexive, symmetric and transitive. Such a relation partitions the set  $S$  into disjoint nonempty equivalence classes  $S_i$  as follows:  $S = S_1 \cup S_2 \cup \dots$ , and for each  $i$  and  $j$  such that  $i \neq j$ :

- $S_i \cap S_j = \emptyset$ ;
- $(\forall x, y : x \in S_i \wedge y \in S_i : xRy)$ ;
- $(\forall x, y : x \in S_i \wedge y \in S_j : \neg(xRy))$ .

Given sets  $A, B$  and  $C$  and relations  $R_1 \subseteq A \times B$  and  $R_2 \subseteq B \times C$ , the *composition* of  $R_2$  and  $R_1$  is the relation  $R_2 \circ R_1 \subseteq A \times C$ , defined as follows:

$$(\forall x, z : x \in A \wedge z \in C : x(R_2 \circ R_1)z \text{ iff } (\exists y : y \in B : xR_1y \wedge yR_2z))$$

For a relation  $R$  on a set  $S$ , we define:

$$R^n = \begin{cases} \text{the identity relation} & \text{if } n = 0 \\ R \circ R^{n-1} & \text{if } n > 0 \end{cases}$$

## A.2 Orderings

A relation  $\sqsubseteq$  on a set  $S$  is a *partial ordering* if it is reflexive, antisymmetric and transitive. A *partially ordered set* (or *poset* for short) is an algebraic structure  $(S, \sqsubseteq)$  consisting of a nonempty set  $S$  and a partial ordering  $\sqsubseteq$  on  $S$ . Where no ambiguity can result, we sometimes omit the ordering relation and simply say that  $S$  is a partially ordered set.

Let  $(S, \sqsubseteq)$  be a poset and  $T \subseteq S$ :

- $T$  has an *upper bound*  $u \in S$  iff  $(\forall x : x \in T : x \sqsubseteq u)$ .
- An element  $u$  in  $S$  is a *least upper bound* (*join*) (or *lub* for short) of  $T$  iff  $(\forall v : v \in S : (u \sqsubseteq v) \equiv (\forall x : x \in T : x \sqsubseteq v))$ , i.e.  $u \sqsubseteq v$  for every upper bound  $v$  of  $T$ . If it exists, the join of a set  $T$  is unique and is denoted by  $\sqcup T$ . We write  $x \sqcup y$  for  $\sqcup\{x, y\}$ .
- $T$  has a *lower bound*  $l \in S$  iff  $(\forall x : x \in T : l \sqsubseteq x)$ .
- An element  $l$  in  $S$  is a *greatest lower bound* (*meet*) (or *glb* for short) of  $T$  iff  $(\forall k : k \in S : (k \sqsubseteq l) \equiv (\forall x : x \in T : l \sqsubseteq x))$ , i.e.  $k \sqsubseteq l$  for every lower bound  $k$  of  $T$ . If it exists, the meet of a set  $T$  is unique.
- An element  $\perp$  in  $S$  is a *bottom* (*least element*) of  $S$  iff  $(\forall x : x \in S : \perp \sqsubseteq x)$ ; similarly an element  $\top$  in  $S$  is a *top* (*greatest element*) iff  $(\forall x : x \in S : x \sqsubseteq \top)$ .

A poset  $(S, \sqsubseteq)$  where the relation  $\sqsubseteq$  is linear, is called a *totally ordered set*. Such a relation is called a *complete partial order* (or *cpo*).

A *chain*  $T$  in a poset  $(S, \sqsubseteq)$ , is a nonempty subset which is totally ordered by  $\sqsubseteq$ . A function  $f$  on  $S$  is *chain continuous* iff

$$f(\sqcup T) \equiv \sqcup \{f(x) : x \in T\}$$

for all nonempty chains  $T$  such that both sides of the equation are defined.

A function  $f$  on a poset  $(S, \sqsubseteq)$  is *monotonic* iff

$$(\forall x, y : x \in S \wedge y \in S : x \sqsubseteq y \Rightarrow f(x) \sqsubseteq f(y))$$

### A.3 Lattices

A poset  $(L, \sqsubseteq)$ , in which every pair of elements has a meet and a join is called a *lattice*.

A *complete lattice* has the additional characteristic that each of its nonempty subsets has a meet and a join. It can be shown that:

- Every finite totally ordered set is a complete lattice.
- Every complete lattice has a bottom and a top.

A subset  $T$  of a complete lattice  $(L, \sqsubseteq)$  is a *complete sublattice* of  $L$  if all meets and joins of subsets of  $T$  are also in  $T$ .

If  $(L_1, \sqsubseteq^1)$  and  $(L_2, \sqsubseteq^2)$  are complete lattices, it can be shown that  $(L_1 \times L_2, \sqsubseteq)$  is also a complete lattice with  $\sqsubseteq$  defined by:

$$(x_1, x_2) \sqsubseteq (y_1, y_2) \equiv x_1 \sqsubseteq^1 y_1 \wedge x_2 \sqsubseteq^2 y_2.$$

Also, for any  $T \subset L_1 \times L_2$ ,

$$\bigsqcup T = (\bigsqcup \{x_1 : (\exists x_2 :: (x_1, x_2) \in T)\}, \bigsqcup \{x_2 : (\exists x_1 :: (x_1, x_2) \in T)\}).$$

This result can also be generalized to  $L_1 \times L_2 \times \dots \times L_n$ .

Let  $A$  be a nonempty set and  $(L, \sqsubseteq^L)$  a lattice. The set  $[A \rightarrow L]$  of functions from  $A$  to  $L$  is partially ordered by  $\sqsubseteq^f$ , called the *pointwise extension* of the partial order on  $L$  and defined by:

$$f \sqsubseteq^f g \equiv (\forall x : x \in A : f(x) \sqsubseteq^L g(x))$$

It can be shown that this makes  $[A \rightarrow L]$  a lattice which is complete iff  $L$  is complete.

For a poset  $(A, \sqsubseteq^A)$  and a lattice  $(L, \sqsubseteq^L)$ , the set of monotonic functions from  $A$  to  $L$  is denoted  $[A \rightarrow L]_M$ . It can be shown that  $([A \rightarrow L]_M, \sqsubseteq^f)$  forms a sublattice of  $[A \rightarrow L]$  and that, for a complete lattice  $L$ ,  $[A \rightarrow L]_M$  is also a complete lattice that has the same bottom and top elements as  $[A \rightarrow L]$ .

## A.4 Ordinals and Cardinals

A definition of the ordinals rest on axiomatic set theory and should be carefully constructed around the Russell paradox, see [116].

An ordinal is a set  $S$  with the properties

$$(\forall x : x \in S : x \subseteq S)$$

and

$$(\forall x, y : x \in S \wedge y \in S : x \in y \vee x = y \vee y \in x)$$

Following convention, we use small Greek letters to denote ordinals;  $\omega$  representing the first infinite ordinal, i.e. the set of natural numbers. There exist ordinals of arbitrarily large cardinality.

The membership relation forms a linear well-founded order on the class of ordinals. For ordinals  $\alpha$  and  $\beta$ , we use the notation

$\alpha < \beta$  for  $\alpha \in \beta$  and

$\alpha \leq \beta$  for  $\alpha = \beta \vee \alpha < \beta$

Because the class of ordinals is well-founded with respect to the ordering  $\leq$ , mathematical induction over the ordinals is valid. This is also called *transfinite induction*. One formulation of transfinite induction is as follows [116]:

Let  $ON$  denote the class of all ordinals and  $P(x)$  a predicate with free variable  $x$ . If, for any ordinal  $\beta$ , whenever  $P(\alpha)$  holds for all ordinals  $\alpha < \beta$ , then  $P(\beta)$  holds, then  $P(\lambda)$  holds for all  $\lambda \in ON$ :

$$(\forall \beta : \beta \in ON : (\forall \alpha : \alpha \in ON \wedge \alpha < \beta : P(\alpha)) \Rightarrow P(\beta)) \Rightarrow (\forall \lambda : \lambda \in ON : P(\lambda))$$

This is simply an instance of the principle of well-founded induction, where the ordering relation is linear.

For all ordinals  $\alpha$  and  $\beta$ , ordinal addition ( $\alpha + \beta$ ), multiplication ( $\alpha \times \beta$ ) and exponentiation ( $\alpha^\beta$ ) may be defined (see [116]).

The cardinal number of a set  $S$  is the smallest ordinal number cardinally equivalent to  $S$ .

The smallest infinite cardinal number is  $\omega = \omega_0$ . If  $\theta$  is a nonzero ordinal number, then we use  $\omega_\theta$  to denote the smallest cardinal number greater than all  $\omega_\xi$  for  $\xi < \theta$ .

## A.5 Fixed Points

Let  $(A, \sqsubseteq)$  be a poset and  $f$  a function  $f : A \rightarrow A$ . An element  $x \in A$  is called a *fixed point* of  $f$  iff  $f(x) = x$ . An element  $x \in A$  is called a *least fixed point* of  $f$  iff  $f(x) = x \wedge (\forall y : y \in A : f(y) = y \Rightarrow x \sqsubseteq y)$ . If the least fixed point of  $f$  exists, it is denoted by  $\mu x.f(x)$ .

**Theorem A.5.0.1 (Limit Theorem (Kleene))** *If  $f$  is a chain continuous function over a complete lattice, then  $f$  has a least fixed point which is  $\bigsqcup \{f^n(\perp)\}_{n \geq 0}$ .*

*(This is a modified version of the original theorem, see [106, 133, 103].)*

**Theorem A.5.0.2 (Generalized Limit Theorem (Hitchcock and Park))** *Given a complete lattice  $L$  with bottom  $\perp$  and a function  $f \in [L \rightarrow L]_M$ . We define:*

*$f^0(\perp) = \perp$ ,  $f^\lambda(\perp) = \bigsqcup \{f(f^\gamma(\perp)) : \gamma < \lambda\}$  for ordinal  $\lambda \neq 0$ .*

*Then there exists an ordinal  $\alpha$ , such that  $\mu x.f(x) = f^\alpha(\perp)$ .*

*(This is a modified version of the original theorem, due to Nelson, see [133].)*

## Appendix B

# A CFG for Annotated Guarded Command Programs

The following notational conventions apply:

- Terminal symbols are written in quotations.
- The notation  $\{item\}$  is used to denote  $item \mid \epsilon$ .

```

Program    ::=  "PROGRAM" "id" ":" Block
Block      ::=  "[{Preamble} {Assertion} VerUnitList "]" |
Preamble   ::=  "CONST" ConstList {"TYPE" TypeList}
                  {"VAR" VarList} {"{" "ABBREV" AbbrevList "}" "}" "}"
                  "TYPE" TypeList {"VAR" VarList}
                  {"{" "ABBREV" AbbrevList "}" "}" "}"
                  "VAR" VarList {"{" "ABBREV" AbbrevList "}" "}" "}"
                  {"{" "ABBREV" AbbrevList "}" "}" "}"
ConstList  ::=  "id" "=" Constant {";" ConstList}
Constant   ::=  {Sign} "integer" | "character" | "TRUE" | "FALSE"
Sign       ::=  "+" | "-"

```

**APPENDIX B. A CFG FOR ANNOTATED GUARDED COMMAND PROGRAMS 175**

<b>TypeList</b>	<b>::=</b>	<b>TypeDecl {“;” TypeList}</b>
<b>TypeDecl</b>	<b>::=</b>	<b>“id” “=” CompTypeSpec</b>
<b>CompTypeSpec</b>	<b>::=</b>	<b>“ARRAY” “[ {Sign} Bound “..” {Sign} Bound “]” “OF” TypeSpec   “RECORD” “[ ” VarList “ ]”</b>
<b>TypeSpec</b>	<b>::=</b>	<b>“id”   CompTypeSpec</b>
<b>Bound</b>	<b>::=</b>	<b>“id” {Sign “integer”}   “integer” {Sign “integer”}</b>
<b>VarList</b>	<b>::=</b>	<b>VarDecl {“;” VarList}</b>
<b>VarDecl</b>	<b>::=</b>	<b>IdList “:” TypeSpec</b>
<b>IdList</b>	<b>::=</b>	<b>“id” {“,” IdList}</b>
<b>AbbrevList</b>	<b>::=</b>	<b>Abbreviation {“;” AbbrevList}</b>
<b>Abbreviation</b>	<b>::=</b>	<b>“id” “(” ParList “)” “=” AssertExpr</b>
<b>ParList</b>	<b>::=</b>	<b>ParDecl {“;” ParList}</b>
<b>ParDecl</b>	<b>::=</b>	<b>IdList “:” “id”</b>
<b>Assertion</b>	<b>::=</b>	<b>“{” {“id” “:”} AssertExpr “}”</b>
<b>AssertExpr</b>	<b>::=</b>	<b>SimpleAssert {“=&gt;” AssertExpr}</b>
<b>SimpleAssert</b>	<b>::=</b>	<b>AssertDisj {Relop SimpleAssert}</b>
<b>Relop</b>	<b>::=</b>	<b>“=”   “&gt;=”   “&gt;”   “&lt;=”   “&lt;”   “&lt;&gt;”</b>
<b>AssertDisj</b>	<b>::=</b>	<b>AssertConj {Orop AssertDisj}</b>
<b>Orop</b>	<b>::=</b>	<b>“+”   “-”   “OR”   “COR”</b>
<b>AssertConj</b>	<b>::=</b>	<b>AssertTerm {Andop AssertConj}</b>
<b>Andop</b>	<b>::=</b>	<b>“AND”   “CAND”   “DIV”   “MOD”   “*”</b>
<b>AssertTerm</b>	<b>::=</b>	<b>{Notop} AssertFactor</b>
<b>Notop</b>	<b>::=</b>	<b>“+”   “-”   “~”</b>
<b>AssertFactor</b>	<b>::=</b>	<b>“id” VarSpec   “(” AssertExpr “)”   “(” QuantExpr “)”   “integer”   “character”   “TRUE”   “FALSE”</b>
<b>VarSpec</b>	<b>::=</b>	<b>“(” ArgList “)”   “(” ValueChanges “)” Selector   {Selector}</b>
<b>ArgList</b>	<b>::=</b>	<b>AssertExpr {“,” ArgList}</b>

**APPENDIX B. A CFG FOR ANNOTATED GUARDED COMMAND PROGRAMS 176**

<b>ValueChanges</b>	<b>::=</b>	<b>Selector “:” Expr {“,” ValueChanges}</b>
<b>Selector</b>	<b>::=</b>	<b>“[” Expr “]” {Selector}   “.” “id” {Selector}</b>
<b>QuantExpr</b>	<b>::=</b>	<b>LogQuant “(” “id” “:” “id” “)” “:” {Expr} “:” AssertExpr   NumQuant “(” “id” “:” “id” “)” “:” “(” BoundExpr “)” “AND” “(” BoundExpr “)” “:” AssertExpr</b>
<b>LogQuant</b>	<b>::=</b>	<b>“FORALL”   “EXISTS”</b>
<b>NumQuant</b>	<b>::=</b>	<b>“NUMBER”   “SUM”   “PRODUCT”  </b>
<b>BoundExpr</b>	<b>::=</b>	<b>Disjunction LessGreater “id”   “id” LessGreater Disjunction</b>
<b>LessGreater</b>	<b>::=</b>	<b>“&gt;=”   “&gt;”   “&lt;=”   “&lt;”</b>
<b>Expr</b>	<b>::=</b>	<b>Disjunction {Relop Disjunction}</b>
<b>Disjunction</b>	<b>::=</b>	<b>Conjunction {Orop Disjunction}</b>
<b>Conjunction</b>	<b>::=</b>	<b>Term {Andop Conjunction}</b>
<b>Term</b>	<b>::=</b>	<b>{Notop} Factor</b>
<b>Factor</b>	<b>::=</b>	<b>CompVarSpec   “(” Expr “)”   “integer”   “character”   “TRUE”   “FALSE”</b>
<b>CompVarSpec</b>	<b>::=</b>	<b>“id” {Selector}</b>
<b>VerUnitList</b>	<b>::=</b>	<b>Statement {Assertion} {“;” VerUnitList}</b>
<b>Statement</b>	<b>::=</b>	<b>“SKIP”   “ABORT”   AssignList   Alternation   VarInvar Repetition   Block</b>
<b>AssignList</b>	<b>::=</b>	<b>VarSpec Assign Expr</b>
<b>Assign</b>	<b>::=</b>	<b>“:=”   “,” AssignList “,”</b>
<b>Alternation</b>	<b>::=</b>	<b>“IF” {GuardComSet} “FI”</b>
<b>GuardComSet</b>	<b>::=</b>	<b>GuardCom {“ ” GuardComSet}</b>
<b>GuardCom</b>	<b>::=</b>	<b>Expr “-&gt;” VerUnitList</b>
<b>VarInvar</b>	<b>::=</b>	<b>Invariant Variant   Variant Invariant</b>
<b>Invariant</b>	<b>::=</b>	<b>“{” “INVAR” {“id” “:”} AssertExpr “}”</b>
<b>Variant</b>	<b>::=</b>	<b>“{” “BOUND” {“id”} “INIT” “id” “:” AssertExpr “}”</b>
<b>Repetition</b>	<b>::=</b>	<b>“DO” {GuardComSet} “OD”</b>

## Appendix C

### The predicate $\text{domain}(e)$

The *domain* predicate is defined for all expressions  $e$  appearing in programs. We want  $\text{domain}(e) \equiv \mathcal{T}$  iff  $e$  is well-defined. The definition is based on the BNF for the nonterminal “Expr” in the grammar of Appendix B. It does not include compound data objects such as arrays of arrays, arrays of records and records with array fields.

Based on the rule

$\text{Expr} ::= \text{Disjunction} \{ \text{Relop Disjunction} \}$

we start by defining:

$\text{domain}(\text{Disjunction1 Relop Disjunction2}) \equiv \text{domain}(\text{Disjunction1}) \wedge \text{domain}(\text{Disjunction2})$

For “Disjunctions” we have two separate cases:

***domain*(Conjunction “+” Disjunction)**

$$\begin{aligned} &\equiv \text{domain}(\text{Conjunction “-” Disjunction}) \\ &\equiv \text{domain}(\text{Conjunction “OR” Disjunction}) \\ &\equiv \text{domain}(\text{Conjunction}) \wedge \text{domain}(\text{Disjunction}) \end{aligned}$$

***domain*(Conjunction “COR” Disjunction)**

$$\equiv \text{domain}(\text{Conjunction}) \text{ cand } (\neg \text{Conjunction} \Rightarrow \text{domain}(\text{Disjunction}))$$

For “Conjunctions” we have:

***domain*(Term “DIV” Conjunction)  $\equiv$  *domain*(Term “MOD” Conjunction)**

$$\equiv \text{domain}(\text{Term}) \wedge (\text{domain}(\text{Conjunction}) \text{ cand } \text{Conjunction} \neq 0)$$

***domain*(Term “CAND” Conjunction)**

$$\equiv \text{domain}(\text{Term}) \text{ cand } (\text{Term} \Rightarrow \text{domain}(\text{Conjunction}))$$

***domain*(Term “\*” Conjunction)  $\equiv$  *domain*(Term “AND” Conjunction)**

$$\equiv \text{domain}(\text{Term}) \wedge \text{domain}(\text{Conjunction})$$

A “Term” consists of a “Factor” or its negation:

***domain*(Notop Factor)  $\equiv$  *domain*(Factor)**

We divide “Factors” into three groups according to how they are constructed. For these we define:

$$\begin{aligned} \text{domain}(\text{“id”}) &\equiv \text{domain}(\text{“id”“.”“id”}) \equiv \text{domain}(\text{“integer”}) \\ &\equiv \text{domain}(\text{“character”}) \equiv \text{domain}(\text{“TRUE”}) \equiv \text{domain}(\text{“FALSE”}) \\ &\equiv \mathcal{T} \end{aligned}$$

**APPENDIX C. THE PREDICATE DOMAIN(*E*)****179**

$$\text{domain}(\text{"id" "[" Expr "]"})$$

$$\equiv \text{domain}(\text{Expr}) \text{ and } (\text{"id"}.lob \leq \text{Expr} \wedge \text{Expr} \leq \text{"id"}.hib)$$

$$\text{domain}(\text{"(" Expr ")"}) \equiv \text{domain}(\text{Expr})$$

## Appendix D

# Substitution in Program Constructs

The substitution of a variable  $y$  for a variable  $x$  in a program construct  $C$ , is denoted  $C_y^x$ . The following definition is based on Definition 4.6 in [81, Chapter 4] as well as the grammar in Appendix B. Wherever substitution in expressions is indicated, ordinary textual substitution should be used:

$$\begin{aligned}
 \text{abort}_y^x &= \text{abort} \\
 \text{skip}_y^x &= \text{skip} \\
 (z := \text{expr})_y^x &= z_y^x := \text{expr}_y^x \\
 ([\text{var } z : T \mid S])_y^x &= \\
 &\quad \begin{cases} [\text{var } z : T \mid S] & \text{if } x = z \\ [\text{var } z : T \mid S_y^x] & \text{if } x \neq z \wedge y \neq z \\ [\text{var } v : T \mid (S_v^z)_y^x] & \text{if } x \neq z \wedge y = w \end{cases} \\
 (S1; S2)_y^x &= S1_y^x; S2_y^x \\
 (\text{if } G1 \rightarrow S1 \square \dots \square Gn \rightarrow Sn \text{ fi})_y^x &= \\
 &\quad \text{if } G1_y^x \rightarrow S1_y^x \square \dots \square Gn_y^x \rightarrow Sn_y^x \text{ fi} \\
 (\text{do } G1 \rightarrow S1 \square \dots \square Gn \rightarrow Sn \text{ od})_y^x &= \\
 &\quad \text{do } G1_y^x \rightarrow S1_y^x \square \dots \square Gn_y^x \rightarrow Sn_y^x \text{ od}
 \end{aligned}$$

## Appendix E

# An Annotated Program and its VCs

The solution to Rubin's problem shown in section 3.5, is presented here in the format of a text file as accepted by the prototype implementation. Line numbers are given for reference purposes.

```
1  PROGRAM Rubin :
2  |[
3  CONST
4      MAXCOL = 10;
5      MAXROW = 10
6
7  TYPE
8      TABLE = ARRAY [1..MAXROW] OF ARRAY [1..MAXCOL] OF INT
9
10 VAR
11     matrix          : TABLE;
12     found            : BOOL;
13     lastrow, lastcol : INT;
14     curcol, currow   : INT
```

## APPENDIX E. AN ANNOTATED PROGRAM AND ITS VCS

182

```

15
16 {ABBREV
17   nonzero(matrix : TABLE; maxcol, rowpos, colpos, lastcol : INT) =
18     (FORALL (col : INT) : (0 < col) AND (col <= colpos) :
19       matrix[rowpos][col] = 0) AND ((lastcol = maxcol) COR
20       (matrix[rowpos][colpos+1] <> 0));
21
22   allzero(matrix : TABLE; maxcol : INT; rowpos : INT) =
23     (FORALL (col : INT) : (0 < col) AND (col <= maxcol) :
24       matrix[rowpos][col] = 0);
25
26   inbounds(maxindex, index, lastindex : INT) =
27     (0 <= index) AND (index <= lastindex) AND (lastindex <=
28       maxindex)
29 }
30 |
31 {(MAXROW > 0) AND (MAXCOL > 0)}
32
33 currow, lastrow := 0, MAXROW;
34
35 {INVAR inbounds (MAXROW, currow, lastrow) AND
36   (FORALL (row : INT) : (0 < row) AND (row <= currow) :
37     (EXISTS (col : INT) : (0 < col) AND (col < MAXCOL) :
38       nonzero(matrix, MAXCOL, row, col, col))) AND ((lastrow = MAXROW) COR
39       allzero(matrix, MAXCOL, currow+1))}
40 {BOUND t1 INIT T1 : lastrow-currow}
41
42 DO currow <> lastrow -> curcol, lastcol := 0, MAXCOL;
43
44   {INVAR inbounds (MAXROW, currow, lastrow) AND
45     (FORALL (row : INT) : (0 < row) AND (row <= currow) :

```

## APPENDIX E. AN ANNOTATED PROGRAM AND ITS VCS

183

```

46      (EXISTS (col : INT) : (0 < col) AND (col < MAXCOL) :
47      nonzero(matrix,MAXCOL,row,col,col))) AND (lastrow = MAXROW)
48 AND (currow <> lastrow) AND (t1 = T1)
49      AND inbounds(MAXCOL, curcol,lastcol) AND nonzero (matrix,MAXCOL,
50 currow+1,curcol,lastcol)})
51      {BOUND t2 INIT T2 : lastcol-curcol}
52
53 DO curcol <> lastcol ->
54     IF matrix[currow+1][curcol+1] = 0 -> curcol := curcol+1
55     | matrix[currow+1][curcol+1] <> 0 -> lastcol := curcol
56     FI
57 OD;
58
59     {inbounds (MAXROW, currow, lastrow) AND
60     (FORALL (row : INT) : (0 < row) AND (row <= currow) :
61     (EXISTS (col : INT) : (0 < col) AND (col < MAXCOL) :
62     nonzero(matrix,MAXCOL,row,col,col))) AND (lastrow = MAXROW)
63     AND (currow <> lastrow) AND (t1 = T1) AND (curcol = lastcol)
64     AND nonzero (matrix,MAXCOL,currow+1,curcol,lastcol)})
65
66     IF curcol = MAXCOL -> lastrow := currow
67     | curcol <> MAXCOL -> currow := currow+1
68     FI
69 OD;
70
71 found := currow <> MAXROW
72
73 {(FORALL (row : INT) : (0 < row) AND (row <= currow) :
74 (EXISTS (col : INT) : (0 < col) AND (col < MAXCOL) :
75 nonzero(matrix,MAXCOL,row,col,col))) AND ((lastrow = MAXROW) COR
76 allzero(matrix,MAXCOL,currow+1)) AND (lastrow = currow) AND

```

```

77   (found = (currow <> MAXROW))}
78 ]|

```

The prototype generated the output file shown below, for this problem. The file contains the results of precondition calculations as well as all verification conditions to be proven in order to show consistency with the given specification. Please note:

- Reference is made to “UNITS”, which consist of a programming statement in the main program text, together with its postcondition.
- Formulae labelled with the words “Statement precondition” show the result of a precondition calculation. All other formulae are verification conditions to be proven.
- No simplifications of formulae have been performed.

UNIT at line 71

#####

Statement precondition:

```

((FORALL row : (0 < row) AND (row <= currow) : (EXISTS col : (0 <
col) AND (col < MAXCOL) : nonzero (matrix, MAXCOL, row, col,
col))) AND ((lastrow = MAXROW) COR allzero (matrix, MAXCOL, currow + 1)) AND
(lastrow = currow) AND ((currow <> MAXROW) = (currow <> MAXROW)))

```

UNIT at line 35

#####

```

(1) ~ (currow <> lastrow) AND (inbounds (MAXROW, currow, lastrow) AND (FORALL
row : (0 < row) AND (row <= currow) : (EXISTS col : (0 < col) AND (
col < MAXCOL) : nonzero (matrix, MAXCOL, row, col, col))) AND ((

```

## APPENDIX E. AN ANNOTATED PROGRAM AND ITS VCS

185

```
lastrow = MAXROW) COR allzero (matrix, MAXCOL, currow + 1)))
```

```
=>
```

```
((FORALL row : (0 < row) AND (row <= currow) : (EXISTS col : (0 <
col) AND (col < MAXCOL) : nonzero (matrix, MAXCOL, row, col,
col))) AND ((lastrow = MAXROW) COR allzero (matrix, MAXCOL, currow + 1)) AND
(lastrow = currow) AND ((currow <> MAXROW) = (currow <> MAXROW)))
```

```
(2) (currow <> lastrow) AND (inbounds (MAXROW, currow, lastrow) AND (FORALL
row : (0 < row) AND (row <= currow) : (EXISTS col : (0 < col) AND (
col < MAXCOL) : nonzero (matrix, MAXCOL, row, col, col))) AND ((
lastrow = MAXROW) COR allzero (matrix, MAXCOL, currow + 1)))
```

```
=>
```

```
((lastrow - currow) > 0)
```

```
(3) (1) (inbounds (MAXROW, currow, lastrow) AND (FORALL row : (0 <
row) AND (row <= currow) : (EXISTS col : (0 < col) AND (col <
MAXCOL) : nonzero (matrix, MAXCOL, row, col, col))) AND (lastrow =
MAXROW) AND (currow <> lastrow) AND (t1 = T1) AND (curcol = lastcol) AND
nonzero (matrix, MAXCOL, currow + 1, curcol, lastcol))
```

```
=>
```

```
((curcol = MAXCOL) OR (curcol <> MAXCOL))
```

```
(2) (inbounds (MAXROW, currow, lastrow) AND (FORALL row : (0 <
row) AND (row <= currow) : (EXISTS col : (0 < col) AND (col <
MAXCOL) : nonzero (matrix, MAXCOL, row, col, col))) AND (lastrow =
MAXROW) AND (currow <> lastrow) AND (t1 = T1) AND (curcol = lastcol) AND
nonzero (matrix, MAXCOL, currow + 1, curcol, lastcol)) AND (curcol =
MAXCOL)
```

```
=>
```

```
((currow - currow) < T1) AND (inbounds (MAXROW, currow, currow) AND (FORALL
row : (0 < row) AND (row <= currow) : (EXISTS col : (0 < col) AND (
```

```
col < MAXCOL) : nonzero (matrix, MAXCOL, row, col, col))) AND ((
currow = MAXROW) COR allzero (matrix, MAXCOL, currow + 1)))
```

```
(inbounds (MAXROW, currow, lastrow) AND (FORALL row : (0 <
row) AND (row <= currow) : (EXISTS col : (0 < col) AND (col <
MAXCOL) : nonzero (matrix, MAXCOL, row, col, col))) AND (lastrow =
MAXROW) AND (currow <> lastrow) AND (t1 = T1) AND (curcol = lastcol) AND
nonzero (matrix, MAXCOL, currow + 1, curcol, lastcol)) AND (curcol <>
MAXCOL)
```

=>

```
((lastrow - (currow + 1)) < T1) AND (inbounds (MAXROW, (currow + 1),
lastrow) AND (FORALL row : (0 < row) AND (row <= (currow + 1)) : (EXISTS
col : (0 < col) AND (col < MAXCOL) : nonzero (matrix, MAXCOL,
row, col, col))) AND ((lastrow = MAXROW) COR allzero (matrix,
MAXCOL, (currow + 1) + 1)))
```

```
(1) ~ (curcol <> lastcol) AND (inbounds (MAXROW, currow, lastrow) AND (FORALL
row : (0 < row) AND (row <= currow) : (EXISTS col : (0 < col) AND (
col < MAXCOL) : nonzero (matrix, MAXCOL, row, col, col))) AND (
lastrow = MAXROW) AND (currow <> lastrow) AND (t1 = T1) AND inbounds (
MAXCOL, curcol, lastcol) AND nonzero (matrix, MAXCOL, currow + 1,
curcol, lastcol))
```

=>

```
(inbounds (MAXROW, currow, lastrow) AND (FORALL row : (0 < row) AND (
row <= currow) : (EXISTS col : (0 < col) AND (col < MAXCOL) :
nonzero (matrix, MAXCOL, row, col, col))) AND (lastrow = MAXROW) AND (
currow <> lastrow) AND (t1 = T1) AND (curcol = lastcol) AND nonzero (
matrix, MAXCOL, currow + 1, curcol, lastcol))
```

```
(2) (curcol <> lastcol) AND (inbounds (MAXROW, currow, lastrow) AND (FORALL
row : (0 < row) AND (row <= currow) : (EXISTS col : (0 < col) AND (
```

## APPENDIX E. AN ANNOTATED PROGRAM AND ITS VCS

187

```

col < MAXCOL) : nonzero (matrix, MAXCOL, row, col, col))) AND (
lastrow = MAXROW) AND (currow <> lastrow) AND (t1 = T1) AND inbounds (
MAXCOL, curcol, lastcol) AND nonzero (matrix, MAXCOL, currow + 1,
curcol, lastcol))
=>
((lastcol - curcol) > 0)

```

```

(3) (inbounds (MAXROW, currow, lastrow) AND (FORALL row : (0 <
row) AND (row <= currow) : (EXISTS col : (0 < col) AND (col <
MAXCOL) : nonzero (matrix, MAXCOL, row, col, col))) AND (lastrow =
MAXROW) AND (currow <> lastrow) AND (t1 = T1) AND inbounds (MAXCOL,
curcol, lastcol) AND nonzero (matrix, MAXCOL, currow + 1, curcol,
lastcol)) AND ((lastcol - curcol) = T2) AND (curcol <> lastcol)
=>
((matrix [currow + 1] [curcol + 1] = 0) OR (matrix [currow + 1] [
curcol + 1] <> 0)) AND ((matrix [currow + 1] [curcol + 1] = 0) => ((
lastcol - (curcol + 1)) < T2) AND (inbounds (MAXROW, currow,
lastrow) AND (FORALL row : (0 < row) AND (row <= currow) : (EXISTS
col : (0 < col) AND (col < MAXCOL) : nonzero (matrix, MAXCOL,
row, col, col))) AND (lastrow = MAXROW) AND (currow <> lastrow) AND ((
lastrow - currow) = T1) AND inbounds (MAXCOL, (curcol + 1), lastcol) AND
nonzero (matrix, MAXCOL, currow + 1, (curcol + 1), lastcol))) AND ((
matrix [currow + 1] [curcol + 1] <> 0) => ((curcol - curcol) <
T2) AND (inbounds (MAXROW, currow, lastrow) AND (FORALL row : (0 <
row) AND (row <= currow) : (EXISTS col : (0 < col) AND (col <
MAXCOL) : nonzero (matrix, MAXCOL, row, col, col))) AND (lastrow =
MAXROW) AND (currow <> lastrow) AND ((lastrow - currow) = T1) AND
inbounds (MAXCOL, curcol, curcol) AND nonzero (matrix, MAXCOL,
currow + 1, curcol, curcol)))

```

```

(inbounds (MAXROW, currow, lastrow) AND (FORALL row : (0 < row) AND (

```

**APPENDIX E. AN ANNOTATED PROGRAM AND ITS VCS**

188

```

row <= currow) : (EXISTS col : (0 < col) AND (col < MAXCOL) :
nonzero (matrix, MAXCOL, row, col, col))) AND ((lastrow = MAXROW) COR
allzero (matrix, MAXCOL, currow + 1))) AND ((lastrow - currow) =
T1) AND (currow <> lastrow)
=>
(inbounds (MAXROW, currow, lastrow) AND (FORALL row : (0 < row) AND (
row <= currow) : (EXISTS col : (0 < col) AND (col < MAXCOL) :
nonzero (matrix, MAXCOL, row, col, col))) AND (lastrow = MAXROW) AND (
currow <> lastrow) AND ((lastrow - currow) = T1) AND inbounds (
MAXCOL, 0, MAXCOL) AND nonzero (matrix, MAXCOL, currow + 1, 0,
MAXCOL))

```

Statement precondition:

```

(inbounds (MAXROW, currow, lastrow) AND (FORALL row : (0 < row) AND (
row <= currow) : (EXISTS col : (0 < col) AND (col < MAXCOL) :
nonzero (matrix, MAXCOL, row, col, col))) AND ((lastrow = MAXROW) COR
allzero (matrix, MAXCOL, currow + 1)))

```

UNIT at line 33

#####

```

((MAXROW > 0) AND (MAXCOL > 0))

```

=>

```

(inbounds (MAXROW, 0, MAXROW) AND (FORALL row : (0 < row) AND (
row <= 0) : (EXISTS col : (0 < col) AND (col < MAXCOL) : nonzero (
matrix, MAXCOL, row, col, col))) AND ((MAXROW = MAXROW) COR allzero (
matrix, MAXCOL, 0 + 1)))

```

# Bibliography

- [1] *Contributions from VERkshop I — A Workshop on Formal Verification, Menlo Park, CA, USA*, April 1980.
- [2] *Contributions from VERkshop II — A Workshop on Formal Verification, Gaithersburg, USA*, April 1981.
- [3] *Proceedings of VERkshop III — A Formal Verification Workshop, Watsonville, CA, USA*, February 1985.
- [4] S. Agerholm. *Mechanizing Program Verification in HOL*. MSc Thesis, Aarhus University Denmark, 1992.
- [5] A. Ah-kee. Proof obligations for blocks and procedures. *Formal Aspects of Computing*, 2:312–330, 1990.
- [6] K. R. Apt. Ten years of Hoare’s logic: A survey—Part I. *ACM Transactions on Programming Languages and Systems*, 3(4):431–483, October 1981.
- [7] E.A. Ashcroft, M. Clint, and C.A.R. Hoare. Remarks on “Program proving: Jumps and functions by M. Clint and C.A.R. Hoare”. *Acta Informatica*, 6:317–318, 1976.
- [8] R.J.R. Back. A calculus of refinements for program derivations. *Acta Informatica*, 25:593–624, 1988.
- [9] R.J.R. Back. *Correctness Preserving Program Refinements: Proof Theory and Applications*. Mathematical Centre Tracts 131, Mathematisch Centrum, Amsterdam, 1980.
- [10] R.J.R. Back. On correct refinement of programs. *Journal of Computer and System Sciences*, 23:49–68, 1981.

- [11] R.J.R. Back. Proving total correctness of nondeterministic programs in infinitary logic. *Acta Informatica*, 15:233–249, 1981.
- [12] R.J.R. Back and J. von Wright. Duality in specification languages: A lattice-theoretical approach. *Acta Informatica*, 27:583–625, 1990.
- [13] R.J.R. Back and J. von Wright. Refinement concepts formalised in higher order logic. *Formal Aspects of Computing*, 2:247–272, 1990.
- [14] R.C. Backhouse. *Program Construction and Verification*. Prentice-Hall International, 1986.
- [15] H. Barringer, J.H.Cheng, and C.B. Jones. A logic covering undefinedness in program proofs. *Acta Informatica*, 21:251–269, 1984.
- [16] Z. Bavel. *Math Companion for Computer Science*. Reston Publishing Company, 1982.
- [17] A. Bijlsma, P.A. Matthews, and J.G. Wiltink. A sharp proof rule for procedures in wp semantics. *Acta Informatica*, 26:409–419, 1989.
- [18] A. Bijlsma, J.G. Wiltink, and P.A. Matthews. Equivalence of the Gries and Martin proof rules for procedure calls. *Acta Informatica*, 23:357–360, 1986.
- [19] G. Birkhoff. *Lattice Theory*. American Mathematical Society, Providence, R.I., third edition, 1967.
- [20] A.J. Blikle. The clean termination of iterative programs. *Acta Informatica*, 16:199–217, 1981.
- [21] A.J. Blikle. On the development of correct specified programs. *IEEE Transactions on Software Engineering*, SE-7(5):519–527, September 1981.
- [22] A.J. Blikle. Three-valued predicates for software specification and validation. In *Proceedings of the VDM-Europe Symposium 1988, LNCS 328*, pages 243–266, Springer-Verlag, 1988.
- [23] E.K. Blum, H. Ehrig, and F. Parisi-Presicce. Algebraic specification of modules and their basic interconnections. *Journal of Computer and System Sciences*, 34:293–339, 1987.

- [24] H.-J. Boehm. Side effects and aliasing can have simple axiomatic descriptions. *ACM Transactions on Programming Languages and Systems*, 7(4):637–655, October 1985.
- [25] H.J. Boom. A weaker precondition for loops. *ACM Transactions on Programming Languages and Systems*, 4(4):668–677, October 1982.
- [26] R.S. Boyer and J.S. Moore. *A Computational Logic*. Academic Press, 1979.
- [27] J. Bradley. *Introduction to Discrete Mathematics*. Addison-Wesley Publishing Company, 1988.
- [28] R. Cartwright and D. Oppen. The logic of aliasing. *Acta Informatica*, 15:365–384, 1981.
- [29] W. Chen and J.T. Udding. Towards a calculus of data refinement. In J.L.A. van de Snepsheut, editor, *Mathematics of Program Construction, LNCS 375*, Springer-Verlag, 1989.
- [30] M. Clint and C.A.R. Hoare. Program proving: Jumps and functions. *Acta Informatica*, 1:214–224, 1972.
- [31] A. Cohn. Correctness properties of the Viper block model: The second level. In G. Birtwistle and P.A. Subrahmanyam, editors, *Current Trends in Hardware Verification and Automated Theorem Proving*, pages 1–91, Springer-Verlag, 1989.
- [32] D. Coleman and J.W. Hughes. The clean termination of Pascal programs. *Acta Informatica*, 11:195–210, 1979.
- [33] R.L. Constable and M.J. O'Donnell. *A Programming Logic*. Winthrop Publishers, Inc., 1978.
- [34] W.P. de Roever. Dijkstra's predicate transformer, non-determinism, recursion and termination. In *Mathematical Foundations of Computer Science, LNCS 45*, pages 472–481, Springer-Verlag, 1976.
- [35] R. DeMillo, R. Lipton, and A. Perlis. Social processes and proofs of theorems and programs. *Communications of the ACM*, 22(5):271–280, May 1979.
- [36] N. Dershowitz. Program abstraction and instantiation. *ACM Transactions on Programming Languages and Systems*, 7(3):446–477, July 1985.

- [37] E.W. Dijkstra. A constructive approach to the problem of program correctness. *BIT*, 8:174–186, 1968.
- [38] E.W. Dijkstra. *A Discipline of Programming*. Prentice-Hall International, 1976.
- [39] E.W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18(8):453–457, August 1975.
- [40] E.W. Dijkstra and W.H.J. Feijen. *A Method of Programming*. Addison-Wesley Publishing Company, 1988.
- [41] E.W. Dijkstra and C.S. Scholten. *Predicate Calculus and Program Semantics*. Springer-Verlag, 1990.
- [42] E.W. Dijkstra and A.J.M. van Gasteren. A simple fixpoint argument without the restriction to continuity. *Acta Informatica*, 23:1–7, 1986.
- [43] G. Dromey. *Program Derivation: The Development of Programs from Specifications*. Addison-Wesley Publishing Company, 1989.
- [44] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification 1: Equations and Initial Semantics*. Springer-Verlag, 1985.
- [45] G.W. Ernst. Rules of inference for procedure calls. *Acta Informatica*, 8:145–152, 1977.
- [46] F.L. Bauer et al. *The Munich Project CIP. Volume I: The Wide Spectrum Language CIP-L*. Volume 183 of *Lecture Notes in Computer Science*, Springer-Verlag, 1985.
- [47] R.L. London et al. Proof rules for the programming language Euclid. *Acta Informatica*, 10:1–26, 1978.
- [48] M.S. Feather. A survey and classification of some program transformation approaches and techniques. In L.G.L.T. Meertens, editor, *Program Specification and Transformation*, pages 165–198, North-Holland, 1987.
- [49] J.H. Fetzer. Program verification: The very idea. *Communications of the ACM*, 31(9):1048–1063, September 1988.

- [50] S. Fickas. Automating the transformational development of software. *IEEE Transactions on Software Engineering*, SE-11(11):1268-1277, November 1985.
- [51] R.W. Floyd. Assigning meanings to programs. *Proceedings of the American Mathematical Society Symposia in Applied Mathematics*, 19:19-32, 1967.
- [52] N. Gehani and A.D. McGettrick, editors. *Software Specification Techniques*. Addison-Wesley Publishing Company, 1986.
- [53] J.V. Giordano. Some verification problems in Pascal-like languages. *ACM SIGSOFT Software Engineering Notes*, 5(1):18-27, January 1980.
- [54] J.A. Goguen. More thoughts on specification and verification. In N. Gehani and A.D. McGettrick, editors, *Software Specification Techniques*, pages 47-52, Addison-Wesley Publishing Company, 1986.
- [55] J.A. Goguen. Thoughts on program specification, design and verification. *ACM SIGSOFT Software Engineering Notes*, 5(3), 1980.
- [56] J.A. Goguen, J.W. Thatcher, and E.G. Wagner. Initial algebra approach to the specification, correctness, and implementation of abstract data types. In R.T. Yeh, editor, *Current Trends in Programming Methodology, Vol. IV, Data Structuring*, Prentice-Hall, 1978.
- [57] D.I. Good. Mechanical proofs about computer programs. In C.A.R. Hoare and J.C. Shepherdson, editors, *Mathematical Logic and Programming Languages*, pages 55-75, Prentice-Hall International, 1985.
- [58] D.I. Good, R.L. London, and W.W. Bledsoe. An interactive program verification system. *IEEE Transactions on Software Engineering*, SE-1(1):59-67, March 1975.
- [59] M.J. Gordon, R. Milner, and C.P. Wadsworth. *Edinburgh LCF*. Volume 78 of *Lecture Notes in Computer Science*, Springer-Verlag, 1979.
- [60] M.J.C. Gordon. HOL: A proof generating system for higher order logic. In G. Birtwistle and P.A. Subrahmanyam, editors, *VLSI Specification, Verification and Synthesis*, Kluwer Academic Publishers, 1988.

- [61] M.J.C. Gordon. *Mechanizing Programming Logics in Higher Order Logic*. Technical Report 145, University of Cambridge, September 1988.
- [62] G. Grätzer. *General Lattice Theory*. Birkhäuser Verlag, 1978.
- [63] D. Gray. A pedagogical verification condition generator. *The Computer Journal*, 30(3):239–248, 1987.
- [64] D. Gries. Coordinate transformation and data refinement. In *International Summer School on Program Design Calculi, Marktoberdorf, Germany*, July 1992.
- [65] D. Gries. The multiple assignment statement. *IEEE Transactions on Software Engineering*, SE-4(2):89–93, March 1978.
- [66] D. Gries. *The Science of Programming*. Springer-Verlag, 1981.
- [67] D. Gries and G. Levin. Assignment and procedure call proof rules. *ACM Transactions on Programming Languages and Systems*, 2(4):564–579, October 1980.
- [68] D. Gries and J. Prins. A new notion of encapsulation. In *Proceedings of SIGPLAN Symposium on Language Issues in Programming Environments*, pages 131–139, 1985.
- [69] Stanford Verification Group. *Stanford Pascal Verifier User Manual*. Technical Report CSD Report No. STAN-CS-79-731, Stanford University, 1979.
- [70] D. Guaspari, C. Marceau, and W. Polak. Formal verification of Ada programs. *IEEE Transactions on Software Engineering*, 16(9):1058–1075, September 1990.
- [71] P. Guerreiro. Another characterization of weakest preconditions. In *LNCS 137*, pages 164–177, Springer-Verlag, 1982.
- [72] R.D. Gumb. On the underlying logics of specification languages. *ACM SIGSOFT Software Engineering Notes*, 7(4):21–23, October 1982.
- [73] J.V. Guttag. Notes on type abstraction. In N. Gehani and A.D. McGettrick, editors, *Software Specification Techniques*, pages 55–73, Addison-Wesley Publishing Company, 1986.

- [74] J.V. Guttag and J.J. Horning. Report on the Larch Shared Language. *Science of Computer Programming*, 6:103–134, 1986.
- [75] J.V. Guttag, J.J. Horning, K.D. Jones, S.J. Garland, A. Modet, and J.M. Wing. *Larch: Languages and Tools for Formal Specification. Texts and Monographs in Computer Science*, Springer-Verlag, 1993.
- [76] J.V. Guttag, J.J. Horning, and J.M. Wing. *Larch in Five Easy Pieces*. Technical Report, Digital Equipment Corporation, Systems Research Center, Palo Alto, California, November 1985.
- [77] R. Harper. *Introduction to Standard ML*. Technical Report ECS-LFCS-86-14, Laboratory for Foundations of Computer Science, Department of Computer Science, University of Edinburgh, November 1986.
- [78] E.C.R. Hehner. do considered od: A contribution to the programming calculus. *Acta Informatica*, 11:287–304, 1979.
- [79] E.C.R. Hehner. *The Logic of Programming*. Prentice-Hall International, 1984.
- [80] M. Heisel. Formalizing and implementing Gries' program development method in dynamic logic. *Science of Computer Programming*, 18(1):107–137, January 1992.
- [81] C. Hemerik. *Formal Definitions of Programming Languages as a Basis for Compiler Construction*. PhD thesis, Technische Hogeschool Eindhoven, 1984.
- [82] W.H. Hesselink. Interpretations of recursion under unbounded nondeterminacy. *Theoretical Computer Science*, 59:211–234, 1988.
- [83] W.H. Hesselink. Predicate-transformer semantics of general recursion. *Acta Informatica*, 26:309–332, 1989.
- [84] C.A.R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 583, October 1969.
- [85] C.A.R. Hoare. Procedures and parameters: An axiomatic approach. In *Symposium on Semantics of Algorithmic Languages, Lecture Notes in Mathematics 188*, pages 102–116, Springer-Verlag, 1971.

- [86] C.A.R. Hoare. Proof of correctness of data representations. *Acta Informatica*, 1:271–281, 1972.
- [87] C.A.R. Hoare. Some properties of predicate transformers. *Journal of the ACM*, 25(3):461–480, 1978.
- [88] C.A.R. Hoare and J. He. The weakest prespecification. *Information Processing Letters*, 24:127–132, January 1987.
- [89] C.A.R. Hoare, J. He, and J.W. Sanders. Prespecification in data refinement. *Information Processing Letters*, 25(2):71–76, May 1987.
- [90] A. Hoogewijs. Partial-predicate logic in computer science. *Acta Informatica*, 24:381–393, 1987.
- [91] T. Hrycej. A knowledge-based problem-specific program generator. *ACM SIGPLAN Notices*, 22:53–61, February 1987.
- [92] S. Igarashi, R.L. London, and D.C. Luckham. Automatic program verification I: A logical basis and its implementation. *Acta Informatica*, 4:145–182, 1975.
- [93] D. Jacobs and D. Gries. General correctness: A unification of partial and total correctness. *Acta Informatica*, 22:67–83, 1985.
- [94] C.B. Jones. Constructing a theory of a data structure as an aid to program development. *Acta Informatica*, 11:119–137, 1979.
- [95] C.B. Jones. *Software Development: A Rigorous Approach*. Prentice-Hall International, 1980.
- [96] C.B. Jones. *Systematic Software Development Using VDM*. Prentice-Hall International, 1986.
- [97] C.B. Jones. VDM proof obligations and their justification. In *VDM—A Formal Method at Work, Proceedings of the VDM-Europe Symposium 1987, LNCS 252*, pages 260–286, Springer-Verlag, 1987.

- [98] C.B. Jones and P.A. Lindsay. A support system for formal reasoning: Requirements and status. In R. Bloomfield, L. Marshall, and R. Jones, editors, *VDM—The Way Ahead, Proceedings of the Second VDM-Europe Symposium 1988, LNCS 328*, pages 139–152, Springer-Verlag, 1988.
- [99] C.B. Jones and R. Moore. Muffin: A user interface design experiment for a theorem proving assistant. In R. Bloomfield, L. Marshall, and R. Jones, editors, *VDM—The Way Ahead, Proceedings of the Second VDM-Europe Symposium 1988, LNCS 328*, pages 335–375, Springer-Verlag, 1988.
- [100] M.B. Josephs. The data refinement calculator for Z specifications. *Information Processing Letters*, 27(1):29–33, February 1988.
- [101] C.R. Karp. *Languages with Expressions of Infinite Length*. North-Holland Publishing Company, 1964.
- [102] P. Kilpatrick and P. McParland. Software support for the refinement of VDM specifications. In R. Bloomfield, L. Marshall, and R. Jones, editors, *VDM—The Way Ahead, Proceedings of the Second VDM-Europe Symposium 1988, LNCS 328*, pages 459–475, Springer-Verlag, 1988.
- [103] S.C. Kleene. *Introduction to Metamathematics*. D. Van Nostrand Company, 1952.
- [104] B. Konikowska, A. Tarlecki, and A.J. Blikle. A three-valued logic for software specification and validation. In *Proceedings of the VDM-Europe Symposium 1988, LNCS 328*, pages 218–242, Springer-Verlag, 1988.
- [105] B. Krieg-Brückner. Integration of program construction and verification: The PROSPECTRA methodology. In A.N. Habermann and U. Montanari, editors, *Innovative Software Factories and Ada, LNCS 275*, pages 1–22, Springer-Verlag, 1987.
- [106] J.-L. Lassez, V.L. Nguyen, and E.A. Sonenberg. Fixed point theorems and semantics: A folk tale. *Information Processing Letters*, 14(3):112–116, May 1982.
- [107] P.A. Lindsay. A survey of mechanical support for formal reasoning. *Software Engineering Journal*, 3(1):3–27, January 1988.

- [108] B.H. Liskov and V. Berzins. An appraisal of program specifications. In N. Gehani and A.D. McGettrick, editors, *Software Specification Techniques*, pages 3-23, Addison-Wesley Publishing Company, 1986.
- [109] D.W. Loveland. Automated theorem-proving: A quarter-century review. In W.W. Bledsoe and D.W. Loveland, editors, *Automated Theorem Proving: After 25 Years, Contemporary Mathematics, Vol. 29*, pages 1-45, American Mathematical Society, 1984.
- [110] T.F. Lunney and R.H. Perrott. Syntax-directed editing. *Software Engineering Journal*, 3(2):37-46, March 1988.
- [111] E. Lutz. Some proofs of data refinement. *Information Processing Letters*, 34(4):179-185, April 1990.
- [112] Z. Manna. *Mathematical Theory of Computing*. McGraw-Hill Book Company, 1974.
- [113] Z. Manna and A. Pnueli. Axiomatic approach to total correctness of programs. *Acta Informatica*, 3:243-263, 1974.
- [114] A.J. Martin. A general proof rule for procedures in predicate transformer semantics. *Acta Informatica*, 20:301-313, 1983.
- [115] J. McCarthy. A basis for a mathematical theory of computation. In P. Braffort and D. Hirschberg, editors, *Computer Programming and Formal Systems*, pages 33-70, North-Holland, 1967.
- [116] E. Mendelson. *Introduction to Mathematical Logic*. D. Van Nostrand Company, second edition, 1979.
- [117] R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348-375, 1978.
- [118] R. Milner. The use of machines to assist in rigorous proof. In C.A.R. Hoare and J.C. Shepherdson, editors, *Mathematical Logic and Programming Languages*, pages 77-87, Prentice-Hall International, 1985.
- [119] C.C. Morgan. Data refinement by miracles. *Information Processing Letters*, 26:243-246, January 1988.

- [120] C.C. Morgan. Procedures, parameters, and abstraction: separate concerns. *Science of Computer Programming*, 11:17–27, 1988.
- [121] C.C. Morgan. *Programming from Specifications*. Prentice Hall International, 1990.
- [122] C.C. Morgan. The specification statement. *ACM Transactions on Programming Languages and Systems*, 10(3):403–419, July 1988.
- [123] C.C. Morgan and P.H.B. Gardiner. Data refinement by calculation. *Acta Informatica*, 27:481–503, 1990.
- [124] C.C. Morgan and K. Robinson. Specification statements and refinement. *IBM Journal of Research and Development*, 31(5):546–555, September 1987.
- [125] C.C. Morgan and T. Vickers. Types and invariants in the refinement calculus. *Science of Computer Programming*, 14:281–304, 1990.
- [126] J.M. Morris. Assignment and linked data structures. In M. Broy and G. Schmidt, editors, *Theoretical Foundations of Programming Methodology*, pages 35–42, D. Reidel Publishing Company, 1982.
- [127] J.M. Morris. A general axiom of assignment. In M. Broy and G. Schmidt, editors, *Theoretical Foundations of Programming Methodology*, pages 25–34, D. Reidel Publishing Company, 1982.
- [128] J.M. Morris. Laws of data refinement. *Acta Informatica*, 26:287–308, 1989.
- [129] J.M. Morris. A proof of the Schorr-Waite algorithm. In M. Broy and G. Schmidt, editors, *Theoretical Foundations of Programming Methodology*, pages 43–51, D. Reidel Publishing Company, 1982.
- [130] J.M. Morris. A theoretical basis for stepwise refinement and the programming calculus. *Science of Computer Programming*, 9:287–306, 1987.
- [131] J.M. Morris. Varieties of weakest liberal preconditions. *Information Processing Letters*, 25:207–210, May 1987.
- [132] J.M. Morris. Well-founded induction and the invariance theorem for loops. *Information Processing Letters*, 32:155–158, August 1989.

- [133] G. Nelson. *A Generalization of Dijkstra's Calculus*. Technical Report 16, Digital Systems Research Center, Palo Alto, California, USA, April 1987.
- [134] G. Nelson and D.C. Oppen. Simplification by cooperating decision procedures. *ACM Transactions on Programming Languages and Systems*, 1(2):245–257, October 1979.
- [135] M.J. O'Donnell. A critique of the foundations of Hoare style programming logics. *Communications of the ACM*, 25(12):927–935, December 1982.
- [136] E.-R. Olderog. On the notion of expressiveness and the rule of adaptation. *Theoretical Computer Science*, 24:337–347, 1983.
- [137] R.A. Overbeek and E.L. Lusk. Data structures and control architecture for implementation of theorem-proving programs. In R. Kowalski and W. Bibel, editors, *Proceedings of the Fifth Conference on Automated Deduction, LNCS 87*, pages 232–249, Springer-Verlag, Berlin-Heidelberg-New York, 1980.
- [138] H. Partsch and R. Steinbrüggen. Program transformation systems. *ACM Computing Surveys*, 15(3):199–236, 1983.
- [139] D.E. Perry. The Inscape Environment. In *Proceedings of the Eleventh International Conference on Software Engineering*, May 1989.
- [140] G. Plotkin. Dijkstra's predicate transformers and Smyth's power domains. In *LNCS 86*, pages 527–553, Springer-Verlag, 1979.
- [141] G.D. Plotkin. A powerdomain construction. *SIAM Journal on Computation*, 5(3):452–487, September 1976.
- [142] A.M. de A. Price. Defining dynamic variables and abstract data types in Pascal. *ACM SIGPLAN Notices*, 19(2):85–91, February 1984.
- [143] N. Ramsey. Developing formally verified Ada programs. *ACM SIGSOFT Software Engineering Notes*, 14(3):257–265, May 1989.
- [144] T. Reps and T. Teitelbaum. The Synthesizer Generator. *ACM SIGSOFT Software Engineering Notes*, 9(3):42–48, May 1984.

- [145] T. Reps and T. Teitelbaum. *The Synthesizer Generator Reference Manual*. Department of Computer Science, Cornell University, 1987.
- [146] C. Reynolds and R.T. Yeh. Induction as the basis for program verification. *IEEE Transactions on Software Engineering*, SE-2(4):244–252, December 1976.
- [147] J.C. Reynolds. *The Craft of Programming*. Prentice-Hall International, 1981.
- [148] D.R. Smith. KIDS: A semi-automatic program development system. *IEEE Transactions on Software Engineering*, 16(9):1024–1043, September 1990.
- [149] M.B. Smyth. Powerdomains. *Journal of Computer and System Sciences*, 16(1):23–36, 1978.
- [150] J.M. Spivey. *The Z Notation: A Reference Manual*. Prentice-Hall International, second edition, 1992.
- [151] R. St-Denis and P.N. Robillard. An approach to knowledge-driven system software development. In *Fourth International Workshop on Software, Specification, and Design*, pages 95–99, 1987.
- [152] R.D. Tennent. A note on undefined expression values in programming logics. *Information Processing Letters*, 24:331–333, March 1987.
- [153] H.C. Thatcher. On the elimination of pointer variables and dynamic allocation in higher level languages. *ACM SIGPLAN Notices*, 19(4):44–46, April 1984.
- [154] J.P. Tremblay and R. Manohar. *Discrete Mathematical Structures with Applications to Computer Science*. McGraw-Hill Book Company, 1975.
- [155] M. Wand. A characterization of weakest preconditions. *Journal of Computer and System Sciences*, 15:209–212, 1977.
- [156] N. Wirth. Program development by stepwise refinement. *Communications of the ACM*, 14(4):221–227, April 1971.
- [157] J.B. Wordsworth. *Software Development with Z*. Addison-Wesley, 1992.

## **BIBLIOGRAPHY**

**202**

- [158] D.B. Wortman. On legality assertions in Euclid. *IEEE Transactions on Software Engineering*, SE-5(4):359–367, July 1979.