Toward Reliable Modular Programs

Thesis by

K. Rustan M. Leino

In Partial Fulfillment of the Requirements
for the Degree of
Doctor of Philosophy

California Institute of Technology
Pasadena, California

1995
(Submitted 5 January 1995)

Abstract

Software is being applied in an ever-increasing number of areas. Computer programs and systems are becoming more complex and consisting of more delicately interconnected components. Errors surfacing in programs are still a conspicuous and costly problem. It's about time we employ some techniques that guide us toward higher reliability of practical programs. The goal of this thesis is just that.

This thesis presents a theory for verifying programs based on Dijkstra's weakest-precondition calculus. A variety of program paradigms used in practice, such as exceptions, procedures, object orientation, and modularity, are dealt with.

The thesis sheds new light on the theory behind programs with exceptions. It develops an elegant algebra, and shows it to be the foundation on which the semantics of exceptions rests. It develops a trace semantics for programs with exceptions, from which the weakest-precondition semantics is derived. It also proves a theorem on programming methodology relating to exceptions, and applies this theorem in the novel derivation of a simple program.

The thesis presents a simple model for object-oriented data types, in which concerns have been separated, resulting in the simplicity of the model.

To deal with large programs, this thesis takes a practical look at modularity and abstraction. It reveals a problem that arises in writing specifications for modular programs where previous techniques fail. The thesis introduces a new specification construct that solves that problem, and gives a formal proof of soundness for modular verification using that construct. The model is a generalization of Hoare's classical data refinement. However, there are more problems to be solved. The thesis reports on some of these problems and suggests some future directions toward more reliable modular programs.

Preface

I'm Rustan Leino. I'll be your host through this thesis. During the last three and a half years, I've been advised, in different capacities, by three people. Jan van de Snepscheut led me through my Master's thesis [49]. His taste and skill for elegant solutions to practical problems put me on the path that has led to the present thesis. I hope we will be able to keep up his enthusiasm. Mani Chandy with his slogan of "Bringing theory to the marketplace" guided me from there. His apt ability to authoritatively sieve the important from the unimportant, resulting in a focus on the road and not on the curbside distractions, was instrumental in making the journey expeditious. At Digital's Systems Research Center (DEC SRC), Greg Nelson, a champ in applying his (and others') theory in practice, directed me to problems yearning for my solutions. His ample support and fruitful collaboration have had a great impact on the destination of this thesis. I'm indebted to each of these individuals for his inspiration and guidance.

As a Microsoftee —before becoming a graduate student and starting a research career—, I learned the craft of programming in the large, and became more aware of the challenges involved in producing correct software and the discipline that requires. I also learned the importance of the run-time check as a device for more quickly detecting errors in a program. This thesis applies to verification in general, but has been motivated by the hope of proving statically that no run-time check will fail during run-time, an area known as *extended static checking*.

In addition to the people mentioned above, numerous other people at Caltech and DEC SRC have been of great help during discussions and in reviewing my thesis. Of these colleagues, I mention Dave Detlefs, Robert Harley, Allan Heydon, Peter Hofstee, Rajit Manohar, Berna Massingill, Adam Rifkin, Paul Sivilotti, John Thornley, and, of course, the members of my thesis committee: Mani Chandy, Alain Martin, Greg Nelson, Beverly Sanders, and Rick Wilson. I am grateful to them all for their stimulus and loyal support. I am particularly grateful to Rajit and Paul for their various contributions to proofs, Berna for her careful proofreading of my thesis, and Dave for his putting these formulas to work in the SRC Extended Static Checker. I'd also like to express much appreciation to DEC SRC for their financial support.

These pages have been prepared in LaTeX, using many useful macros by Marcel van der Goot and Rajit Manohar. The fancy font used in various headers is a public domain PostScript font called Civitype, developed by S.G. Moye. All figures were made from PostScript programs I wrote. I have written the thesis in a personal

style so that I may get to know my readers better.

Last, but not least, I want to thank Indi for her never-ending loving support. You've been of tremendous help in every way during this time.

TJOFLÖJT,

K.R.M.L.
January 1995
Pasadena, CA, U.S.A.

This thesis is available as Technical Report Caltech-CS-TR-95-03.

Contents

0 Introduction **0**

0.0 Motivation . . . 0
0.1 Contents . . . 2
0.2 Preliminaries . . . 4

I Control Structures **10**

Imperative programming languages . . . 12
Outline . . . 12

1 Semantics of programs with exceptions **14**

1.0 Weakest precondition . . . 14
1.1 Assignment . . . 16
1.2 Unit statements and compositions . . . 16
1.3 Block . . . 17
1.4 Partial commands . . . 18
1.5 Choice compositions . . . 20
1.6 Iteration . . . 21
1.7 Specification statement . . . 22
1.8 Refinement . . . 24

2 Functions of two arguments and their compositions **26**

2.0 Function compositions . . . 26
2.1 Left and right composition . . . 27
2.2 Double composition . . . 28
2.3 Ceiling and floor . . . 30
2.4 Transposition . . . 32
2.5 The connection with programs . . . 33
2.6 Concluding remarks . . . 34

3 Trace semantics for exceptions **36**

3.0 Introduction . . . 36
3.1 Trace sets . . . 37
3.2 Program constructs as trace sets . . . 38

3.3 Weakest preconditions of trace sets 43
3.4 Calculating the weakest preconditions 45

4 A theorem on programming methodology **48**
4.0 Hoare triples 48
4.1 Free occurrences of *raise* 49
4.2 Usage of exceptions 51

5 Constructing a program with exceptions **52**
5.0 A program derivation 52
5.1 Discussion 53

6 Modeling common programming languages **56**
6.0 Procedures 56
6.1 Alternative statements 59
6.2 Statements that "go wrong" 60
6.3 Expressions 61

II Data Structures **64**
Data structures 66
Outline 66

7 Data types **68**
7.0 Types 68
7.1 Types in common programming languages 70
7.2 Declaring new types 73
7.3 Maps and specifications 75

8 Objects **78**
8.0 Subtypes 78
8.1 Data fields 79
8.2 Methods 79
8.3 Method implementations 79
8.4 Object simplicity 80
8.5 Language implementations of objects 81
8.6 Objects in common programming languages 81

9 Abstraction **84**
9.0 Abstract variables 84
9.1 Abstract variables and refinement 86
9.2 Abstract data fields 89

III Modularity **92**
Modules and modular verification . . . 94
Outline . . . 94

10 Specifications in modular programs **96**
10.0 Motivation . . . 96
10.1 Problem . . . 97
10.2 Solution . . . 101
10.3 Enforcing the requirements . . . 104
10.4 Soundness of modular verification . . . 105
10.5 A generalization of classical data refinement . . . 106
10.6 Other specification languages . . . 108

11 Generating verification conditions **110**
11.0 A notation for modular programs . . . 110
11.1 Definitions . . . 112
11.2 Proving refinements . . . 122
11.3 The importance of residues . . . 124

12 Soundness of modular verification **128**
12.0 Requirements . . . 128
12.1 Soundness . . . 128
12.2 Proof outline . . . 129
12.3 All side effects are benevolent . . . 132
12.4 X . . . 135
12.5 Epilogue . . . 141

13 depends in perspective **144**
13.0 Specification of a consumer . . . 144
13.1 Shortcomings of **depends** . . . 146
13.2 Private values . . . 148

IV Epilogue **152**

14 Summary **154**

Chapter 0

Introduction

In this chapter, I present an introduction to, and the motivation for, this thesis. I also provide an outline of the contents of the thesis, show the dependencies between the chapters, and discuss some preliminaries such as notation.

0.0 Motivation

Today, computer programs are being written for an ever-growing number of purposes. Unfortunately, not all of this software is correct. Programmers introduce errors into programs for a variety of reasons. The errors may be the result of, *e.g.*, typos, logical mistakes, incorrect assumptions, vague or changing specifications, or lack of specifications. Whatever the cause, the effect of software errors can be very costly, *e.g.*, a malfunctioning computerized radiation therapy machine has claimed the lives of humans [52], a broken telephone switch has resulted in loss of service [44], problems with an automated baggage-handling system have delayed the opening of an airport [27], an erroneous word processor can cause the loss of important information, and an error-prone program can degrade the reputation of a software company. Even the errors that are found prior to the shipping of or use of a software product are costly, primarily in terms of man-hours spent finding and correcting errors, and in terms of delayed time to market, resulting in loss of market share and revenues.

To reduce the number of errors in a program, or to increase one's confidence in a program, one can *test* the program on a given test suite. If the program is observed to behave correctly for these test cases, the program is shipped to the customer. One then hopes there will be other cases that customers try for which the program also behaves correctly.

Another way to reduce the number of errors in and increase one's confidence in a program is to write precise specifications and mathematically *verify* that the program meets those specifications.

> **Remark 0.0.** Notice that I said "*reduce* the number of errors" and "*increase* one's confidence", as opposed to "*eliminate* all errors", because the specifications, too, can be written incorrectly, or the program could

be used incorrectly by the customer. However, specifications are generally concerned with fewer details than programs, and may thus be easier to get right. Moreover, the probability of making the same error in both specification and program is much lower than just making an error in the program text itself.

Mathematically proving a program correct shows the program will work not just for one test suite, but for all permitted uses of the program. This technique also has the advantage over testing in that it can be applied prior to the completion of an implementation; thus, errors can be found earlier, which reduces costs.

To sustain mathematical proofs of program correctness, programs must be given a mathematical meaning, a *semantics*. An example thereof is Dijkstra's *weakest-precondition* semantics [17], which has achieved considerable success in modeling programs mathematically, because of its high level of expressivity (using predicates over the state space) and its simplicity.

Weakest preconditions have been around for almost two decades, and other techniques for reasoning about programs (*e.g.*, Hoare triples [36]) for over two decades. So why is it that not every programmer uses these techniques every day? Part of the answer is that programmers actually do, implicitly—the study of semantics has had a profound influence on the design of programming languages that programmers use, and programmers may have firmed their understanding of programs through learning about semantics as undergraduates.

A major reason these techniques have not penetrated the everyday life of programmers more visibly is the challenges that the proving of large programs poses. For example, a large program gives rise to large formulas to be proven. Proving all of these by hand would be a virtually impossible task, especially for large programs that undergo change—one small change in the program may necessitate reproving the entire program. Instead, we may consider receiving assistance from automatic theorem provers (see, *e.g.*, [69, 74]). This is an area that still needs more work.

Another task that presents challenges is the mathematical modeling of programs. Although their design is influenced by semantics, common programming languages provide some features whose mathematical meanings are difficult to capture concisely, *e.g.*, arbitrary pointers.

Yet another area that presents challenges is the writing of precise specifications! If we don't know how to write specifications, how can we expect to be able to verify that a program behaves correctly?

A step toward the (full) verification of programs is an area called *extended static checking*. The idea is to prove programs correct, but only with respect to certain properties. In particular, a program is given enough specifications to prove the absence of *checked run-time errors*, like **nil**-dereferencing, array index out-of-bounds errors, and failing assertions. This is likely to lighten the burden in the areas of (automatic) theorem proving and specification writing. For example, rather than needing a means to express and reason about "permutation of" in the full specification and verification of a sorting routine, the specification would only need to be strong enough to show that the implementation never indexes the given array outside its bounds. Despite the fact that the verification of the latter does not guarantee the array to be sorted upon termination of the sorting routine, such a verification removes the possibility of a program execution resulting in a checked run-time error and greatly increases our confidence in the program.

> **Remark 0.1.** Drawing from my personal experience with developing programs at Microsoft, and also since then (*e.g.*, the implementation in [49]), I have found that almost all errors ever detected were detected as results of failing run-time checks. Proving the absence of checked run-time errors would thus have shown the absence of most errors detected in these programs.

The technique also offers the benefit of detecting errors earlier. As a bonus, run-time checks can be removed from the executable code, making the executable smaller and faster [26] (see also [78]).

Extended static checking gets its name from the idea that it is to be incorporated into a compiler, much like static type checkers in today's compilers. This would indeed bring the direct benefit of the science of program correctness to programmers, an event one may expect to do marvels for the correctness of their programs.

0.1 Contents

This thesis concerns the specification and verification of *modular* programs that feature *exceptions* and *objects*. "*Modular*" means that programs are divided into pieces that can be compiled separately. Moreover, through the application of *modular verification*, the modules of such programs can be verified separately. This is important, because it is modularity and modular verification that allow us to tackle large programs. An *exception* is a form of a structured jump and is sometimes convenient in programming. A small example thereof is given in Chapter 5, but exceptions are more frequently used in large programs. *Objects* are a means of organizing data in a program and facilitate the sharing of code. Object-oriented programming is gaining popularity. Its utility is most visible when programming in the large.

The main contributions of this thesis are several aspects of reasoning about programs with exceptions, a separation of concerns in the meaning of objects, and the invention of a new specification construct for the specification and verification of modular programs. Of these three main contributions, the last mentioned, which in some sense is an extension of classical *data refinement* [38], is bound to have the greatest impact on making the specification and verification of large programs feasible in practice.

Although they work independently of each other, these three contributions, collectively, are steps toward making modular programs more reliable.

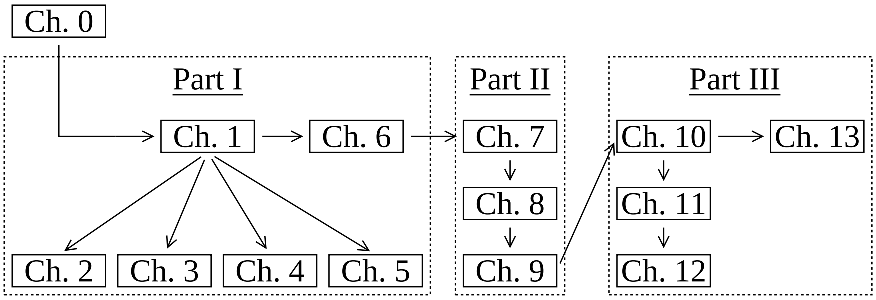

Figure 0.0: Roadmap to dependencies between chapters

0.1.0 Outline

Compiled from sources like [17, 70, 67, 51], I kick off Part I, Control Structures, by presenting a mathematical semantics, based on weakest preconditions, of sequential imperative programming constructs for languages with exceptions (Chapter 1). Then, for the duration of a few chapters, I devote my attention to the control flow of programs with exceptions (so-called *exceptional* programs, no pun intended). I show that the weakest preconditions of the basic exceptional statements and compositions have their foundation in a beautiful algebra over functions of two arguments (Chapter 2). I show an operational semantics of programs with exceptions and its connection with the weakest precondition semantics (Chapter 3). I also prove a theorem that suggests a use for exceptions (Chapter 4), and from it show a heuristic for program construction applied to the derivation of a simple program (Chapter 5). With an eye to "real" programming languages and their common usage, I present procedures, and show how run-time errors, partial expressions, short-circuit boolean operators, and expressions with side effects are modeled mathematically (Chapter 6).

In Part II, Data Structures, I turn my attention to the data structures of programs and show how common data structures are modeled mathematically (Chapter 7). The most interesting of these concerns objects (Chapter 8), for which I propose a simple mathematical treatment that separates the concerns involved. As a warm-up to Part III, I also present data abstraction in its classical form [38] (Chapter 9).

Part III, Modularity, concerns modular specification and verification of programs. I demonstrate our lack of understanding of writing specifications in modular programs (with an emphasis on object-oriented programs), and contribute a new specification construct, **depends**, as a necessary aid in writing such specifications (Chapter 10). I define a precise interpretation of modular programs and their specifications (Chapter 11), and prove that this interpretation lends itself to sound modular verification of programs (Chapter 12). Finally, I show some remaining problems in the area of writing specifications of modular programs (Chapter 13).

Chapter 14 summarizes the thesis and offers some concluding remarks.

Figure 0.0 shows the dependencies between the chapters. I recommend reading the chapters in any order suggested by the partial order in that figure.

0.1.1 Originality and collaboration

Roughly speaking, it is Chapters 2–5, 8, and 10–12 that contain the new work presented in this thesis. The other chapters present pertinent material from the literature, composed and presented in such a way as to set the stage for the chapters containing new material.

Most of Chapters 1 and 6 is a composition of well-known work in semantics (*cf.* [17, 51, 70, 67, 2, 40, 29, 33]). Chapters 2–4 contain joint work with Jan L.A. van de Snepscheut, and are published in [51]. Although the program in Chapter 5 occurs in [51], the derivation of this program and the heuristic used in that process appear first in [50]. Chapter 7 is based on [40, 37, 17, 58], and the first part of Chapter 8 is from [15]. The portion of Chapter 8 that concerns object simplicity, however, resulted from work I did with Greg Nelson at Digital's Systems Research Center. The ideas in Chapter 9 are mostly from [38]. Chapters 10–12 stem from joint work with Greg Nelson. Chapter 13 presents my assessment of the consequences of the work in Chapters 10–12, and some future directions for that work.

0.2 Preliminaries

In this section, I explain the notation I use in this thesis. I take most of the notation and proof format from [21]. I assume familiarity with the predicate calculus (see, *e.g.*, [21, 31, 19]).

Functions

Function application is written with an infix dot. For example, a function f applied to a value a is denoted $f.a$. Function application is left associative. Thus, $wp.S.Q$ means $(wp.S).Q$.

Substitution

For a list of identifiers x and an (equally long) list of expressions y, I use $[x := y]$ as the postfix, left-associative substitution function. The expression

$$Q[x := y]$$

denotes Q in which all free occurrences of (each identifier in) x are replaced by (the corresponding expression in) y.

Operators and binding powers

Function application ($.$) and substitution ($[:=]$) have the highest binding power. Then comes negation ($\neg$), followed by the arithmetic and set operators whose relative binding powers are the usual ones. Next are relations like $=$ (when used in predicates and arithmetic expressions), $\leq$, and $<$.

The remaining boolean connectives are given lower binding power. Among them, $\wedge$ and $\vee$ bind the strongest, then $\Rightarrow$ and $\Leftarrow$, and last $\equiv$.

The operators of statement compositions bind weaker than the boolean connectives. Of these, ; and $\lhd$ have the strongest binding power, then $\rightarrow$, and last $\square$ and $\boxtimes$. Refinement ($\sqsubseteq$) is given lower binding power than all of these.

In the equality among programs or among predicates, $=$ binds weaker than any other operator.

Predicates

A *predicate* is a boolean function over some state space. Conjunction ($\wedge$, "and") and disjunction ($\vee$, "or") are examples of operators on predicates. The universal quantification, $\forall$ (conjunction), of an expression T over the values of a list of variables x constrained by a predicate R, is written

$$\langle \forall x \mid R \triangleright T \rangle \qquad .$$

The expression can be read as "for all x such that R holds, T holds" or "the conjunction over x such that R of T". For example,

$$\langle \forall \theta \mid 0 \leq \theta \leq \pi \triangleright 0 \leq sin.\theta \rangle$$

expresses that for all values of θ satisfying $0 \leq \theta \leq \pi$, the sine of θ is at least 0. In the general expression, x is called the *dummy*, R the *range* of the dummy, and T the *term* of the quantified expression.

Existential quantification, $\exists$ (disjunction), and union quantification, $\cup$, are written similarly.

If the range is *true* or if it is understood from the context, then it is often omitted for brevity. The quantified expression is then written

$$\langle \forall x \triangleright T \rangle \qquad ,$$

and similarly for the other quantifiers. Rules for manipulating quantifiers can be found in, *e.g.*, [21, 31].

A function from predicates to predicates is called a *predicate transformer*.

Everywhere and lifting

For any predicate P, $[P]$ (pronounced "P *everywhere*") denotes that P holds in every state. The brackets $[\,]$ are called *everywhere* brackets. Thus, for a predicate on the state space whose variables are denoted by z, $[P]$ is a shorthand for

$$\langle \forall z \triangleright P.z \rangle \qquad .$$

In general, $[P]$ is preferred to $\langle \forall z \triangleright P.z \rangle$, because it allows expressions to omit irrelevant details. Similarly, I often write

$$(P \Rightarrow Q).z$$

instead of

$$P.z \Rightarrow Q.z \qquad ,$$

and similarly for operators other than $\Rightarrow$. This process is called *lifting*. A special case is $P \equiv Q$, which denotes the predicate that is *true* in exactly those states where P and Q yield the same value. Consequently, $[P \equiv Q]$ means that P and Q are equal as predicates, a fact that is frequently expressed as $P = Q$.

A predicate whose value, as a function, equals the predicate *true* or the predicate *false* is called a *boolean scalar*. For example, $[P]$ is a boolean scalar.

Junctivity, distributivity, and monotonicity

A function f is *conjunctive* if it distributes over conjunction, that is,

$$f.\langle \forall X \mid X \in B \triangleright X \rangle \;=\; \langle \forall X \mid X \in B \triangleright f.X \rangle \qquad . \tag{0.0}$$

A description of the bags (multisets) B of predicates for which this property holds is usually used as a prefix of "conjunctive". For example, f is said to be *universally* conjunctive if (0.0) holds for any B, *positively* conjunctive if (0.0) holds for nonempty bags B ("positively" refers to the positive cardinality of B), and so on.

Similarly, a function is *disjunctive* if it distributes over disjunction.

As junctivity is simply distributivity, "universally distributive over conjunction" or "universally conjunction-distributive" are synonymous to "universally conjunctive". This notion then also extends to, for example, a function being positively *union-distributive*, meaning that it distributes over any nonempty bag of sets.

A function is positively, finitely, and linearly conjunctive (disjunctive), *i.e.*, it distributes over any nonempty finite conjunction (disjunction, respectively) of predicates totally ordered by $[\;\Rightarrow\;]$, exactly when it is *monotonic* [21]. Thus, conjunctivity (or disjunctivity) implies monotonicity. Monotonicity of a predicate transformer f is often written

$$\langle \forall X, Y \triangleright [X \Rightarrow Y] \Rightarrow [f.X \Rightarrow f.Y] \rangle \qquad .$$

In proof hints (see below), I often abbreviate "distribution of $\vee$ over $\forall$" by "$\vee$ over $\forall$", and similarly for other operators.

Sets

I use standard mathematical notation for sets and pairs. Operator $\setminus$ denotes asymmetric set difference, defined for any sets A and B as

$$\langle \forall x \triangleright x \in A \setminus B \equiv x \in A \wedge x \notin B \rangle \qquad .$$

Deviating from standard mathematical notation, I write a quantified set constructor in a notation similar to that of other quantifiers. For example,

$$\{ n \mid 0 \le n < N \triangleright n^2 \}$$

is the set of the first N squares.

The definition of the set constructor can be written

$$\langle \forall y \triangleright\ y \in \{x \mid R \triangleright\ T\} \equiv \langle \exists x \mid R \triangleright\ y = T \rangle\rangle \qquad .$$

Alternatively, the definition can be written

$$\{x \mid R \triangleright\ T\} = \langle \cup x \mid R \triangleright\ \{T\}\rangle \qquad ,$$

where the latter is a quantification over set union. Quantified set constructors can be used, for example, in expressing properties like

$$A \setminus B \;=\; \{x \mid x \in A \wedge x \notin B \triangleright\ x\} \qquad .$$

As for other quantified expressions, if the range is *true* or is understood, I omit the range and simply write

$$\{x \triangleright\ T\} \qquad .$$

Proof format

I use an explicit proof format proposed by W.H.J. Feijen. It gives a hint for each step. Let me give an example. For any positive integer x, let $P.x$ denote the (unique) bag of prime factors whose product equals x. Then, the calculation, for all x,

$$\begin{array}{ll}
 & even.(x^2) \\
= & \quad \{\ even.y \ \equiv\ 2 \in P.y \text{, with } y := x^2 \ \} \\
 & 2 \in P.(x^2) \\
= & \quad \{\ P.(y \cdot z) = P.y \cup P.z \text{, with } y, z := x, x \ \} \\
 & 2 \in P.x \cup P.x \\
= & \quad \{\ y \in X \cup Y \ \equiv\ y \in X \ \vee\ y \in Y \text{, with } y, X, Y := 2, P.x, P.x \ \} \\
 & 2 \in P.x \ \vee\ 2 \in P.x \\
= & \quad \{\ \vee \text{ is idempotent } \} \\
 & 2 \in P.x \\
= & \quad \{\ even.y \ \equiv\ 2 \in P.y \text{, with } y := x \ \} \\
 & even.x
\end{array}$$

demonstrates

$$\langle \forall x \triangleright\ even.(x^2) \ \equiv\ even.x \rangle \qquad .$$

For predicates, a calculation like

$$\begin{array}{ll}
 & A \\
= & \quad \{\ \text{hint why } [A \ \equiv\ B] \ \} \\
 & B \\
\Rightarrow & \quad \{\ \text{hint why } [B \ \Rightarrow\ C] \ \} \\
 & C \\
= & \quad \{\ \text{hint why } [C \ \equiv\ D] \ \} \\
 & D
\end{array}$$

shows that $[A \ \Rightarrow\ D]$.

Programming languages

I often make references to common programming languages. I refer more often to Modula-3 [71] than to other languages. However, principles and techniques generally extend to other languages like Ada [1], Modula-2 [81], Pascal [42], C [45], or C++ [23] as well.

Nomenclature

I do not make any distinction between the terms *statement*, *command*, and *guarded command*. Each refers to a component of a program. I use these terms interchangeably.

A *client* of an interface is another module or interface that makes use of the first interface.

Verification process refers to the process of verifying a program. The process may be conducted by a human or by a machine.

Part I

Control Structures

Imperative programming languages

My goal is to reason about the correctness of programs. I focus our attention on sequential imperative languages, where I have in mind some language like (the sequential subset of) Modula-3, Ada, Modula-2, Pascal, or maybe even a disciplined subset of C or C++. These languages provide a variety of program constructs. Since many of these constructs are but variations of, or shorthands for, other constructs, we find that we can write most of the constructs in Dijkstra's guarded command language [17], with some extensions [51, 70, 67].

The advantage of using guarded commands as the programming notation is that we have a simple, precise, and concise mathematical meaning, or *semantics*, for such programs. The idea is to map each program to a *predicate transformer* [17]. A *predicate* is a boolean function on the state space of a program, and a predicate transformer is a function from predicates to predicates. Predicate transformers, as do predicates, draw their mathematical properties from complete boolean lattices [82, 7, 76]. The particular mappings from programs to predicate transformers that I use are called *weakest precondition* and *weakest liberal precondition* [17].

Outline

The structure of Part I is as follows. Chapter 1 introduces the program constructs and their weakest-precondition semantics.

Chapters 2 through 5 are concerned with *exceptions*—a form of structured jumps. Motivated by the weakest preconditions of exceptional statements, I show a nice algebra over functions of two arguments in Chapter 2. In Chapter 3, I justify the particular weakest preconditions given to the basic exceptional statements and compositions. I do that by showing a more concrete semantics (*viz.*, a *trace* semantics) for exceptions, from which I *derive* the weakest preconditions. In Chapter 4, I deal with the use of exceptions, and develop a theorem that suggests a method for using exceptions in program development. An example application of that theorem is presented in Chapter 5, where I show a novel derivation of a simple program.

Chapters 2 through 5 are quite independent of each other, and also of the subsequent material in this thesis. Thus, the reader can study those chapters according to interest. To guide in that selection, the following table associates these chapters with interest areas.

Chapter 2	Algebra
Chapter 3	Semantics
Chapter 4	Programming methodology
Chapter 5	Program derivation

The material presented in Chapters 2 through 5 appears in modified form in [51] and [50].

I end this Part with Chapter 6, which concerns the use of the semantics from Chapter 1 to model popular programming constructs in common programming languages.

Chapter 1

Semantics of programs with exceptions

In this chapter, I define the control structures of the programming notation (*guarded commands*) I use in this thesis. I define each statement in terms of its mathematical interpretation, *viz.*, its *weakest precondition* and *weakest liberal precondition.* In Chapter 6, I deal with the relation between these statements and those found in common programming languages, whenever this relation is not immediately apparent. I conclude the present chapter by defining the notion of *refinement.*

1.0 Weakest precondition

For any statement S and predicate Q on the final state of S, Dijkstra [17] defines $wp.S.Q$ to be a predicate on the initial state of S:

> $wp.S.Q$ is $true$ of exactly those initial states from which execution of S is guaranteed to terminate and to terminate in a state satisfying Q.

I consider program statements that have *two* ways of terminating, *normally* and *exceptionally.* Therefore, the weakest precondition of a statement maps a *pair* of predicates on the final state to a predicate on the initial state. I use the notation $wp.S.(P, Q)$ and present the following interpretation [13, 60, 51].

> $wp.S.(P, Q)$ is $true$ of exactly those initial states from which execution of S is guaranteed to terminate and to either terminate exceptionally in a state satisfying P or normally in a state satisfying Q.

So, for example, if, for some statement S,

$$wp.S.(false, true) = true \qquad ,$$

then S always terminates normally, never exceptionally, because no state satisfies $false$.

I am restricting my attention to programs with one exceptional outcome. A generalization to an arbitrary number of outcomes is straightforward (see the aforementioned references, [61], Section 2.6, or Section 6.2).

1.0.0 Termination or lack thereof

The fact that the weakest precondition captures that programs do terminate is referred to as *total correctness*. However, when verifying nontrivial programs, we are often willing to settle for less or to prove termination separately. For that purpose, we can consider another attribute of a program statement: its *weakest liberal precondition* (*wlp*) [17]. Like *wp* , *wlp* maps a program to a predicate transformer, but *wlp* only guarantees that the postcondition will be reached *if* the program terminates. This is called *partial correctness*.

I introduce *wlp* for exceptional program S and postcondition pair (P,Q) as follows.

> $wlp.S.(P,Q)$ is *true* of exactly those initial states from which execution of S is guaranteed to terminate exceptionally in a state satisfying P or normally in a state satisfying Q or to not terminate at all.

Stated differently, $wp.S.(P,Q)$ guarantees that S terminates exceptionally in P or normally in Q, whereas $wlp.S.(P,Q)$ only guarantees that S will not terminate exceptionally in $\neg P$ nor normally in $\neg Q$.

Note that $wp.S$ and $wlp.S$ differ only if S might not terminate. For brevity, I introduce only *wp* in this chapter. Except for the iterative statement, the equation defining the *wlp* of each statement I introduce is the same as the equation defining the *wp* of the statement but with every occurrence of *wp* replaced by *wlp* .

1.0.1 Monotonicity

In the next several sections, I introduce the program statements and compositions of a simple programming notation. Two kinds of monotonicity are of importance. First is the monotonicity of $wp.S$, for each statement S. $wp.S$ is monotonic (with respect to $[\ \Rightarrow\ ]$—"implication everywhere") if for all predicates P, P', Q, Q', we have

$$[P \Rightarrow P'] \wedge [Q \Rightarrow Q'] \Rightarrow [wp.S.(P,Q) \Rightarrow wp.S.(P',Q')] \qquad .$$

The importance of this monotonicity will be clear in Section 1.6.

There is an ordering on commands called the *refinement ordering*, explained in Section 1.8. The other important kind of monotonicity is that every statement composition is monotonic with respect to this refinement ordering in its constituent statements. The importance of this monotonicity is explained in Section 1.8.

Every statement composition I introduce satisfies the second kind of monotonicity, and every simple statement I introduce satisfies the first kind of monotonicity. Consequently, every command that can be constructed from my simple statements and statement compositions satisfies the first kind of monotonicity.

1.1 Assignment

The state of a program consists of a number of independent coordinates called *variables*. The *assignment* statement updates the values of these variables.

I now define the assignment statement. As previously advertised, I do so by giving its weakest precondition. For any list v of program variables and an (equally long) list E of expressions, I define $v := E$ by

$$wp.(v := E).(P, Q) \;=\; Q[v := E] \qquad . \tag{1.0}$$

The right-hand side of this formula is the predicate Q with every free occurrence of v replaced by E (see Section 0.2). The operational interpretation of $v := E$ is that the list of expressions E is computed, after which variables v are updated with the respective computed values of E. For example,

$$x, y := y, x$$

has the effect of swapping the values of variables x and y.

I assume that E is *total*, meaning that E is defined in every state in which the command is ever executed. I discuss *partial* expressions in Section 6.3.

Here, and throughout this thesis, I assume the evaluation of expressions to have no effect on the program state. Programs written in common programming languages often contain expressions *with* such so-called *side effects*, a topic I treat in Section 6.3.

We calculate,

$$\begin{array}{ll} & wp.(x := E).(\mathit{false}, \mathit{true}) \\ = & \quad \{ \;\; := \;\; \} \\ & \mathit{true}[x := E] \\ = & \quad \{ \;\; \text{substitution} \;\; \} \\ & \mathit{true} \qquad . \end{array}$$

This calculation lets us conclude that the assignment statement always terminates normally.

1.2 Unit statements and compositions

I now define two statements that do not modify the program state, skip and raise. The former always terminates normally, the latter always exceptionally.

$$wp.\mathit{skip}.(P, Q) \;=\; Q \tag{1.1}$$
$$wp.\mathit{raise}.(P, Q) \;=\; P \tag{1.2}$$

Sequential (normal) composition of two statements S and T, written $S;T$ (and pronounced "S *semi* T"), is defined as

$$wp.(S;T).(P, Q) \;=\; wp.S.(P, wp.T.(P, Q)) \qquad . \tag{1.3}$$

In words, $wp.(S;T).(P,Q)$ is *true* of those initial states from which either S terminates exceptionally in P (then T is ignored), or S terminates normally in a state from which T either terminates exceptionally in P or terminates normally in Q.

I also define *exceptional composition*, also known as the *exception handler*. It is written $S \lhd T$ (and pronounced "S *try* T"—that's "try", almost as in "tri-angle"). The idea is that T "*handles*" any exception raised by S.

$$wp.(S \lhd T).(P,Q) \;=\; wp.S.(wp.T.(P,Q),Q) \tag{1.4}$$

Hence, $wp.(S \lhd T).(P,Q)$ is *true* of those initial states from which either S terminates normally in Q (then the handler T is ignored), or S terminates exceptionally in a state from which the handler T terminates exceptionally in P or normally in Q.

> **Remark 1.0.** If it were not clear from the above English descriptions of $;$ and $\lhd$, it is certainly clear from formulas (1.3) and (1.4) that there is some duality between the two program compositions. Reviewing (1.1) and (1.2), we also detect a duality. Indeed, by identifying a program with its weakest precondition, we find that functions *skip* and *raise* project to one of the two components of a pair. I will write these functions as R and L, respectively. Furthermore, we can write $;$ as *right composition* and $\lhd$ as *left composition*, denoted $\circ\rangle$ and $\langle\circ$, respectively, over functions of some type $D \times D \to D$. That is, for any domain D, functions $f,g\colon D \times D \to D$, and elements $p,q \in D$, we have
>
> $$\begin{aligned} (f \langle\circ\, g).(p,q) &= f.(g.(p,q),q) \quad , \text{ and} \\ (f \circ\rangle\, g).(p,q) &= f.(p,g.(p,q)) \quad . \end{aligned}$$
>
> Propelled by this discovery, I explore the phenomenon in Chapter 2.

Other convenient statements that can be defined in terms of *skip*, *raise*, $;$, and $\lhd$ are presented in Section 2.5.

1.3 Block

The *block* statement, written

$$[\![\, v \bullet S \,]\!] \quad ,$$

where v is a list of identifiers, introduces local variables v for use in S, the *body* of the block statement.

$$wp.([\![\, v \bullet S \,]\!]).(P,Q) \;=\; \langle\, \forall v \triangleright\; wp.S.(P,Q) \,\rangle \tag{1.5}$$

In words, the block statement guarantees (P,Q) upon termination exactly when $wp.S.(P,Q)$ holds initially, *for any value of* v. Thus, S must not depend on v having any particular initial value.

1.4 Partial commands

The weakest precondition of a command S is said to be *strict* just when

$$wp.S.(\mathit{false}, \mathit{false}) \;=\; \mathit{false} \qquad .$$

In [17], this property is referred to as the (Law of the) Excluded Miracle, because statements that lack this property do not, in general, lend themselves to a practical implementation—$wp.S.(\mathit{false}, \mathit{false})$ characterizes those initial states from which S is guaranteed to terminate in a state satisfying *false*!

A statement S whose weakest precondition is not strict is called *partial* [60, 70] (or *miraculous* or *feasible* [67]), because one may think of it as being executable only from the initial states satisfying

$$\neg wp.S.(\mathit{false}, \mathit{false}) \qquad .$$

(See also Remark 1.1 below.) A command that is not partial is said to be *total*.

Despite the fact that they do not always admit a realistic implementation, partial commands are important and useful when handled with care [70, 67, 66]. I give some examples below.

1.4.0 The guard statement

A primitive partial command is the *guard* statement, $g \to S$, where g is a predicate and S is a command. Operator $\to$ binds stronger than $;$ and $\lhd$, but weaker than logical connectives. The non-exceptional definition of the guard statement is

$$wp.(g \to S).Q \quad = \quad \neg g \vee wp.S.Q \qquad .$$

I extend this definition in the obvious way.

$$wp.(g \to S).(P, Q) \quad = \quad \neg g \vee wp.S.(P, Q) \tag{1.6}$$

Note that (1.6) can also be written

$$wp.(g \to S).(P, Q) \quad = \quad g \Rightarrow wp.S.(P, Q) \qquad ,$$

which has appeal because $\to$ and $\Rightarrow$ look similar.

An operational interpretation of the guard statement is that $g \to S$ is like S, except that in addition to the states from which S cannot be started, $g \to S$ cannot be started in states where g does not hold. An alternative operational interpretation is that $g \to S$ “invokes a miracle” when g does not hold and invokes S otherwise.

> **Remark 1.1.** In the first of these interpretations, one can let the execution of a partial command from a state in which it cannot begin cause the entire program to backtrack. This is what the text processing language LIM does [10]. Nevertheless, I will continue to use the terminology “invoke a miracle”.

For a total command S, g is called the *guard* of the command $g \to S$. This concept can be generalized: The *guard* of any (total or partial) command S, denoted $guard.S$, is defined by

$$guard.S \;=\; \neg wp.S.(false, false) \qquad .$$

This is, the guard of a statement characterizes those initial states from which execution of S can start.

Note that

$$guard.S \to S \;=\; S \qquad .$$

In fact, for any g such that $[guard.S \;\Rightarrow\; g]$,

$$g \to S \;=\; S \qquad .$$

This shows, for example, that $g \to$ is idempotent, *i.e.*,

$$g \to (g \to S) \;=\; g \to S \qquad .$$

It also shows that $true$ is a left identity of $\to$.

$$true \to S \;=\; S$$

1.4.1 Examples

I give a few examples. I make use of the *assert* statement, which I discuss in Section 6.2.0. For now, all we need to know is that **assert** g terminates normally just when g holds initially.

$$wp.(\textbf{assert}\ g).(P, Q) \;=\; g \wedge Q$$

If S is total, then $g \to S$ invokes a miracle precisely when g does not hold initially. Thus, despite its use of $\to$,

$$\textbf{assert}\ g\ ;\ g \to S$$

is total, because the statement $g \to S$ is only reached if g holds. In general, for any S,

$$\textbf{assert}\ guard.S\ ;\ S$$

is total.

Using a guard statement as the body of a block proves convenient. For example, for a total command S, executing

$$[\![\ x \bullet x^2 = 9 \to S\ ]\!]$$

has the effect of executing S from a state where $x = 3 \vee x = -3$ holds. The reader is invited to prove

$$wp.([\![\ x \bullet x^2 = 9 \to \textbf{assert}\ x = 3 \vee x = -3\ ]\!]).(false, true) \;=\; true \qquad .$$

Remark 1.2. Juno-2 [35] is a language that achieves expressive power by taking advantage of partial commands in this way.

Other convenient uses of blocks with partial commands are described in [70].

Partial commands come in handy also when developing programs through refinements, as is shown, for example, in [67, 66].

1.5 Choice compositions

Next, I introduce two *choice* compositions, $\square$ ("*box*") and $\boxtimes$ ("*else*") [70], each with lower binding power than $\rightarrow$.

$$wp.(S \,\square\, T).(P,Q) \;=\; wp.S.(P,Q) \wedge wp.T.(P,Q)$$
$$S \boxtimes T \;=\; guard.S \rightarrow S \,\square\, \neg guard.S \rightarrow T$$

We have that $\square$ is associative and so is $\boxtimes$ (the fact that $\square$ is follows directly from its definition; the fact that $\boxtimes$ is makes a nice exercise for the reader). Beware that, contrary to what the appearance of its symbol suggests, $\boxtimes$ is not symmetric.

The execution of $S \,\square\, T$ consists of the execution of exactly one of S and T, and so does the execution of $S \boxtimes T$. The difference is that while $S \,\square\, T$ guarantees nothing about which of S and T is chosen for execution, execution of $S \boxtimes T$ reduces to execution of S whenever $S \boxtimes T$ is started in a state where execution of S can begin (*i.e.*, does not invoke a miracle) and reduces to T otherwise.

An example of a choice composition is

$$g0 \rightarrow S0 \,\square\, g1 \rightarrow S1 \,\square\, g2 \rightarrow S2 \qquad . \tag{1.7}$$

If $S0, S1, S2$ are total commands, execution of (1.7) can result in the execution of $S0$ if $g0$ holds (initially), $S1$ if $g1$ holds, and $S2$ if $g2$ holds. If $g0 \vee g1 \vee g2$ holds initially, exactly one of $S0, S1, S2$ is executed; if $\neg g0 \wedge \neg g1 \wedge \neg g2$ holds initially, then (1.7) invokes a miracle.

The statement

$$g0 \rightarrow S0 \boxtimes g1 \rightarrow S1 \boxtimes g2 \rightarrow S2 \tag{1.8}$$

is like (1.7), except that $S0$ is chosen just when $g0$ holds, $S1$ just when $\neg g0 \wedge g1$ holds, and $S2$ just when $\neg g0 \wedge \neg g1 \wedge g2$ holds. The statement

$$g0 \rightarrow S0 \boxtimes g1 \rightarrow S1 \boxtimes g2 \rightarrow S2 \boxtimes skip \tag{1.9}$$

is similar to (1.8), except that it terminates without changing the state (it "skips") where (1.8) invokes a miracle. (Note that, since $true$ is a left identity element of $\rightarrow$, the last $skip$ can also be written as $true \rightarrow skip$.)

For similarity with common notation, I permit myself to surround *total* commands with **if fi** brackets. So that I don't need to attach any meaning to these brackets, I will refrain from the use of **if** S **fi** if S is partial. So, I may write (1.9) equivalently as

$$\textbf{if } g0 \rightarrow S0 \boxtimes g1 \rightarrow S1 \boxtimes g2 \rightarrow S2 \boxtimes true \rightarrow skip \textbf{ fi} \qquad . \tag{1.10}$$

1.6 Iteration

Iteration of a statement S is written $\textbf{do}\ g \mapsto S\ \textbf{od}$.

> **Remark 1.3.** The symbol $\mapsto$ is not to be confused with the guard operator $\rightarrow$. The construct $\textbf{do}\ g \mapsto S\ \textbf{od}$ is a construct parameterized by a predicate g and a statement S.

Let this *iterative statement* (or *loop*) be denoted by DO. Operationally, the execution of DO consists of repeated executions of S for as long as g holds. We think of DO as satisfying

$$DO \quad = \quad g \rightarrow S; DO \ \square\ \neg g \rightarrow skip \qquad .$$

Inspired by this, I define $wp.DO.(P, Q)$ as the least fixed point of the equation

$$X \quad = \quad (g \Rightarrow wp.S.X) \wedge (\neg g \Rightarrow Q) \qquad , \tag{1.11}$$

solved for X. Because $wp.S$ is monotonic (*cf.* Section 1.0.1), the right-hand side of (1.11) is a monotonic function of X; hence, due to the Knaster-Tarski Theorem (see, *e.g.*, [76]), (1.11) does indeed have a least fixed point, so $wp.DO.(P, Q)$ is well-defined. It is for this reason that the first kind of monotonicity mentioned in Section 1.0.1 is important.

> **Remark 1.4.** The fact that the loop satisfies the second kind of monotonicity mentioned in that section follows from the fact that the right-hand side of (1.11) is monotonic in S (see, *e.g.*, (130) in [76]). Note that this property holds even for partial statements S.

Similarly, $wlp.DO.(P, Q)$ is defined as the *greatest* fixed point of

$$X \quad = \quad (g \Rightarrow wlp.S.X) \wedge (\neg g \Rightarrow Q) \qquad ,$$

solved for X (*cf.* [21]).

The fact that the definition of DO involves a fixed point may appear worrisome—if proving something about a loop would require computing a particular fixed point, the practical application of the semantics of the loop would be hampered. However, the Invariance Theorem [21] shows a sufficient condition for proving that a loop establishes a particular postcondition. It states that

$$[Pre \Rightarrow wp.(\textbf{do}\ g \mapsto S\ \textbf{od}).(P, Q)] \tag{1.12}$$

follows from

$$\langle \exists J \triangleright [Pre \Rightarrow J] \wedge [J \wedge g \Rightarrow wp.S.(P, J)] \wedge [J \wedge \neg g \Rightarrow Q] \rangle \tag{1.13}$$

and a proof that the loop eventually terminates. The J in (1.13) is called the *loop invariant*. Equation (1.13) states that there exists an invariant J that satisfies three conditions. First, the invariant holds prior to the execution of the loop. Second, provided the invariant holds initially, an execution of the loop body terminates exceptionally in a state satisfying P (upon which the loop terminates) or normally in a state satisfying, once again, the invariant. Third, the invariant conjoined with the negation of the guard imply Q, the desired normal postcondition of the loop.

The existential quantification in (1.13) may look intimidating because J ranges over all predicates. However, the programmer who writes the loop has a good idea of what an invariant of the loop may be. Having the programmer supply that invariant (J) simplifies (1.13) to the satisfiability of each of the conjuncts in the term of its quantification.

> **Remark 1.5.** Instead of having the programmer provide the invariant, methods of "widening" and "narrowing" can be used in an attempt to synthesize a proper invariant (see [12, 8]).

Termination can be handled in a similar way by letting the programmer supply a *bound function* [17]. If termination is not of concern, *wlp* can be used. The *wlp* equation corresponding to (1.12) follows from the *wlp* equation corresponding to (1.13) alone, without any further proof of termination.

The loop that never terminates, in its simplest form written as **do** $\mathit{true} \mapsto \mathit{skip}$ **od**, is often referred to as *abort* [17].

1.7 Specification statement

As a final statement, I introduce the *specification statement* [67]. Its non-exceptional form takes the shape

$$w : [\mathit{Pre},\ \mathit{Post}] \qquad ,$$

where w (called the *frame*) is an (unordered) list of variables, and Pre (the *precondition*) and Post (the *postcondition*) are predicates. If started in a state satisfying Pre, the statement will terminate in a state satisfying Post, having modified the values only of variables w. The weakest precondition of this statement is thus given by

$$wp.(w : [\mathit{Pre},\ \mathit{Post}]).Q \;=\; \mathit{Pre} \wedge \langle \forall w \mid \mathit{Post} \triangleright Q \rangle \qquad . \tag{1.14}$$

That is, for $w : [\mathit{Pre},\ \mathit{Post}]$ to establish Q, Pre must hold initially. In addition, the following must hold initially: For any values of variables w that satisfy Post, Q holds. (I defer until Section 6.2 describing what the operational interpretation of the statement is when Pre does not hold initially.)

We often want to specify the final values of w in terms of their initial values. We can then think of "saving" the initial values of some variables v (admittedly usually a subset of w), as in

$$[\![\ v_0 \bullet v_0 := v\ ;\ w : [\mathit{Pre},\ \mathit{Post}]\ ]\!] \qquad . \tag{1.15}$$

Then, $Post$ can refer to v_0. I abbreviate (1.15) simply as

$$w : [Pre,\ Post] \qquad ,$$

where $Post$ may subscript the name of any variable with 0 to refer to the initial value of that variable. I assume that actual names of program variables never end with a subscripted 0. I refer to a variable subscripted with 0 as an *initial-value* variable. (The terms *entry*, *logical*, and *mathematical* variables are also in use.) $Post$ can be thought of as a *two-state* predicate, since it is a predicate on both the initial and final states.

Note that w and Pre do not mention any initial-value variables. Also note that it doesn't change the semantics of the specification statement if more variables are saved in (1.15) than are needed in $Post$; hence, when referring to the initial-value variables in $Post$, we may safely consider any superset of those initial-value variables that actually occur in $Post$.

Let v_0 be (any superset of) the initial-value variables in $Post$. The following calculation arrives at the definition of the specification statement that permits initial-value variables.

$$\begin{array}{ll}
& wp.(w : [Pre,\ Post]).Q \\
= & \qquad \{ \ \text{shorthand for (1.15)} \ \} \\
& wp.(\llbracket\ v_0 \bullet v_0 := v\ ;\ w : [Pre,\ Post]\ \rrbracket).Q \\
= & \qquad \{ \ \text{(1.5): } wp \text{ of } \llbracket \bullet \rrbracket \ \} \\
& \langle \forall\, v_0 \triangleright wp.(v_0 := v\ ;\ w : [Pre,\ Post]).Q \,\rangle \\
= & \qquad \{ \ \text{(1.3,1.0): } wp \text{ of } ; \text{ and } := \ \} \\
& \langle \forall\, v_0 \triangleright (wp.(w : [Pre,\ Post]).Q)[v_0 := v] \,\rangle \\
= & \qquad \{ \ v_0 \text{ does not occur free in term of quantification} \ \} \\
& (wp.(w : [Pre,\ Post]).Q)[v_0 := v] \\
= & \qquad \{ \ \text{(1.14): } wp \text{ of specification statement without initial-value variables} \ \} \\
& (Pre \wedge \langle \forall\, w \mid Post \triangleright Q \,\rangle)[v_0 := v] \\
= & \qquad \{ \ v_0 \text{ does not occur free in } Pre \ \} \\
& Pre \wedge \langle \forall\, w \mid Post \triangleright Q \,\rangle[v_0 := v]
\end{array} \tag{1.16}$$

> **Remark 1.6.** The observation about $v_0 := v$ is one I thought to be folklore. Later, I traced it back to having been a discovery [18, pp. 217–219]. Maybe this attests how the general understanding of semantics has grown during the last two decades.

In the realm of exceptional programs, I extend the postcondition in the specification statement to be a pair of predicates.

$$\begin{array}{l}
wp.(w : [Pre,\ (ePost, nPost)]).(P, Q) \quad = \\
\qquad Pre \wedge \langle \forall\, w \triangleright (ePost \Rightarrow P) \wedge (nPost \Rightarrow Q) \,\rangle[v_0 := v]
\end{array} \tag{1.17}$$

Note that the specification statement can be a partial command, *e.g.*, $w : [true,\ (false, false)]$.

1.8 Refinement

The specification statement offers a high-level notation that conveniently expresses *what* the command does. However, not only does it have the possibility of being partial, the specification statement cannot easily be compiled automatically into more primitive statements, because it does not give any clues as to *how* the command is to arrive at the specified final state. The statement thus lends itself well to writing a specification, the implementation of which requires guidance from the programmer. We want to be able to prove that the implementation meets the specification. This leads to the concept of *program refinement*, first proposed by Dijkstra [16] and Wirth [80], and first given a mathematical foundation by Back [2].

Informally, a statement S is *refined by* a statement T, denoted $S \sqsubseteq T$, just when T meets any specification that S does. Formally, the non-exceptional form of $\sqsubseteq$ is defined by

$$(S \sqsubseteq T) \quad = \quad \langle \forall R \triangleright [wp.S.R \Rightarrow wp.T.R] \rangle \qquad , \tag{1.18}$$

where R ranges over all predicates on the final state. In the complete boolean lattice of predicate transformers, elements are ordered by $\sqsubseteq$, which is the lifting of $[\;\Rightarrow\;]$ in the complete boolean lattice of predicates [82]. If wlp, not wp, is of interest, then (1.18) is written with wp replaced by wlp.

Examples of refinements are

$$x : [true,\, x_0 < x] \;\sqsubseteq\; x := x + 3$$

and

$$S \,\square\, T \;\sqsubseteq\; S \qquad .$$

In both of these examples, the left-hand sides allow a greater degree of nondeterminism than the respective right-hand sides.

So, given $S \sqsubseteq T$, we can always replace the program S by the program T. T is therefore sometimes said to be "better than" S [39]. Note, however, that some commands are "too good". For example, $wp.(false \rightarrow skip).R = true$, so $false \rightarrow skip$ refines (or "is better than") *any* command. However, $false \rightarrow skip$ is a partial command that usually cannot realistically be implemented (the exception is again backtracking, see Remark 1.1). This shows that when refining a program, there is a risk of ending up with a non-implementable program [39, 68].

We may wonder if $S \sqsubseteq T$ allows us to replace S by T in *any* context. That is, if S is a subcomponent of a larger program P, does replacing S by T in P result in a refinement of P? Because this is important in program development, I require that this property hold for all programs under consideration. The fact that S is a subcomponent of P is captured by writing P as a function of S, say $f.S$. Then, the requirement can be written down as

$$(S \sqsubseteq T) \;\Rightarrow\; (f.S \sqsubseteq f.T) \qquad ,$$

a formula we recognize as expressing the monotonicity of f. It is primarily for this reason that I consider only those statement compositions that are monotonic in their constituent statements (see Section 1.0.1).

For exceptional programs S and T, I define refinement by

$$(S \sqsubseteq T) \quad = \quad \langle \forall P, Q \triangleright [wp.S.(P,Q) \Rightarrow wp.T.(P,Q)] \rangle \qquad , \tag{1.19}$$

and similar for the wlp counterpart.

Chapter 2

Functions of two arguments and their compositions

To show that program constructs in a language with exceptions are not appreciably more difficult to reason about than those in a language without exceptions, I show that the weakest preconditions of these constructs make up a nice algebra over functions of two arguments.

I first introduce the algebra, and then, as alluded to in Remark 1.0 in Section 1.2, make the connection with program statements.

2.0 Function compositions

Consider functions f and g of type $D \to D$, for any domain D. We are accustomed to composing these functions, that is, applying one after the other. We use $\circ$ to denote function composition, and recognize its familiar definition, for any element x of D,

$$(f \circ g).x = f.(g.x) \qquad .$$

Now, consider functions f and g of type $D \times D \to D$. These cannot be composed in the same way as the previous functions, because the expression

$$f.(g.(x,y)) \qquad ,$$

where (x,y) is a pair of type $D \times D$, doesn't make any sense since the types don't match up: the expression $g.(x,y)$ has type D, whereas the domain of f is $D \times D$.

From this, we conclude that composing functions of *two* arguments requires an operator different from $\circ$. In fact, such functions can be composed in several different ways.

In the rest of this chapter, I use f, g, and h to denote any functions of type $D \times D \to D$ for any domain D, and p and q to denote any elements in D. As before, an ordered pair with components p and q is written (p,q).

2.1 Left and right composition

Of the different ways functions from pairs to elements can be composed, we first consider *left* and *right* composition, written $\langle\!\circ$ and $\circ\!\rangle$, respectively. I define these as follows.

$$(f \langle\!\circ\, g).(p,q) \;=\; f.(g.(p,q),q) \qquad (2.0)$$
$$(f \circ\!\rangle\, g).(p,q) \;=\; f.(p,g.(p,q)) \qquad (2.1)$$

Theorem

$\langle\!\circ$ is associative. (2.2)

$\circ\!\rangle$ is associative. (2.3)

Proof.

$$\begin{array}{ll}
 & (f \langle\!\circ\, (g \langle\!\circ\, h)).(p,q) \\
= & \quad \{ \quad (2.0)\text{: def. of } \langle\!\circ \quad \} \\
 & f.((g \langle\!\circ\, h).(p,q),q) \\
= & \quad \{ \quad (2.0)\text{: def. of } \langle\!\circ \quad \} \\
 & f.(g.(h.(p,q),q),q) \\
= & \quad \{ \quad (2.0)\text{: def. of } \langle\!\circ \quad \} \\
 & (f \langle\!\circ\, g).(h.(p,q),q) \\
= & \quad \{ \quad (2.0)\text{: def. of } \langle\!\circ \quad \} \\
 & ((f \langle\!\circ\, g) \langle\!\circ\, h).(p,q)
\end{array}$$

I omit the proof of the other case as it is similar to the present case (and will do so in many more proofs). □

Two functions of special interest are L and R, defined as follows.

$$L.(p,q) \;=\; p \qquad (2.4)$$
$$R.(p,q) \;=\; q \qquad (2.5)$$

Theorem

L is the identity of $\langle\!\circ$. (2.6)

R is the identity of $\circ\!\rangle$. (2.7)

Proof.

$$\begin{array}{ll}
 & (L \langle\!\circ\, f).(p,q) \\
= & \quad \{ \quad (2.0)\text{: def. of } \langle\!\circ \quad \} \\
 & L.(f.(p,q),q) \\
= & \quad \{ \quad (2.4)\text{: def. of } L \quad \} \\
 & f.(p,q) \\
= & \quad \{ \quad (2.4)\text{: def. of } L \quad \} \\
 & f.(L.(p,q),q) \\
= & \quad \{ \quad (2.0)\text{: def. of } \langle\!\circ \quad \} \\
 & (f \langle\!\circ\, L).(p,q)
\end{array}$$

□

Theorem

L is a left zero of $\circ\rangle$. (2.8)

R is a left zero of $\langle\circ$. (2.9)

Proof.

$$
\begin{array}{ll}
& (L \circ\rangle\, g).(p,q) \\
= & \quad \{ \quad (2.1)\text{: def. of } \circ\rangle \quad \} \\
& L.(p, g.(p,q)) \\
= & \quad \{ \quad (2.4)\text{: def. of } L \quad \} \\
& p \\
= & \quad \{ \quad (2.4)\text{: def. of } L \quad \} \\
& L.(p,q)
\end{array}
$$

□

2.2 Double composition

As a different way to compose functions of two arguments, I define *double* composition, written $\langle\circ\rangle$.

$$(f \langle\circ\rangle\, g).(p,q) \;=\; f.(g.(p,q), g.(p,q)) \tag{2.10}$$

We have the following correspondences between single (left and right) compositions and double composition.

Theorem

$$f \langle\circ\rangle\, g \;=\; (f \langle\circ\, R) \circ\rangle\, g \tag{2.11}$$

$$f \langle\circ\rangle\, g \;=\; (f \circ\rangle\, L) \langle\circ\, g \tag{2.12}$$

Proof.

$$
\begin{array}{ll}
& ((f \langle\circ\, R) \circ\rangle\, g).(p,q) \\
= & \quad \{ \quad (2.1)\text{: def. of } \circ\rangle \quad \} \\
& (f \langle\circ\, R).(p, g.(p,q)) \\
= & \quad \{ \quad (2.0)\text{: def. of } \langle\circ \quad \} \\
& f.(R.(p, g.(p,q)), g.(p,q)) \\
= & \quad \{ \quad (2.5)\text{: def. of } R \quad \} \\
& f.(g.(p,q), g.(p,q)) \\
= & \quad \{ \quad (2.10)\text{: def. of } \langle\circ\rangle \quad \} \\
& (f \langle\circ\rangle\, g).(p,q)
\end{array}
$$

□

Remark 2.0. I stated this property as an equality between functions; however, its proof applies those functions to an arbitrary pair (p,q) . I strive toward calculations at the level of functions, since they tend to be more concise and easier to read. As it turns out, having shown the above relation between single and double compositions, I am now able to carry out the calculations at the level of functions. This phenomenon is commonly referred to as *lifting* (Section 0.2).

I continue with some theorems regarding the associativity and distributivity of the composition operators.

Theorem

$$(f \langle\circ\rangle g) \langle\circ h = f \langle\circ\rangle (g \langle\circ h) \tag{2.13}$$
$$(f \langle\circ\rangle g) \circ\rangle h = f \langle\circ\rangle (g \circ\rangle h) \tag{2.14}$$

Proof.

$(f \langle\circ\rangle g) \langle\circ h$

$=$ { (2.12): double/single trade, and (2.2): $\langle\circ$ is associative }

$(f \circ\rangle L) \langle\circ g \langle\circ h$

$=$ { (2.12): double/single trade, and (2.2): $\langle\circ$ is associative }

$f \langle\circ\rangle (g \langle\circ h)$ □

Now for a useful theorem whose proof is rather curious—maybe the most interesting proof in this chapter.

Theorem

$\langle\circ\rangle$ is associative. (2.15)

Proof.

$f \langle\circ\rangle (g \langle\circ\rangle h)$

$=$ { (2.11): double/single trade }

$f \langle\circ\rangle ((g \langle\circ R) \circ\rangle h)$

$=$ { (2.14): mutual associativity of $\langle\circ\rangle$ $\circ\rangle$ }

$(f \langle\circ\rangle (g \langle\circ R)) \circ\rangle h$

$=$ { (2.13): mutual associativity of $\langle\circ\rangle$ $\langle\circ$ }

$((f \langle\circ\rangle g) \langle\circ R) \circ\rangle h$

$=$ { (2.11): double/single trade }

$(f \langle\circ\rangle g) \langle\circ\rangle h$ □

Theorem

L and R are left identities of $\langle\circ\rangle$. (2.16)

Proof.

$L \langle\circ\rangle g$

$=$ { (2.11): double/single trade }

$(L \langle\circ R) \circ\rangle g$

$=$ { (2.6): L is identity of $\langle\circ$ }

$R \circ\rangle g$

$=$ { (2.7): R is identity of $\circ\rangle$ }

$$
\begin{array}{ll}
 & g \\
= & \quad \{ \ (2.6)\text{: } L \text{ is identity of } \langle\!\circ \ \} \\
 & L \langle\!\circ\, g \\
= & \quad \{ \ (2.7)\text{: } R \text{ is identity of } \circ\!\rangle \ \} \\
 & (R \circ\!\rangle L) \langle\!\circ\, g \\
= & \quad \{ \ (2.12)\text{: double/single trade } \} \\
 & R \langle\!\circ\!\rangle g
\end{array}
\qquad \square
$$

A consequence of this theorem, since L and R differ, is that $\langle\!\circ\!\rangle$ lacks a right identity. However, $\langle\!\circ\!\rangle$ with L or R as a second argument is still interesting, as is shown by the next theorem.

Theorem

$$f \langle\!\circ\!\rangle L = f \circ\!\rangle L \tag{2.17}$$
$$f \langle\!\circ\!\rangle R = f \langle\!\circ\, R \tag{2.18}$$

Proof.

$$
\begin{array}{ll}
 & f \langle\!\circ\!\rangle L \\
= & \quad \{ \ (2.12)\text{: double/single trade } \} \\
 & (f \circ\!\rangle L) \langle\!\circ\, L \\
= & \quad \{ \ (2.6)\text{: } L \text{ is identity of } \langle\!\circ \ \} \\
 & f \circ\!\rangle L
\end{array}
\qquad \square
$$

I find these instances where $\langle\!\circ\!\rangle$ equals $\circ\!\rangle$ or $\langle\!\circ$ curious—in fact, so curious that I will devote the next section to it.

2.3 Ceiling and floor

Intrigued by (2.17) and (2.18), I introduce some special notation, $\lceil\ \rceil$ and $\lfloor\ \rfloor$, defined as follows.

$$\lceil f \rceil = f \langle\!\circ\!\rangle L \tag{2.19}$$
$$\lfloor f \rfloor = f \langle\!\circ\!\rangle R \tag{2.20}$$

This leads us to the following theorems.

Theorem

$$\lceil L \rceil = L = \lceil R \rceil \tag{2.21}$$
$$\lfloor L \rfloor = R = \lfloor R \rfloor \tag{2.22}$$

Proof.

$$
\begin{array}{ll}
 & \lceil L \rceil \\
= & \quad \{ \ (2.19)\text{: def. of } \lceil\ \rceil \ \} \\
 & L \langle\circ\rangle L \\
= & \quad \{ \ (2.16)\text{: } L \text{ is left identity of } \langle\circ\rangle \ \} \\
 & L \\
= & \quad \{ \ (2.16)\text{: } R \text{ is left identity of } \langle\circ\rangle \ \} \\
 & R \langle\circ\rangle L \\
= & \quad \{ \ (2.19)\text{: def. of } \lceil\ \rceil \ \} \\
 & \lceil R \rceil
\end{array}
$$

□

How we think about an operator influences the notation we choose. This is important, because the notation we choose in turn inspires how we think about the operator! For example, we know well not to make the mistake of writing $+$ and $\cdot$ for $\vee$ and $\wedge$, respectively; doing that effectively hides the fact that $\vee$ distributes over $\wedge$, a property not enjoyed by the arithmetic operators $+$ and $\cdot$ [25]. The following theorem justifies the use of ceiling and floor. These ceiling and floor operators can also be shown to be monotonic (with respect to any ordering over functions of two arguments), just like the ones operating on real numbers.

Theorem

$\lceil\ \rceil$ and $\lfloor\ \rfloor$ are idempotent. (2.23)

Proof.

$$
\begin{array}{ll}
 & \lceil \lceil f \rceil \rceil \\
= & \quad \{ \ (2.19)\text{: def. of } \lceil\ \rceil\text{, twice, and } (2.15)\text{: } \langle\circ\rangle \text{ is associative} \ \} \\
 & f \langle\circ\rangle L \langle\circ\rangle L \\
= & \quad \{ \ (2.16)\text{: } L \text{ is left identity of } \langle\circ\rangle \ \} \\
 & f \langle\circ\rangle L \\
= & \quad \{ \ (2.19)\text{: def. of } \lceil\ \rceil \ \} \\
 & \lceil f \rceil
\end{array}
$$

□

Theorem

$$f \langle\circ\rangle g = \lceil f \rceil \langle\circ\, g \tag{2.24}$$

$$f \langle\circ\rangle g = \lfloor f \rfloor \circ\rangle\, g \tag{2.25}$$

Proof.

$$
\begin{array}{ll}
 & f \langle\circ\rangle g \\
= & \quad \{ \ (2.12)\text{: double/single trade} \ \} \\
 & (f \circ\rangle L) \langle\circ\, g \\
= & \quad \{ \ (2.17)\text{: } \circ\rangle L \text{ and } \langle\circ\rangle L \ \}
\end{array}
$$

$(f \langle\circ\rangle L) \langle\circ\, g$

$=$ { (2.19): def. of $\lceil\ \rceil$ }

$\lceil f\rceil \langle\circ\, g$ $\square$

Theorem

$$\lceil f \langle\circ\rangle g\rceil = \lceil f\rceil \langle\circ\rangle \lceil g\rceil \tag{2.26}$$

$$\lfloor f \langle\circ\rangle g\rfloor = \lfloor f\rfloor \langle\circ\rangle \lfloor g\rfloor \tag{2.27}$$

Proof. Using the associativity of $\langle\circ\rangle$ in every step, we calculate,

$\lceil f \langle\circ\rangle g\rceil$

$=$ { (2.19): def. of $\lceil\ \rceil$ }

$f \langle\circ\rangle g \langle\circ\rangle L$

$=$ { (2.16): L is left identity of $\langle\circ\rangle$ }

$f \langle\circ\rangle L \langle\circ\rangle g \langle\circ\rangle L$

$=$ { (2.19): def. of $\lceil\ \rceil$, twice }

$\lceil f\rceil \langle\circ\rangle \lceil g\rceil$. $\square$

Theorem

$$\lceil f \langle\circ\rangle g\rceil = \lceil f\rceil \langle\circ\, \lceil g\rceil$$

$$\lfloor f \langle\circ\rangle g\rfloor = \lfloor f\rfloor \circ\rangle \lfloor g\rfloor$$

Proof.

$\lceil f \langle\circ\rangle g\rceil$

$=$ { (2.26): $\langle\circ\rangle$ over $\lceil\ \rceil$ }

$\lceil f\rceil \langle\circ\rangle \lceil g\rceil$

$=$ { (2.24) }

$\lceil\lceil f\rceil\rceil \langle\circ\, \lceil g\rceil$

$=$ { (2.23): idempotence of $\lceil\ \rceil$ }

$\lceil f\rceil \langle\circ\, \lceil g\rceil$ $\square$

2.4 Transposition

I introduce operator $\sim$ with higher binding power than composition and function application, defined as follows.

$$\sim f.(p,q) = f.(q,p) \tag{2.28}$$

Clearly, $\sim$ is an *involution*, that is, $\sim\sim$ is the identity function ($\sim\sim f = f$).

> **Remark 2.1.** Having introduced a new operator, defined at the level of pairs, the proof of the next theorem is done at the point level.

Theorem

$$\begin{array}{rcl} \sim(f \langle\!\circ\, g) &=& \sim f \circ\!\rangle \sim g \\ \sim(f \circ\!\rangle\, g) &=& \sim f \langle\!\circ \sim g \\ \sim(f \langle\!\circ\!\rangle\, g) &=& \sim f \langle\!\circ\!\rangle \sim g \end{array}$$

Proof. For left composition, we have,

$$\begin{array}{ll} & \sim(f \langle\!\circ\, g).(p,q) \\ = & \quad \{ \quad (2.28)\text{: def. of } \sim \quad \} \\ & (f \langle\!\circ\, g).(q,p) \\ = & \quad \{ \quad (2.0)\text{: def. of } \langle\!\circ \quad \} \\ & f.(g.(q,p),p) \\ = & \quad \{ \quad (2.28)\text{: def. of } \sim \quad \} \\ & f.(\sim g.(p,q),p) \\ = & \quad \{ \quad (2.28)\text{: def. of } \sim \quad \} \\ & \sim f.(p,\sim g.(p,q)) \\ = & \quad \{ \quad (2.1)\text{: def. of } \circ\!\rangle \quad \} \\ & (\sim f \circ\!\rangle \sim g).(p,q) \qquad , \end{array}$$

and similar for right composition. For double composition, we have,

$$\begin{array}{ll} & \sim(f \langle\!\circ\!\rangle\, g).(p,q) \\ = & \quad \{ \quad (2.28)\text{: def. of } \sim \quad \} \\ & (f \langle\!\circ\!\rangle\, g).(q,p) \\ = & \quad \{ \quad (2.10)\text{: def. of } \langle\!\circ\!\rangle \quad \} \\ & f.(g.(q,p),g.(q,p)) \\ = & \quad \{ \quad (2.28)\text{: def. of } \sim\text{, twice} \quad \} \\ & f.(\sim g.(p,q),\sim g.(p,q)) \\ = & \quad \{ \quad (2.28)\text{: def. of } \sim \quad \} \\ & \sim f.(\sim g.(p,q),\sim g.(p,q)) \\ = & \quad \{ \quad (2.10)\text{: def. of } \langle\!\circ\!\rangle \quad \} \\ & (\sim f \langle\!\circ\!\rangle \sim g).(p,q) \qquad . \end{array}$$

□

This theorem shows the duality between $\langle\!\circ$ and $\circ\!\rangle$.

2.5 The connection with programs

Section 1.2 motivated this chapter. By identifying a program with its weakest precondition, the connection between programs and functions of two arguments is summarized as follows.

$$\begin{array}{rcll} \mathit{skip} &=& R & \\ \mathit{raise} &=& L & \\ S;T &=& S \circ\!\rangle\, T & \text{or more succinctly:} \quad ; \;=\; \circ\!\rangle \\ S \lhd T &=& S \langle\!\circ\, T & \text{or more succinctly:} \quad \lhd \;=\; \langle\!\circ \end{array}$$

I proceed to explain the connections between the other operators in the algebra and programs. Using (2.25) and (2.24), we have

$$\begin{aligned} S \langle\circ\rangle T &= (S \lhd \mathit{skip}); T \\ S \langle\circ\rangle T &= (S; \mathit{raise}) \lhd T \end{aligned} \qquad .$$

From the definitions of $\lfloor\ \rfloor$ and $\lceil\ \rceil$, and from (2.18) and (2.17), we have

$$\begin{aligned} \lfloor S \rfloor &= S \lhd \mathit{skip} \\ \lceil S \rceil &= S; \mathit{raise} \end{aligned} \qquad .$$

In words, the execution of $\lfloor S \rfloor$ is like that of S, except that when S terminates at all, $\lfloor S \rfloor$ terminates normally. Similarly, the execution of $\lceil S \rceil$ is like that of S, except that when S terminates at all, $\lceil S \rceil$ terminates exceptionally. The execution of $S \langle\circ\rangle T$ consists of the execution of S followed by, provided S terminates at all, the execution of T.

Transposition $\sim S$ is the statement that terminates just when S does, and upon termination complements the outcome. We can implement $\sim S$ as

$$[\![\, b \bullet ((S; b := \mathit{true}) \lhd b := \mathit{false}) \ ; \ \textbf{if}\ b \rightarrow \mathit{raise} \ \square\ \neg b \rightarrow \mathit{skip}\ \textbf{fi} \,]\!] \qquad .$$

As a program construct, $\sim$ seems to be of limited use, but maybe that's just our lack of imagination. In the algebra, however, it allows us to prove the duality between $\langle\circ$ and $\circ\rangle$.

Modula-3 is an example of a programming language with exceptions. In addition to the $\lhd$ construct, it has a so-called *try finally* statement. Execution of

TRY S FINALLY T END

consists of the execution of S followed by the execution of T. If the execution of S terminates exceptionally, then execution of T is followed by reraising the exception. This construct can be captured by

$$(S \lhd (T; \mathit{raise})); T \qquad .$$

Finally, I remark on the relation between the theory presented herein and existing programming languages. We find that usual programming languages introduce an asymmetry between left and right composition. For example, statements begin their execution in a normal state, $\lhd$ is often much longer to type than ;, and $\lhd$ may not be as efficient as ; (see, *e.g.*, [71]). However, the properties presented in this chapter suggest a more symmetric treatment of ; and $\lhd$.

2.6 Concluding remarks

We have seen that the algebra over functions of two arguments and their compositions —the algebra that underlies weakest preconditions for programs with exceptions— is simple and elegant. Consequently, there is good hope of getting a computer to perform calculations within this algebra, as is needed to do automatic verification of programs with exceptions.

A generalization of the present algebra to functions of type $D^n \to D$ for any n is found in [61]—in fact, the generalization does not even require that n be finite. [61] also presents a way, via *partitioned predicates* (see Remark 3.3), to handle exceptions without requiring $wp.S$ be a function from a *pair* of predicates to a predicate, but instead allowing it to remain a function from one predicate to another.

In previous studies of exceptions (and mathematics in general), this algebra has gone unnoticed. One reason for this has been suboptimal choices of notation for exceptional weakest preconditions. Where I write $wp.S.(P, Q)$, [60] writes $wp(S, Q, P)$ and [13] uses the two functions $wp(S, e, P)$ and $wp(S, ;, Q)$. My notation has the advantage over the others of permitting the separation of the function $wp.S$ from its argument (P, Q). Only then does the opportunity to discover the algebra over functions of type $D \times D \to D$ arise.

> **Remark 2.2.** Dijkstra came to the same conclusion regarding his non-exceptional wp, as is witnessed by contrasting the notation in [17] with that in [21].

The notation in [13] is readily extensible to more than one exception. However, to specify that a program does not raise any exceptions, one needs information about all declared exceptions, so the notation is not as extensible as it may first appear.

One of two other approaches to allowing more than one exception is to add a special variable which indicates which exception is raised. The variable would be updated immediate preceding a *raise* statement. The other possibility is to extend the pair to an arbitrary tuple. This is done in [4] and [61].

Application of the present algebra is not limited to the semantics of programs with exceptions—it can be used for any functions of type $D \times D \to D$. Another appearance of such functions in computer science is [22]. Also, [61] considers some other application areas, including relational databases and embedded systems.

Chapter 3

Trace semantics for exceptions

In Chapter 1, I introduced the weakest preconditions for exceptional program constructs. More precisely, I let the weakest preconditions (and weakest liberal preconditions) *define* these programs constructs. In Section 2.5, I described constructs like $\triangleleft$, $\lfloor\ \rfloor$, and *try finally* operationally. We may thus ask, "Does our operational interpretation of these constructs correspond to their mathematical weakest precondition definitions?". Not only is this issue relevant when using the constructs to write programs, but it becomes unavoidable when we try to implement the program constructs.

So, the question is, "Are these constructs the ones we think they are?". Focusing on assignment, the unit statements, and normal and exceptional composition, I lead us in this chapter to come to grips with the answer to that question.

3.0 Introduction

One approach to convincing ourselves that we have indeed defined the intended constructs is taken by Manasse and Nelson in [60], where exceptional programs are translated into simpler constructs. The implementation of these simpler constructs is more familiar to us, and is discussed in [21, Ch. 10]. In the present chapter, I take an approach different from that in [60]—in some sense, I approach the problem from the other end. To understand my approach, let's start by contrasting different semantics of a program.

3.0.0 Concreteness of a semantics

One reason the weakest-precondition semantics is so useful is that it provides a high-level view of programs—only initial and final states play a rôle.

In contrast, the advantage of a more *concrete* semantics is that it gets us closer to the implementation of the construct under consideration. On the other hand, a more concrete semantics suffers from being unwieldy to work with when proving programs.

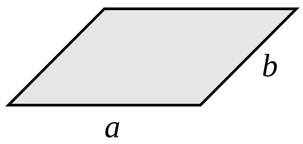

Figure 3.0: Normal state space

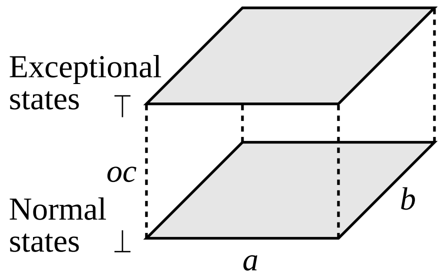

Figure 3.1: Exceptional state space

Since a goal in this thesis is to make the prevailing mathematics as simple as possible, a more concrete semantics does not appear desirable. However, by defining our program constructs more concretely, and then from that concrete definition *deriving* the weakest preconditions, we get the best of both worlds: We are better convinced that we are modeling the statements that we have in mind, and we get a calculus that is abstract enough to work with.

Notice that once we have established the connection between the more concrete semantics and the weakest-precondition semantics, we are no longer interested in the more concrete semantics.

3.0.1 Outline of chapter

As the more concrete semantics, I choose a *trace semantics* (*cf.* [75]). In the rest of this chapter, I take programs that can raise and handle exceptions along the path of Lukkien [59], which describes first an operational semantics in terms of traces and then derives a weakest-precondition semantics from it.

Section 3.1 describes my trace semantics model. In Section 3.2, I start off with a clean slate and define the basic exceptional statements and compositions by their trace semantics. Section 3.3 defines the meaning of weakest preconditions (wp) in the trace semantics setting. Section 3.4 calculates wp for the statements defined in Section 3.2. We can then compare the wp in this chapter with the wp from Chapter 1, and will find the two equal. Thus, the definitions in Chapter 1 *do* match our operational interpretation of the commands.

3.1 Trace sets

For a program without exceptions, the program state space is the Cartesian product of the program variables. For example, the state space of a program with two program variables, a and b say, can be thought of as a two-dimensional space, as depicted in Figure 3.0.

To model exceptional programs, I augment the state space with a binary "*outcome*" coordinate, oc, depicted for the two-variable example in Figure 3.1. Coordinate oc partitions the resulting state space into *normal* ($oc = \bot$) and *exceptional* ($oc = \top$)

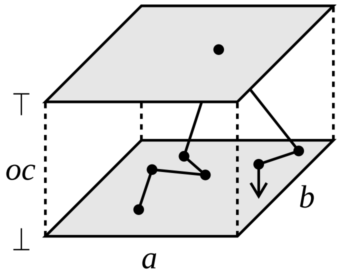

Figure 3.2: Example trace

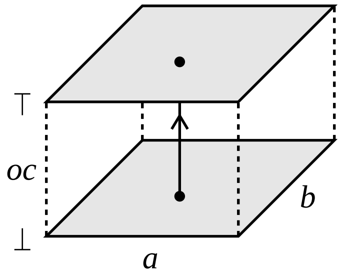

Figure 3.3: Example trace of *raise*

states. For state x, I write $nor.x$ to indicate that the outcome is normal, and $exc.x$ to indicate that the outcome is exceptional. I write X for the set of all states, including those with an exceptional outcome.

The semantics of a program is defined via traces. In this chapter, a trace is a nonempty sequence of states that starts in a normal state; no actions are recorded in the traces. A trace set is a (possibly infinite) set of (possibly infinite) traces. For program S, I identify S with the set of all traces that can be the result of executing S. Figure 3.2 depicts a sample trace.

Catenation, which binds stronger than function application, will be denoted by juxtaposition. Variables s and t range over (possibly empty) sequences of states, and x and y over states. I define $fin.s$ to hold just when the length of s is finite, and $inf.s$ to hold otherwise. For nonempty sequence s, I let $first.s$ denote the first state in s, and, if s is finite, $last.s$ the last state in s. For state x, I write $\lfloor x \rfloor$ for $x[oc := \bot]$ and $\lceil x \rceil$ for $x[oc := \top]$, that is, x in which the value of coordinate oc has been replaced by $\bot$ and $\top$, respectively. Stated differently, $\lfloor x \rfloor$ and $\lceil x \rceil$ are the two states that are the projections of x onto the normal and exceptional planes, respectively.

> **Remark 3.0.** The given trace semantics models programs with one exception. To model n distinct exceptions directly in the trace semantics, one can change oc from being a binary-valued coordinate to a coordinate that can assume $1+n$ values: the normal value plus one for each exception.

3.2 Program constructs as trace sets

In this section, I define each program construct as a trace set.

3.2.0 Primitive statements

By way of introduction, I define the trace semantics of the assignment statement. Let v be a regular program variable and E be a total expression (see Section 1.1). Then, in the absence of exceptions, one would write

$$v := E \;=\; \{\, x \triangleright x\ (x[v := E.x])\ \} \qquad ,$$

in which every trace has length two: it consists of initial state x and final state $x[v := E.x]$, that is, state x in which the value of coordinate v has been replaced by the value of expression E evaluated in state x. The set $v := E$ contains such a trace for every state $x \in X$. In the presence of exceptions, I restrict x to be a normal state and write

$$v := E \;=\; \{\, x \mid nor.x \,\triangleright\, x\,(x[v := E.x]) \,\} \qquad . \tag{3.0}$$

Statement *skip* is defined as

$$skip \;=\; \{\, x \mid nor.x \,\triangleright\, x \,\} \tag{3.1}$$

in which the latter occurrence of x denotes a trace of length one. I could have chosen

$$skip \;=\; \{\, x \mid nor.x \,\triangleright\, x\,x \,\}$$

to get traces of length two, but, for reasons discussed below, I prefer (3.1).

I write the raising of an exception as the statement *raise*, and define its trace semantics as

$$raise \;=\; \{\, x \mid nor.x \,\triangleright\, x\lceil x \rceil \,\} \qquad , \tag{3.2}$$

that is, the set of all traces of length two starting with a normal state and ending with an exceptional state; the two states are equal in all other coordinates. Figure 3.3 shows a sample trace from set *raise*. Alternatively, I might have written

$$raise \;=\; oc := \top \qquad ,$$

except that *oc* is not a regular program variable; it is a variable that I have introduced for describing the trace semantics only.

3.2.1 Normal composition

The definition of sequential composition is changed to accommodate exceptional outcomes. If there were no exceptions, one could define

$$\begin{aligned} S;T \;=\; & \{\, s,x,t \mid sx \in S \wedge xt \in T \wedge fin.s \,\triangleright\, sxt \,\} \;\cup \\ & \{\, s \mid s \in S \wedge inf.s \,\triangleright\, s \,\} \qquad , \end{aligned}$$

which distinguishes between those traces in which execution of S does or does not terminate. In words, $S;T$ contain the set of traces that start with a finite trace from S (sx) and continue with a trace from T (xt), where the last state in the trace from S (x) is the same as the first state of T (x). $S;T$ also contains the infinite traces in S.

In the presence of exceptions, I refine the definition to

$$\begin{aligned} S;T \;=\; & \{\, s,x,t \mid sx \in S \wedge xt \in T \wedge fin.s \wedge nor.x \,\triangleright\, sxt \,\} \;\cup \\ & \{\, s \mid s \in S \wedge (inf.s \vee exc.(last.s)) \,\triangleright\, s \,\} \qquad . \end{aligned} \tag{3.3}$$

What distinguishes this definition from the previous one is that the connecting state (x) is restricted to be a normal state. The infinite traces of S and the finite traces of S that end with an exceptional state are not joined with any trace from T. This definition captures the fact that execution of $S;T$ reduces to execution of S in the case where that execution terminates exceptionally.

We have

Theorem

; is associative. (3.4)

I omit the proof of this theorem.

Theorem

skip is the left identity of ;. (3.5)

Proof.

$$
\begin{array}{ll}
 & skip; T \\
= & \quad \{\ (3.3)\text{: def. of } ;\ \} \\
 & \{ s,x,t \mid sx \in skip \ \wedge\ xt \in T \ \wedge\ fin.s \ \wedge\ nor.x \ \triangleright\ sxt \} \cup \\
 & \{ s \mid s \in skip \ \wedge\ (inf.s \ \vee\ exc.(last.s)) \ \triangleright\ s \} \\
= & \quad \{\ (3.1)\text{: def. of } skip\ \} \\
 & \{ x,t \mid xt \in T \ \wedge\ nor.x \ \triangleright\ xt \} \\
= & \quad \{\ \text{each trace starts in a normal state}\ \} \\
 & T
\end{array}
$$

□

Theorem

skip is the right identity of ;. (3.6)

Proof.

$$
\begin{array}{ll}
 & S; skip \\
= & \quad \{\ (3.3)\text{: def. of } ;\ \} \\
 & \{ s,x,t \mid sx \in S \ \wedge\ xt \in skip \ \wedge\ fin.s \ \wedge\ nor.x \ \triangleright\ sxt \} \cup \\
 & \{ s \mid s \in S \ \wedge\ (inf.s \ \vee\ exc.(last.s)) \ \triangleright\ s \} \\
= & \quad \{\ (3.1)\text{: def. of } skip\ \} \\
 & \{ s,x \mid sx \in S \ \wedge\ fin.s \ \wedge\ nor.x \ \triangleright\ sx \} \cup \\
 & \{ s \mid s \in S \ \wedge\ (inf.s \ \vee\ exc.(last.s)) \ \triangleright\ s \} \\
= & \quad \{\ fin \text{ and } inf \text{ are each other's complements, and ditto for } nor \text{ and } exc\ \} \\
 & \{ s \mid s \in S \ \triangleright\ s \} \\
= & \\
 & S
\end{array}
$$

□

For the above two theorems to hold, it is essential that *skip* does not duplicate the state in a trace when joined by a semicolon with another statement. This is why the trace set of *skip* contains traces of length one instead of traces of length two.

In the next theorem, I consider ; applied to arbitrary trace sets.

Theorem

; is positively $\cup$-distributive in both arguments. (3.7)

> **Remark 3.1.** In its left argument, ; is even *universally* $\cup$-distributive, as the proof shows, but that property is not needed in the present discussion.

Proof. With S ranging over any bag of trace sets, we calculate,

$$
\begin{array}{ll}
& \langle \cup S \triangleright S \rangle ; T \\
= & \quad \{ \ (3.3)\text{: def. of } ; \ \} \\
& \{ s,x,t \mid sx \in \langle \cup S \triangleright S \rangle \wedge xt \in T \wedge \mathit{fin}.s \wedge \mathit{nor}.s \triangleright sxt \} \cup \\
& \{ s \mid s \in \langle \cup S \triangleright S \rangle \wedge (\mathit{inf}.s \vee \mathit{exc}.(\mathit{last}.s)) \triangleright s \} \\
= & \quad \{ \ \text{interchange unions} \ \} \\
& \{ S,s,x,t \mid sx \in S \wedge xt \in T \wedge \mathit{fin}.s \wedge \mathit{nor}.s \triangleright sxt \} \cup \\
& \{ S,s \mid s \in S \wedge (\mathit{inf}.s \vee \mathit{exc}.(\mathit{last}.s)) \triangleright s \} \\
= & \quad \{ \ \text{nesting} \ \} \\
& \langle \cup S \triangleright \{ s,x,t \mid sx \in S \wedge xt \in T \wedge \mathit{fin}.s \wedge \mathit{nor}.s \triangleright sxt \} \rangle \cup \\
& \langle \cup S \triangleright \{ s \mid s \in S \wedge (\mathit{inf}.s \vee \mathit{exc}.(\mathit{last}.s)) \triangleright s \} \rangle \\
= & \quad \{ \ \text{combine terms} \ \} \\
& \langle \cup S \triangleright \ \{ s,x,t \mid sx \in S \wedge xt \in T \wedge \mathit{fin}.s \wedge \mathit{nor}.s \triangleright sxt \} \cup \\
& \qquad\quad \{ s \mid s \in S \wedge (\mathit{inf}.s \vee \mathit{exc}.(\mathit{last}.s)) \triangleright s \} \rangle \\
= & \quad \{ \ (3.3)\text{: def. of } ; \ \} \\
& \langle \cup S \triangleright S;T \rangle \qquad .
\end{array}
$$

For the other argument, and with T ranging over any nonempty bag of trace sets, we calculate,

$$
\begin{array}{ll}
& S; \langle \cup T \triangleright T \rangle \\
= & \quad \{ \ (3.3)\text{: def. of } ; \ \} \\
& \{ s,x,t \mid sx \in S \wedge xt \in \langle \cup T \triangleright T \rangle \wedge \mathit{fin}.s \wedge \mathit{nor}.s \triangleright sxt \} \cup \\
& \{ s \mid s \in S \wedge (\mathit{inf}.s \vee \mathit{exc}.(\mathit{last}.s)) \triangleright s \} \\
= & \quad \{ \ \text{interchange union} \ \} \\
& \{ T,s,x,t \mid sx \in S \wedge xt \in T \wedge \mathit{fin}.s \wedge \mathit{nor}.s \triangleright sxt \} \cup \\
& \{ s \mid s \in S \wedge (\mathit{inf}.s \vee \mathit{exc}.(\mathit{last}.s)) \triangleright s \} \\
= & \quad \{ \ \text{nesting} \ \} \\
& \langle \cup T \triangleright \{ s,x,t \mid sx \in S \wedge xt \in T \wedge \mathit{fin}.s \wedge \mathit{nor}.s \triangleright sxt \} \rangle \cup \\
& \{ s \mid s \in S \wedge (\mathit{inf}.s \vee \mathit{exc}.(\mathit{last}.s)) \triangleright s \} \\
= & \quad \{ \ \text{range of } T \text{ is nonempty} \ \}
\end{array}
$$

$$
\begin{array}{ll}
& \langle \cup T \triangleright \ \{ s,x,t \mid sx \in S \ \wedge \ xt \in T \ \wedge \ \mathit{fin}.s \ \wedge \ \mathit{nor}.s \ \triangleright \ sxt \} \ \cup \\
& \qquad \{ s \mid s \in S \ \wedge \ (\mathit{inf}.s \ \vee \ \mathit{exc}.(\mathit{last}.s)) \ \triangleright \ s \} \rangle \\
= & \quad \{ \ (3.3)\text{: def. of } ; \ \} \\
& \langle \cup T \ \triangleright \ S;T \ \rangle \qquad .
\end{array}
$$

□

3.2.2 Exceptional composition

Next, I define the trace semantics of the exception handler.

$$
\begin{array}{rl}
S \lhd T = & \{ s,x,t \mid sx \in S \wedge \lfloor x \rfloor t \in T \wedge \mathit{fin}.s \wedge \mathit{exc}.x \ \triangleright \ sx \lfloor x \rfloor t \ \} \ \cup \\
& \{ s \mid s \in S \wedge (\mathit{inf}.s \vee \mathit{nor}.(\mathit{last}.s)) \ \triangleright \ s \ \}
\end{array} \tag{3.8}
$$

This set is similar to $S;T$, but has some important differences. Finite traces of S where the last state is exceptional (sx) are joined with traces from T that begin with the normal projection ($\lfloor x \rfloor$) of the last state of the trace from S. Moreover, unlike normal composition, no state is dropped here, so both the last state in the trace from S (x) and the first state in the trace from T ($\lfloor x \rfloor$) appear in the traces in $S \lhd T$. Also, infinite traces of S and finite traces of S that end with a *normal* state are included in $S \lhd T$, but are not joined with any trace from T.

> **Remark 3.2.** It would be nice to have
>
> raise is the left and right identity of $\lhd$,
>
> but neither part of this property holds, because the traces of raise have length two, and therefore add an extra state to the traces of the exception handler. Furthermore, the exception handler, too, repeats the connecting state, x, with $\mathit{oc} := \bot$ in the second occurrence.

3.2.3 Other statements

I omit discussion of the remaining statements; aside from differences noted in this section, their definitions are as in [59].

Compared to [59], the definitions of *abort* and of the if-statement need not be changed because the initial state is always normal. The definition of the do-statement need not be changed because it is defined in terms of the if-statement, sequential composition, and *skip*. The latter two have already been redefined to cater for exceptional states, and the only properties used in the context of the do-statement are that sequential composition is associative (which it still is—see (3.4)), that *skip* is its identity element (which it still is—see (3.5) and (3.6)), and that sequential composition is positively $\cup$-distributive in both arguments (which it still is—see (3.7)). For the purpose of defining the do-statement, [59] introduces a partial command $g?$, which can be seen as the command $g \rightarrow \mathit{skip}$. However, this command is not recognized as a partial command in [59].

3.3 Weakest preconditions of trace sets

I define function $wp.S.(P,Q)$ to be the weakest condition on the initial state such that: execution of program S terminates, every exceptional terminating state satisfies P, and every normal terminating state satisfies Q. Since the oc coordinate is not part of the program but of the trace semantics only, I require that P, Q, and $wp.S.(P,Q)$ be independent of the oc coordinate, that is, $Q.x = Q.\lfloor x \rfloor = Q.\lceil x \rceil$. I do so by restricting P and Q to predicates in which oc does not occur, and by designing wp carefully (see (3.11) below). As a result, $wp.S.(\mathit{false},Q)$ in this chapter coincides with Dijkstra's $wp.S.Q$ [17].

In the sequel, I often need to distinguish between conditions on the exceptional and on the normal states. I write a pair of conditions to capture this distinction.

$$(P,Q).x \quad = \quad (exc.x \Rightarrow P.x) \wedge (nor.x \Rightarrow Q.x) \tag{3.9}$$

> **Remark 3.3.** In [61], the generalization of such a pair to any tuple is called a *partition predicate.*

This construction is universally conjunctive, that is,

$$\langle \forall i \triangleright (P_i, Q_i) \rangle = (\langle \forall i \triangleright P_i \rangle, \langle \forall i \triangleright Q_i \rangle) \quad , \tag{3.10}$$

as is shown by the calculation

$$\begin{array}{ll}
 & \langle \forall i \triangleright (P_i, Q_i) \rangle.x \\
= & \qquad \{ \text{ lifting } \} \\
 & \langle \forall i \triangleright (P_i, Q_i).x \rangle \\
= & \qquad \{ \text{ (3.9): def. of a pair predicate } \} \\
 & \langle \forall i \triangleright (exc.x \Rightarrow P_i.x) \wedge (nor.x \Rightarrow Q_i.x) \rangle \\
= & \qquad \{ \text{ pred. calc. } \} \\
 & (exc.x \Rightarrow \langle \forall i \triangleright P_i.x \rangle) \wedge (nor.x \Rightarrow \langle \forall i \triangleright Q_i.x \rangle) \\
= & \qquad \{ \text{ lifting } \} \\
 & (exc.x \Rightarrow \langle \forall i \triangleright P_i \rangle.x) \wedge (nor.x \Rightarrow \langle \forall i \triangleright Q_i \rangle.x) \\
= & \qquad \{ \text{ (3.9): def. of a pair predicate } \} \\
 & (\langle \forall i \triangleright P_i \rangle, \langle \forall i \triangleright Q_i \rangle).x \qquad .
\end{array}$$

The definition of $wp.S.(P,Q)$ for state x is a condition on x such that every trace t of S that begins with initial state $\lfloor x \rfloor$ is of finite length and satisfies $(P,Q).(last.t)$.

$$wp.S.(P,Q).x \quad = \quad \langle \forall t \mid first.t = \lfloor x \rfloor \wedge t \in S \triangleright fin.t \wedge (P,Q).(last.t) \rangle \tag{3.11}$$

Above, I said that this definition needed to be designed with care. The trick is to define $wp.S.(P,Q).x$ for *any* x, not just for normal states x, despite the fact that a trace always begins with a normal state. Since x can be any state, $\lfloor x \rfloor$ is needed in the definition (see, *e.g.*, the second step in the calculation leading to (3.18)).

I continue with some theorems regarding wp.

Theorem

$$wp.S \text{ is positively conjunctive.} \tag{3.12}$$

Proof. With K ranging over any nonempty bag of pairs, we calculate,

$$
\begin{array}{ll}
 & wp.S.\langle \forall K \triangleright K \rangle.x \\
= & \quad \{ \ (3.11)\text{: def. of } wp \ \} \\
 & \langle \forall t \mid first.t = \lfloor x \rfloor \ \wedge \ t \in S \triangleright fin.t \ \wedge \ \langle \forall K \triangleright K \rangle.(last.t) \rangle \\
= & \quad \{ \ \text{lifting} \ \} \\
 & \langle \forall t \mid first.t = \lfloor x \rfloor \ \wedge \ t \in S \triangleright fin.t \ \wedge \ \langle \forall K \triangleright K.(last.t) \rangle \rangle \\
= & \quad \{ \ \wedge \ \text{ over } \forall\text{, since range is nonempty} \ \} \\
 & \langle \forall t \mid first.t = \lfloor x \rfloor \ \wedge \ t \in S \triangleright \langle \forall K \triangleright fin.t \ \wedge \ K.(last.t) \rangle \rangle \\
= & \quad \{ \ \text{interchange of quantifications} \ \} \\
 & \langle \forall K \triangleright \langle \forall t \mid first.t = \lfloor x \rfloor \ \wedge \ t \in S \triangleright fin.t \ \wedge \ K.(last.t) \rangle \rangle \\
= & \quad \{ \ (3.11)\text{: def. of } wp \ \} \\
 & \langle \forall K \triangleright wp.S.K.x \rangle \\
= & \quad \{ \ \text{lifting} \ \} \\
 & \langle \forall K \triangleright wp.S.K \rangle.x \quad .
\end{array}
$$

□

A consequence of this theorem (*cf.* [21]) is

$$wp.S \text{ is monotonic.} \tag{3.13}$$

Theorem

$$wp.S.(P, Q) \ = \ wp.S.(P, true) \wedge wp.S.(true, Q) \tag{3.14}$$

Proof. Follows from (3.10) and (3.12). □

Using the "everywhere" operator, written $[\,]$ (*cf.* [21]), I can state the next theorems.

Theorem

$$[R \Rightarrow wp.S.(P, Q)] \ = \ [R \Rightarrow wp.S.(P, true)] \wedge [R \Rightarrow wp.S.(true, Q)] \tag{3.15}$$

Proof. Follows from (3.14) and predicate calculus. □

Theorem

$$[R \Rightarrow wp.S.(P, Q)] \ = \ \langle \exists M \triangleright [R \Rightarrow wp.S.(M, Q)] \wedge [M \Rightarrow P] \rangle \tag{3.16}$$

$$[R \Rightarrow wp.S.(P, Q)] \ = \ \langle \exists M \triangleright [R \Rightarrow wp.S.(P, M)] \wedge [M \Rightarrow Q] \rangle \tag{3.17}$$

Proof.

$$
\begin{array}{ll}
 & [R \;\Rightarrow\; wp.S.(P,Q)] \\
= & \quad \{\ \text{one-point rule}\ \} \\
 & \langle \exists M \mid [M \;\equiv\; P] \triangleright [R \;\Rightarrow\; wp.S.(M,Q)]\ \rangle \\
\Rightarrow & \quad \{\ \text{weakening}\ \} \\
 & \langle \exists M \triangleright [M \;\Rightarrow\; P] \;\wedge\; [R \;\Rightarrow\; wp.S.(M,Q)]\ \rangle \\
\Rightarrow & \quad \{\ (3.13)\text{: } wp.S \text{ is monotonic}\ \} \\
 & \langle \exists M \triangleright [R \;\Rightarrow\; wp.S.(P,Q)]\ \rangle \\
= & \quad \{\ \text{range is nonempty}\ \} \\
 & [R \;\Rightarrow\; wp.S.(P,Q)]
\end{array}
$$

The proof of the other is similar. □

3.4 Calculating the weakest preconditions

I calculate wp for the various program constructs, arriving at a link with the wp definitions of Chapter 1. First, I look at $skip$. For any state x,

$$
\begin{array}{ll}
 & wp.skip.(P,Q).x \\
= & \quad \{\ \ (3.11)\text{: def. of } wp\ \ \} \\
 & \langle \forall t \mid first.t = \lfloor x \rfloor \;\wedge\; t \in skip \triangleright fin.t \;\wedge\; (P,Q).(last.t)\ \rangle \\
= & \quad \{\ \ (3.1)\text{: def. of } skip\ \ \} \\
 & (P,Q).\lfloor x \rfloor \\
= & \quad \{\ \ nor.\lfloor x \rfloor \text{, and (3.9): def. of } (P,Q)\ \ \} \\
 & Q.\lfloor x \rfloor \\
= & \quad \{\ \ Q \text{ is independent of } oc\ \ \} \\
 & Q.x \qquad ,
\end{array}
$$

and hence

$$
wp.skip.(P,Q) \;=\; Q \qquad . \tag{3.18}
$$

By a similar calculation, we obtain, for any state x,

$$
\begin{array}{ll}
 & wp.raise.(P,Q).x \\
= & \quad \{\ \ (3.11)\text{: def. of } wp\ \ \} \\
 & \langle \forall t \mid first.t = \lfloor x \rfloor \;\wedge\; t \in raise \triangleright fin.t \;\wedge\; (P,Q).(last.t)\ \rangle \\
= & \quad \{\ \ (3.2)\text{: def. of } raise\ \ \} \\
 & (P,Q).\lceil \lfloor x \rfloor \rceil \\
= & \quad \{\ \ \lceil \lfloor x \rfloor \rceil = \lceil x \rceil \text{, and } exc.\lceil x \rceil \text{, and (3.9): def. of } (P,Q)\ \ \} \\
 & P.\lceil x \rceil \\
= & \quad \{\ \ P \text{ is independent of } oc\ \ \} \\
 & P.x \qquad ,
\end{array}
$$

and hence

$$wp.raise.(P,Q) \;=\; P \qquad . \tag{3.19}$$

Similarly,

$$wp.(v := E).(P,Q) \;=\; Q[v := E] \tag{3.20}$$

can be shown.

More involved are the calculations for sequential and exceptional composition. For the latter, we calculate,

$$
\begin{array}{ll}
 & wp.(S \lhd T).(P,Q).x \\
= & \qquad \{ \quad (3.11)\text{: def. of } wp \quad \} \\
 & \langle \forall t \mid first.t = \lfloor x \rfloor \;\wedge\; t \in S \lhd T \;\triangleright\; fin.t \;\wedge\; (P,Q).(last.t) \,\rangle \\
= & \qquad \{ \quad (3.8)\text{: def. of } \lhd \text{, with } t := sy\lfloor y \rfloor t \text{ and } t := s \quad \} \\
 & \langle \forall s,y,t \mid \; first.sy\lfloor y \rfloor t = \lfloor x \rfloor \;\wedge\; sy \in S \;\wedge\; \lfloor y \rfloor t \in T \;\wedge\; fin.s \;\wedge\; exc.y \triangleright \\
 & \qquad\qquad fin.sy\lfloor y \rfloor t \;\wedge\; (P.Q).(last.sy\lfloor y \rfloor t) \,\rangle \;\wedge \\
 & \langle \forall s \mid \; first.s = \lfloor x \rfloor \;\wedge\; s \in S \;\wedge\; (inf.s \;\vee\; nor.(last.s)) \;\triangleright\; fin.s \;\wedge\; (P,Q).(last.s) \,\rangle \\
= & \qquad \{ \quad first.sy\lfloor y \rfloor t = first.sy \text{, and } fin.s = fin.sy \text{, and } last.sy\lfloor y \rfloor t = last.\lfloor y \rfloor t \quad \} \\
 & \langle \forall s,y,t \mid \; first.sy = \lfloor x \rfloor \;\wedge\; sy \in S \;\wedge\; \lfloor y \rfloor t \in T \;\wedge\; fin.sy \;\wedge\; exc.y \triangleright \\
 & \qquad\qquad fin.sy\lfloor y \rfloor t \;\wedge\; (P.Q).(last.\lfloor y \rfloor t) \,\rangle \;\wedge \\
 & \langle \forall s \mid \; first.s = \lfloor x \rfloor \;\wedge\; s \in S \;\wedge\; (inf.s \;\vee\; nor.(last.s)) \;\triangleright\; fin.s \;\wedge\; (P,Q).(last.s) \,\rangle \\
= & \qquad \{ \quad \text{rename } s \text{ and } t \text{ as } sy := s \text{ and } \lfloor y \rfloor t := t \text{ in first quantification} \quad \} \\
 & \langle \forall s,t \mid \; first.s = \lfloor x \rfloor \;\wedge\; s \in S \;\wedge\; first.t = \lfloor last.s \rfloor \;\wedge\; t \in T \;\wedge\; fin.s \;\wedge\; exc.(last.s) \triangleright \\
 & \qquad\quad fin.st \;\wedge\; (P.Q).(last.t) \,\rangle \;\wedge \\
 & \langle \forall s \mid \; first.s = \lfloor x \rfloor \;\wedge\; s \in S \;\wedge\; (inf.s \;\vee\; nor.(last.s)) \;\triangleright\; fin.s \;\wedge\; (P,Q).(last.s) \,\rangle \\
= & \qquad \{ \quad \text{nesting, and } fin.s \;\Rightarrow\; (fin.st = fin.t) \quad \} \\
 & \langle \forall s \mid \; first.s = \lfloor x \rfloor \;\wedge\; s \in S \;\wedge\; fin.s \;\wedge\; exc.(last.s) \triangleright \\
 & \qquad \langle \forall t \mid first.t = \lfloor last.s \rfloor \;\wedge\; t \in T \;\triangleright\; fin.t \;\wedge\; (P.Q).(last.t) \,\rangle\rangle \;\wedge \\
 & \langle \forall s \mid \; first.s = \lfloor x \rfloor \;\wedge\; s \in S \;\wedge\; (inf.s \;\vee\; nor.(last.s)) \;\triangleright\; fin.s \;\wedge\; (P,Q).(last.s) \,\rangle \\
= & \qquad \{ \quad \text{shunting twice, and (3.11): def. of } wp \quad \} \\
 & \langle \forall s \mid \; first.s = \lfloor x \rfloor \;\wedge\; s \in S \;\triangleright\; inf.s \;\vee\; nor.(last.s) \;\vee\; wp.T.(P,Q).(last.s) \,\rangle \;\wedge \\
 & \langle \forall s \mid \; first.s = \lfloor x \rfloor \;\wedge\; s \in S \;\triangleright\; (fin.s \;\wedge\; exc.(last.s)) \;\vee\; (fin.s \;\wedge\; (P,Q).(last.s)) \,\rangle
\end{array}
$$

Leaving the range $first.s = \lfloor x \rfloor \;\wedge\; s \in S$ as understood, we continue the calculation,

$$
\begin{array}{ll}
 & \langle \forall s \;\triangleright\; inf.s \;\vee\; nor.(last.s) \;\vee\; wp.T.(P,Q).(last.s) \,\rangle \;\wedge \\
 & \langle \forall s \;\triangleright\; (fin.s \;\wedge\; exc.(last.s)) \;\vee\; (fin.s \;\wedge\; (P,Q).(last.s)) \,\rangle \\
= & \qquad \{ \quad \text{combine terms, and factor} \quad \} \\
 & \langle \forall s \triangleright \; (inf.s \;\vee\; nor.(last.s) \;\vee\; wp.T.(P,Q).(last.s)) \;\wedge \\
 & \qquad fin.s \;\wedge\; (exc.(last.s) \;\vee\; (P,Q).(last.s)) \,\rangle \\
= & \qquad \{ \quad \text{absorption, and pred. calc.} \quad \} \\
 & \langle \forall s \;\triangleright\; fin.s \;\wedge\; \; (exc.(last.s) \;\Rightarrow\; wp.T.(P,Q).(last.s)) \;\wedge \\
 & \qquad\qquad\quad (nor.(last.s) \;\Rightarrow\; (P,Q).(last.s)) \,\rangle \\
= & \qquad \{ \quad nor.(last.s) \;\Rightarrow\; ((P,Q).(last.s) = Q.(last.s)) \quad \}
\end{array}
$$

$$\langle \forall s \triangleright \mathit{fin}.s \wedge (\mathit{exc}.(\mathit{last}.s) \Rightarrow \mathit{wp}.T.(P,Q).(\mathit{last}.s)) \wedge (\mathit{nor}.(\mathit{last}.s) \Rightarrow Q.(\mathit{last}.s)) \rangle$$

$=$ { (3.9): def. of a pair }

$$\langle \forall s \triangleright \mathit{fin}.s \wedge (\mathit{wp}.T.(P,Q),Q).(\mathit{last}.s) \rangle$$

$=$ { (3.11): def. of wp (recall the understood range) }

$$\mathit{wp}.S.(\mathit{wp}.T.(P,Q),Q).x \quad ,$$

and obtain

$$\mathit{wp}.(S \lhd T).(P,Q) = \mathit{wp}.S.(\mathit{wp}.T.(P,Q),Q) \quad . \tag{3.21}$$

For sequential composition, we can show

$$\mathit{wp}.(S;T).(P,Q) = \mathit{wp}.S.(P,\mathit{wp}.T.(P,Q)) \quad , \tag{3.22}$$

using a calculation very similar to that of exceptional composition, if slightly easier.

Comparing the present wp of the assignment statement (3.20), *skip* (3.18), *raise* (3.19), ; (3.22), $\lhd$ (3.21) with the Chapter 1 wp definitions of these statements (1.0, 1.1, 1.2) and statement compositions (1.3, 1.4), we conclude the two wp's to be equal. Hence, I have justified the definitions of wp for these constructs.

Remark 3.4. With reference to Remark 3.2, note that in the weakest-precondition semantics, *raise* *is* an identity of $\lhd$. We saw this not to be the case in the trace semantics because of repeated states. This comes back to the fact that the trace semantics is more concrete than the wp semantics, which only concerns itself with the first and last states of traces.

Chapter 4

A theorem on programming methodology

From the weakest-precondition semantics given for a programming notation, one often derives some theorems that are used in reasoning about programs. Ideally, they suggest hints for methodical program construction. An example of that is the Invariance Theorem for iterative statements (*cf.* [21] and Section 1.6). I give a theorem that suggests using exception handlers in a way similar to split binary semaphores (*cf.* [63]). The theorem is in terms of Hoare triples for normal termination, and free occurrences of *raise* for exceptional termination.

4.0 Hoare triples

Inspired by [36], I define the Hoare triple of exceptional programs by

$$\{R\}\, S\, \{Q\} \;=\; [R \Rightarrow wp.S.(true, Q)] \qquad , \tag{4.0}$$

from which the Hoare triple for each of the basic statements can be calculated.

$$\begin{array}{lcl}
\{R\}\, v := E\, \{Q\} & = & [R \Rightarrow Q[v := E]] \\
\{R\}\, skip\, \{Q\} & = & [R \Rightarrow Q] \\
\{R\}\, raise\, \{Q\} & = & true \\
\{R\}\, S0; S1\, \{Q\} & = & \langle\, \exists M \triangleright\ \{R\}\, S0\, \{M\} \wedge \{M\}\, S1\, \{Q\}\, \rangle \qquad (4.1) \\
\{R\}\, S0 \lhd S1\, \{Q\} & = & \{R\}\, S0\, \{Q\} \wedge \\
\quad \langle\, \exists M \triangleright\ [R \Rightarrow wp.S0.(M, true)] \wedge \{M\}\, S1\, \{Q\}\, \rangle & & \qquad (4.2)
\end{array}$$

I prove the last two of these. For sequential composition,

$$\begin{array}{ll}
 & \{R\}\, S0; S1\, \{Q\} \\
= & \quad \{\ (4.0)\text{: def. of triple}\ \} \\
 & [R \Rightarrow wp.(S0; S1).(true, Q)] \\
= & \quad \{\ (1.3)\text{: } wp \text{ of } ;\ \} \\
 & [R \Rightarrow wp.S0.(true, wp.S1.(true, Q))] \\
= & \quad \{\ (3.17) \text{ with } S, P, Q := S0, true, wp.S1.(true, Q)\ \}
\end{array}$$

$$
\begin{array}{ll}
 & \langle \exists M \triangleright [R \Rightarrow wp.S0.(true, M)] \wedge [M \Rightarrow wp.S1.(true, Q)] \rangle \\
= & \qquad \{ \text{ (4.0): def. of triple, twice } \} \\
 & \langle \exists M \triangleright \{R\}\, S0\, \{M\} \wedge \{M\}\, S1\, \{Q\} \rangle \qquad ,
\end{array}
$$

and for exceptional composition,

$$
\begin{array}{ll}
 & \{R\}\, S0 \lhd S1\, \{Q\} \\
= & \qquad \{ \text{ (4.0): def. of triple } \} \\
 & [R \Rightarrow wp.(S0 \lhd S1).(true, Q)] \\
= & \qquad \{ \text{ (1.4): } wp \text{ of } \lhd \ \} \\
 & [R \Rightarrow wp.S0.(wp.S1.(true, Q), Q)] \\
= & \qquad \{ \text{ (3.16) with } S, P := S0, wp.S1.(true, Q) \ \} \\
 & \langle \exists M \triangleright [R \Rightarrow wp.S0.(M, Q)] \wedge [M \Rightarrow wp.S1.(true, Q)] \rangle \\
= & \qquad \{ \text{ (3.15) with } P := M \ \} \\
 & \langle \exists M \triangleright [R \Rightarrow wp.S0.(M, true)] \wedge [R \Rightarrow wp.S0.(true, Q)] \wedge [M \Rightarrow wp.S1.(true, Q)] \rangle \\
= & \qquad \{ \text{ (4.0): def. of triple, twice } \} \\
 & \langle \exists M \triangleright [R \Rightarrow wp.S0.(M, true)] \wedge \{R\}\, S0\, \{Q\} \wedge \{M\}\, S1\, \{Q\} \rangle \\
= & \qquad \{ \ \wedge \text{ distributes over } \exists \ \} \\
 & \{R\}\, S0\, \{Q\} \wedge \langle \exists M \triangleright [R \Rightarrow wp.S0.(M, true)] \wedge \{M\}\, S1\, \{Q\} \rangle \qquad .
\end{array}
$$

4.1 Free occurrences of *raise*

I now give the definition of *free occurrence of raise*. Statement *raise* occurs free in

- *raise*
- $S0; S1$ just when it occurs free in $S0$ or in $S1$
- $S0 \lhd S1$ just when it occurs free in $S1$
- the other statements when it occurs free in one or more of their constituent statements.

If every execution of a free occurrence of *raise* in a statement S starts in a state satisfying a predicate P whenever the execution of S starts in a state satisfying the predicate R, then I say "every free occurrence of *raise* in S has precondition P in context R". The next theorem establishes the correspondence between this informal statement and a mathematical formula.

Theorem

$$
\begin{array}{ll}
 & \text{Every free (occurrence of) } raise \text{ in } S \text{ has precondition } P \text{ in context } R \\
= & \\
 & [R \Rightarrow wp.S.(P, true)]
\end{array}
\qquad (4.3)
$$

Proof. The proof is by induction over the syntax of S. For assignment, we calculate,

$$
\begin{array}{ll}
 & \text{every free } \mathit{raise} \text{ in } v := E \text{ has precondition } P \text{ in context } R \\
= & \quad \{ \text{ there are no free occurrences of } \mathit{raise} \text{ in } v := E \ \} \\
 & \mathit{true} \\
= & \quad \{ \ (1.0)\text{: } \mathit{wp} \text{ of } v := E \text{; pred. calc. } \} \\
 & [R \Rightarrow \mathit{wp}.(v := E).(P, \mathit{true})] \qquad .
\end{array}
$$

The proof for skip is similar. For raise,

$$
\begin{array}{ll}
 & \text{every free } \mathit{raise} \text{ in } \mathit{raise} \text{ has precondition } P \text{ in context } R \\
= & \quad \{ \ \mathit{raise} \text{ is a free occurrence of } \mathit{raise} \ \} \\
 & [R \Rightarrow P] \\
= & \quad \{ \ (1.2)\text{: } \mathit{wp} \text{ of } \mathit{raise} \ \} \\
 & [R \Rightarrow \mathit{wp}.\mathit{raise}.(P, \mathit{true})] \qquad .
\end{array}
$$

Now the compositions, beginning with normal.

$$
\begin{array}{ll}
 & \text{every free } \mathit{raise} \text{ in } S0; S1 \text{ has precondition } P \text{ in context } R \\
= & \quad \{ \ \text{def. of free occurrences of } \mathit{raise} \text{ in } S0; S1 \text{, and} \\
 & \quad\quad \text{notion of ``context'' for } ; \ \} \\
 & \text{every free } \mathit{raise} \text{ in } S0 \text{ has precondition } P \text{ in context } R\text{, and} \\
 & \text{every free } \mathit{raise} \text{ in } S1 \text{ has precondition } P \text{ in context } M \\
 & \quad \text{for some } M \text{ satisfying } \{R\}\, S0\, \{M\} \\
= & \quad \{ \ \text{induction hypothesis, twice} \ \} \\
 & [R \Rightarrow \mathit{wp}.S0.(P, \mathit{true})] \ \wedge \\
 & \langle \exists M \triangleright [M \Rightarrow \mathit{wp}.S1.(P, \mathit{true})] \ \wedge \ \{R\}\, S0\, \{M\} \rangle \\
= & \quad \{ \ (4.0)\text{: def. of triple } \} \\
 & [R \Rightarrow \mathit{wp}.S0.(P, \mathit{true})] \ \wedge \ \langle \exists M \triangleright [M \Rightarrow \mathit{wp}.S1.(P, \mathit{true})] \ \wedge \ [R \Rightarrow \mathit{wp}.S0.(\mathit{true}, M)] \rangle \\
= & \quad \{ \ \wedge \text{ over } \exists \ \} \\
 & \langle \exists M \triangleright [M \Rightarrow \mathit{wp}.S1.(P, \mathit{true})] \ \wedge \ [R \Rightarrow \mathit{wp}.S0.(P, \mathit{true})] \ \wedge \ [R \Rightarrow \mathit{wp}.S0.(\mathit{true}, M)] \rangle \\
= & \quad \{ \ (3.15) \text{ with } Q := M \ \} \\
 & \langle \exists M \triangleright [R \Rightarrow \mathit{wp}.S0.(P, M)] \ \wedge \ [M \Rightarrow \mathit{wp}.S1.(P, \mathit{true})] \rangle \\
= & \quad \{ \ (3.17) \text{ with } S, Q := S0, \mathit{wp}.S1.(P, \mathit{true}) \ \} \\
 & [R \Rightarrow \mathit{wp}.S0.(P, \mathit{wp}.S1.(P, \mathit{true}))] \\
= & \quad \{ \ (1.3)\text{: } \mathit{wp} \text{ of } ; \ \} \\
 & [R \Rightarrow \mathit{wp}.(S0; S1).(P, \mathit{true})]
\end{array}
$$

Finally, for exceptional composition,

$$
\begin{array}{ll}
 & \text{every free } \mathit{raise} \text{ in } S0 \lhd S1 \text{ has precondition } P \text{ in context } R \\
= & \quad \{ \ \text{def. of free occurrences of } \mathit{raise} \text{ in } S0 \lhd S1 \text{, and} \\
 & \quad\quad \text{notion of ``context'' for } \lhd \ \} \\
 & \text{every free } \mathit{raise} \text{ in } S1 \text{ has precondition } P \text{ in the context of} \\
 & \quad \text{some exceptional postcondition of } S0 \text{ from } R \\
= & \quad \{ \ \text{induction hypothesis} \ \}
\end{array}
$$

$$\begin{array}{ll}
 & \langle \exists M \triangleright [M \Rightarrow wp.S1.(P, true)] \wedge [R \Rightarrow wp.S0.(M, true)] \rangle \\
= & \quad \{ \ (3.16) \text{ with } S, P, Q := S0, wp.S1.(P, true), true \ \} \\
 & [R \Rightarrow wp.S0.(wp.S1.(P, true), true)] \\
= & \quad \{ \ (1.4)\text{: } wp \text{ of } \lhd \ \} \\
 & [R \Rightarrow wp.(S0 \lhd S1).(P, true)] \quad .
\end{array}$$

□

4.2 Usage of exceptions

As a consequence of (4.2) and (4.3), we have

Theorem

$$\begin{array}{ll}
 & \{R\}\, S0 \lhd S1 \,\{Q\} \\
= & \\
 & \{R\}\, S0 \,\{Q\} \ \wedge \\
 & \langle \exists M \triangleright \ \{M\}\, S1 \,\{Q\} \ \wedge \\
 & \qquad \text{every free occurrence of } raise \text{ in } S0 \text{ has precondition } M \text{ in context } R \,\rangle
\end{array}$$

This theorem suggests that, when constructing $S0$ in $S0 \lhd S1$, one should have in mind a precondition M for all *raise* statements that occur free in $S0$; the benefit is then that one may assume that same precondition in the construction of $S1$.

In the next chapter, I put this theorem into action in the construction of a simple program. The theorem is also quite useful in large programs, because it shapes the way we think about using exceptions: If a particular exception is raised only when some particular global condition holds, then the theorem tells us that every handler of that exception can assume that condition upon entry. This suggests that the declaration of an exception in a programming notation provide a way to state this condition.

Finally, I remark on exception parameters, as featured, *e.g.*, in Modula-3 and CLU [54]. The values of these parameters can be considered part of the "global condition" to which I alluded above, since they are accessible both where the exception is raised and where it is handled. Thus, the theorem suggests that these parameters be initialized, at the time the exception is being raised, to satisfy some particular condition. Like the case for other global values, the exception handler can then depend on that condition upon entry.

Chapter 5

Constructing a program with exceptions

In Chapter 4, I developed a theorem regarding the use of exceptions. In this chapter, I employ that theorem in a novel construction of a simple program.

5.0 A program derivation

Equipped with exceptions, the task in this section is to design a program that, given value x and two-dimensional array a of size $M \times N$, computes b, i, j to satisfy

$$Q : \qquad (b \Rightarrow Q0) \wedge (\neg b \Rightarrow Q1) \qquad ,$$

where

$$Q0 : \qquad 0 \leq i < M \ \wedge \ 0 \leq j < N \ \wedge \ a[i,j] = x$$
$$Q1 : \qquad \langle \forall\, m,n \mid 0 \leq m < M \wedge 0 \leq n < N \triangleright a[m,n] \neq x \rangle \qquad .$$

It is understood that the values of x and a may not be changed. As our guide, we will use the theorem from Chapter 4, which states that if, in a proof, every free occurrence of *raise* in S has precondition K, then K can be used as the precondition for handler T in a proof of $S \lhd T$.

To get us started, I suggest we make use of the fact that control sometimes flows through T and sometimes not. I let these two cases correspond to the cases b and $\neg b$. But which goes with which? In order to conclude that x is not present in a, every element of a needs to be tested. This can be done using two nested loops. In order to set i and j to a coordinate of a whose value is x, the program first needs to find such a coordinate. This, too, can be done using two nested loops, but there is no reason to continue the search once an x has been encountered. Exceptions provide a means of breaking out of such loops prematurely. For this reason, I write the first approximation of our program as

$$(S; b := \mathit{false}) \lhd b := \mathit{true}$$

for some S to be developed.

Let us now develop program S, whose normal postcondition we want to be $Q1$. Moreover, the theorem from Chapter 4 tells us that any *raise* statement in S must have precondition $Q0$. Twice using the well-known technique of replacing a constant by a variable (see, *e.g.*, [17], [29, Ch. 16], [20], [77, Ch. 8], or [5, Ch. 4]), we find invariants

$$\begin{array}{ll} P0: & 0 \le i \le M \ \wedge \ \langle \forall\, m,n \mid 0 \le m < i \wedge 0 \le n < N \triangleright a[m,n] \ne x \rangle \\ P1: & P0 \ \wedge \ i \ne M \ \wedge \ 0 \le j \le N \ \wedge \ \langle \forall\, n \mid 0 \le n < j \triangleright a[i,n] \ne x \rangle \end{array}$$

for the outer and inner loops, respectively, and calculate S as

$$\begin{array}{ll} \multicolumn{2}{l}{i := 0\ ;} \\ \mathbf{do}\ i \ne M \mapsto & j := 0\ ; \\ & \mathbf{do}\ j \ne N \mapsto \text{“establish } a[i,j] \ne x\text{”}\ ;\ j := j+1\ \mathbf{od}\ ; \\ & i := i+1 \\ \mathbf{od} & . \end{array}$$

The program segment “establish $a[i,j] \ne x$” concerns us since we are not allowed to modify any of these variables at this stage. Were it not for the presence of exceptions, we would need a miracle at this time. But, since we do have exceptions at our disposal, we just need to verify that one can be raised if $a[i,j] \ne x$ does not hold. We observe that $P1 \wedge j \ne N \wedge a[i,j] = x$ implies $Q0$, so we *can* use a *raise* here. Replacing “establish $a[i,j] \ne x$” with

$$\mathbf{if}\ a[i,j] = x \rightarrow \mathit{raise}\ \square\ a[i,j] \ne x \rightarrow \mathit{skip}\ \mathbf{fi} \qquad ,$$

we are done, and write the entire program as

$$\begin{array}{l} (\ \ i := 0\ ; \\ \quad \mathbf{do}\ i \ne M \mapsto \ j := 0\ ; \\ \qquad\qquad\qquad \mathbf{do}\ j \ne N \mapsto \\ \qquad\qquad\qquad\quad \mathbf{if}\ a[i,j] = x \rightarrow \mathit{raise}\ \square\ a[i,j] \ne x \rightarrow \mathit{skip}\ \mathbf{fi}\ ; \\ \qquad\qquad\qquad\quad j := j+1 \\ \qquad\qquad\qquad \mathbf{od}\ ; \\ \qquad\qquad\qquad i := i+1 \\ \quad \mathbf{od}\ ; \\ \quad b := \mathit{false} \\ ) \\ \lhd \\ \quad b := \mathit{true} \qquad . \end{array}$$

5.1 Discussion

To prove the correctness of our program, we only needed to show that normal termination of the loops maintains the invariants. The rest follows from the invariants, the guards, and the theorem from Chapter 4. The proof of this program is thus simple.

Looking back at the program through operational spectacles, we see the loops followed by $b := \mathit{false}$ as trying to establish the absence of x in a. However, should an x be present, an exception is raised when an x is first encountered. The operation of this program is thus easy to understand.

Finally, consider a similar program that, without exceptions, uses stronger guards to facilitate exiting the loops before $i = M$ and $j = N$, respectively. For example, twice applying the Bounded Linear Search Theorem [17, 20], we arrive at the program

$$
\begin{array}{l}
b, m := \mathit{false}, 0 \ ; \\
\textbf{do} \ \neg b \ \wedge \ m \neq M \mapsto \\
\quad\quad c, n := \mathit{false}, 0 \ ; \\
\quad\quad \textbf{do} \ \neg c \ \wedge \ n \neq N \mapsto \\
\quad\quad\quad\quad c, i, j, n := a[m, n] = x, m, n, n + 1 \\
\quad\quad \textbf{od} \ ; \\
\quad\quad b, m := c, m + 1 \\
\textbf{od} \quad\quad .
\end{array}
$$

In addition to having more complicated invariants (because the invariants need to record the information to prove the postcondition for both conjuncts of Q), this program is arguably less efficient than the one that uses exceptions (because of the extra tests). Thus, we consider our program efficient. The point, however, is not to show that a structured jump can produce a more efficient program; the point is that we have an easy way of constructing such a program hand in hand with its proof.

The heated forum discussion [73] discusses programs that attempt to solve a related programming problem. The above problem can also efficiently be solved using recursion or `goto` statements, as [34] demonstrates with a nice derivation. How exception handling can simplify the structure of certain programs is also discussed in, for example, [13]. More recently, [46] shows exceptions in the construction of programs through refinements.

Chapter 6

Modeling common programming languages

In this chapter, I discuss the relation between the program semantics given in Chapter 1 and the mathematical modeling of popular features found in common programming languages like Modula-3, Ada, or C. I discuss procedures, conditional statements, checked run-time errors, and expressions.

6.0 Procedures

Imperative programming languages provide a mechanism to encapsulate and reuse code, *viz.*, *procedures*. Most languages allow a procedure's *declaration* (or *prototype*) to be given separately from its *implementation* (or *body*). The purpose of the declaration is to convey the information needed to *call* the procedure; this information also circumscribes the implementation. In Modula-3, Ada, and C, the declaration gives the *signature* of the procedure, *i.e.*, a declaration of its parameters, result values, and set of exceptions that it may raise. The separation between declaration and implementation is important when verifying programs. However, the signature does not suffice for this purpose; the declaration must also give a *specification* of the procedure. Calls to the procedure get their semantics from this specification, and the implementation needs to be verified to meet the specification.

6.0.0 Procedure specifications

A simple procedure is declared and specified by

$$\textbf{spec}\ P(\,)\ \textbf{is}\ \mathit{spec} \quad ,$$

where P names the new procedure and *spec* is its specification. The specification is Larch-like [33] and consists of a set of clauses of the forms

$$\begin{array}{l} \textbf{modifies } w \\ \textbf{requires } \mathit{Pre} \\ \textbf{ensures } \mathit{nPost} \\ \textbf{except-ensures } \mathit{ePost} \quad , \end{array} \tag{6.0}$$

where w is a list of variables, and $\mathit{Pre}, \mathit{nPost}, \mathit{ePost}$ are predicates. nPost and ePost are two-state predicates, and thus may mention initial-value variables (see Section 1.7). The specification may contain any number of the clauses in any order. This is equivalent to listing all w's in one **modifies** clause and taking the conjunction of Pre's, nPost's, and ePost's, respectively, for the other clauses. For example, the specification

$$\begin{array}{l} \textbf{modifies } x \quad \textbf{requires } 0 \leq x \quad \textbf{modifies } y \\ \textbf{except-ensures } y_0 < y \quad \textbf{requires } x + y = 10 \end{array}$$

is the same as

$$\begin{array}{l} \textbf{modifies } x, y \quad \textbf{requires } 0 \leq x \wedge x + y = 10 \\ \textbf{ensures } \mathit{true} \quad \textbf{except-ensures } y_0 < y \quad . \end{array}$$

The one exception to the given rule is that the absence of **except-ensures** clauses is treated as **except-ensures** *false* rather than **except-ensures** *true*. This way, a specification must explicitly advertise the fact that it may have an exceptional outcome. (Penelope [62] provides some convenient shorthands for writing specifications of exceptional behaviors.)

The meaning of specification (6.0) is that of the specification statement

$$w : [\mathit{Pre},\ (\mathit{ePost}, \mathit{nPost})] \tag{6.1}$$

(see Section 1.7).

6.0.1 Parameters and result values

Procedures can take *parameters* and can return *result values*. The specification

$$\textbf{spec } r := P(x) \textbf{ is } \mathit{spec} \tag{6.2}$$

declares and specifies a procedure P. P takes a list of parameters, here named x, and returns a list of result values, here named r. x and r are names that may be mentioned in *spec*. x may be *read* (used) by the procedure but not *written* (updated). Thus, x may not appear in the **modifies** list in *spec*. Each procedure call of P (described below) instantiates x with some value. r, on the other hand, may be written by P, but its initial value is unspecified. Consequently, the **modifies** list in *spec* is treated as always containing r, and r may not appear initial-valued in the postconditions in *spec*.

The described behavior of x and r model *copy-in* (or *value*) and *copy-out* parameters.

6.0.2 Procedure calls

A procedure like (6.2) is invoked by the program statement

$$\textbf{call}\ v := P(E) \qquad , \tag{6.3}$$

where v is a list of variables and E is a list of expressions. The lengths of r and v are to be equal, and so are those of x and E. The semantics of this procedure call is defined by

$$[\![\ x, r \bullet x := E\ ;\ w, r : [Pre,\ (ePost, nPost)]\ ;\ v := r\ ]\!] \qquad . \tag{6.4}$$

When a result value r is involved, I write *spec* for the command

$$w, r : [Pre,\ (ePost, nPost)] \tag{6.5}$$

rather than for the command (6.1). Thus, (6.4) can be rendered as

$$[\![\ x, r \bullet x := E\ ;\ spec\ ;\ v := r\ ]\!] \qquad .$$

6.0.3 Procedure implementation

Since procedures are essentially given as specification statements, we don't expect compilers to produce executable code from them (see Sections 1.7 and 1.8). Instead, a programmer supplies an *implementation* of the procedure. A **call** statement is compiled into a subroutine call to the *implementation*, but the piece of code that contains the **call** statement is verified using the *specification* of the procedure. In order for this transformation to be correct, an implementation must be a refinement of the specification (see Section 1.8). Exactly then (well, see Section 6.0.4 below) do we say that the implementation *meets* its specification.

The notation I use for introducing the implementation of a procedure like (6.2) is

$$\textbf{impl}\ r := P(x)\ \textbf{is}\ gc \qquad ,$$

where gc is a guarded command. The signature $r := P(x)$ is repeated here (as opposed to just listing the name P) to emphasize that x and r are identifiers that may be referred to in gc. A procedure has exactly one implementation. P meets its specification just when

$$w, r : [Pre,\ (ePost, nPost)]\ \sqsubseteq\ gc \qquad . \tag{6.6}$$

Applying the definition of $\sqsubseteq$ (1.19), this proof obligation is

$$\langle\, \forall\, P, Q \,\triangleright\, [wp.(w, r : [Pre,\ (ePost, nPost)]).(P, Q)\ \Rightarrow\ wp.gc.(P, Q)]\,\rangle \qquad , \tag{6.7}$$

where P and Q range over all predicates on the final state. This quantification may appear overwhelming to a verification process. As a cure, I present an alternative rendering of this proof obligation in Chapter 11, where I also prove the two renderings equivalent. The alternative rendering does not quantify over all predicates and is therefore sometimes preferable to (6.7), especially when doing automatic verification.

6.0.4 Termination

I left one important detail out of the discussion of when an implementation meets its specification: termination. Since the guarded command *gc* may contain procedure calls, it may, for example, recursively call itself. One must therefore ensure that such recursion eventually ends, so that *gc* may be shown to terminate. For example, consider the following procedure specification and implementation.

spec $P()$ **is** *spec* ;
impl $P()$ **is call** $P()$

Here, we find that the implementation of P is indeed a refinement of its specification (in fact, the two are equal). Nevertheless, as programmers we know the call to P will not establish what is prescribed by *spec*, but will instead result in infinite recursion.

Proving, in the presence of mutually recursive procedures (and also replaceable methods, see Section 8.2), that all procedure calls in an implementation terminate can be quite tricky. This proof obligation may be dealt with separately from the proof obligation of the refinement. If termination is of no concern, *wlp* (as opposed to *wp*) can be used. Then, the refinement (6.6) is the only proof obligation for showing that a procedure implementation meets its specification. This is what I use in Part III, Chapter 11.

6.1 Alternative statements

In languages like Modula-3, Ada, and C, we find the presence of alternative statements like `IF`. The modeling of these is straightforward. For example, the Modula-3 statement

`IF` *b0* `THEN` *S0* `ELSIF` *b1* `THEN` *S1* `ELSE` *S2* `END`

is modeled as

$$b0 \rightarrow S0 \;\square\; b1 \rightarrow S1 \;\square\; S2 \qquad .$$

Since all statements in Modula-3, Ada, and C are total (*cf.* Section 1.4), I may write this statement as

$$\mathbf{if}\; b0 \rightarrow S0 \;\square\; b1 \rightarrow S1 \;\square\; S2 \;\mathbf{fi}$$

(*cf.* (1.10)).

The simpler statement

`IF` *b* `THEN` *S* `END`

is modeled as

$$\mathbf{if}\; b \rightarrow S \;\square\; \mathit{skip} \;\mathbf{fi} \qquad .$$

Because this statement occurs so frequently, I introduce

$$\mathbf{if}\; b \;\mathbf{then}\; S \;\mathbf{fi}$$

as a shorthand for it.

6.2 Statements that "go wrong"

When reasoning about a program, it is often useful to subdivide the notion of non-termination into cases where the program results in a failure caused by a *checked run-time error* like a **nil**-dereference or an array index out-of-bounds error *vs.* cases where the program does not terminate because of, for example, an infinite loop. I will refer to the former as the program "*going wrong*".

This is important, for example, when verifying a run-time system—the execution of a run-time may never terminate, yet one wants to be certain it does not result in a checked run-time error. Note that neither wp nor wlp caters for this distinction.

The distinction can be captured by viewing programs as having *three* possible outcomes: normal, exceptional, and erroneous. This is a straightforward extension of having only two outcomes (*cf.* Section 2.6). The wlp of a program is now

> $wlp.S.(P, Q, E)$ is $true$ of exactly those initial states from which execution of S is guaranteed
>
> - to terminate exceptionally in a state satisfying P, or
> - to terminate normally in a state satisfying Q, or
> - to terminate erroneously (go wrong) in a state satisfying E, or
> - to not terminate at all.

The interpretation of wp is extended similarly.

With these triples, I define a third unit statement, $wrong$ (*cf.* Section 1.2),

$$wp.wrong.(P, Q, E) \;=\; E \qquad wlp.wrong.(P, Q, E) \;=\; E \qquad . \tag{6.8}$$

Statement $wrong$ is used to model a run-time error. Note that execution of $wrong$ actually does terminate.

In some programming languages, like Ada, a programmer can provide a handler for run-time errors. That calls for a third type of statement composition (the other two being ; and $\lhd$, see Section 1.2). Alternatively, if checked run-time errors are to be avoided at all costs, perhaps because they may result in the operating system aborting program execution (*cf.* Modula-3), no additional statement composition is needed. This is the view I take in this thesis. Hence, when computing the weakest (liberal) precondition of a statement, we are always interested in using $false$ as the third component in the postcondition triple. To avoid cluttering formulas, I thus show only the first two components, the third component always being $false$ implicitly.

Rewriting (6.8) with the convention of omitting the third component, we have

$$wp.wrong.(P, Q) \;=\; false \qquad wlp.wrong.(P, Q) \;=\; false \qquad . \tag{6.9}$$

> **Remark 6.0.** A desirable property of a program S is that $wlp.S$ be universally conjunctive. Although (6.9) seems to indicate that $wlp.wrong$ is not universally conjunctive (since $wlp.wrong.(true, true)$ is not $true$),

one should remember that $wlp.S.(P, Q)$ in (6.9) is only a shorthand for $wlp.S.(P, Q, false)$. The general form (6.8) is indeed universally conjunctive, so, for example, we have

$$wlp.wrong.(true, true, true) \;=\; true \qquad .$$

With the convention of omitting the third component, all statement definitions given in Chapter 1 remain unchanged. Of those statements, only the specification statement introduces a way for a statement to go wrong. In Section 1.7, I deferred discussing the operational interpretation of the specification statement started in a state that does not satisfy the precondition. I give that interpretation now: If Pre does not hold in the state from which $w : [Pre,\ (ePost, nPost)]$ is executed, the program goes wrong. Hence, the full definition of the specification statement is given by

$$\begin{array}{l} wp.(w : [Pre,\ (ePost, nPost)]).(P, Q, E) \quad = \\ \qquad (Pre \wedge \langle \forall\, w \triangleright\ (ePost \Rightarrow P) \wedge (nPost \Rightarrow Q)\ \rangle[v_0 := v]) \ \vee\ (\neg Pre \wedge E) \qquad . \end{array}$$

Since the statement always terminates, its wlp coincides with its wp. With $E := false$, we have definition (1.17), as given originally.

6.2.0 Assert statement

The $wrong$ statement always goes wrong. The *assert* statement is a statement that goes wrong only under a parameterized condition. For any predicate b, the assert statement is defined as

$$\textbf{assert}\ b \quad = \quad \textbf{if}\ \neg b\ \textbf{then}\ wrong\ \textbf{fi} \qquad .$$

The weakest precondition of **assert** thus satisfies

$$wp.(\textbf{assert}\ b).(P, Q) \;=\; b \wedge Q \qquad , \tag{6.10}$$

and similarly for its weakest liberal precondition. Interpreted operationally, **assert** b goes wrong just when executed in a state where b does not hold. If b does hold, **assert** b terminates normally and does not alter the program state. Note that **assert** $false$ coincides with $wrong$.

6.3 Expressions

Expressions in many programming languages allow conveniences like *short-circuit* (or *conditional*) operators. Calls to procedures with one result value are also allowed in expressions. Another issue is that some expressions are not always defined. In this section, I treat the modeling of such expressions.

The first step in this modeling is to break complex expressions up into smaller ones. This is a well-known technique in, for example, the generation of intermediate three-address code in compiler design [0]. For example,

$$x := y + P(z, Q(w))$$

is elaborated into

$$\begin{array}{ll} [\![& t0, t1 \\ \bullet & \textbf{call } t0 := Q(w) \\ ; & \textbf{call } t1 := P(z, t0) \\ ; & x := y + t1 \\ ]\!] & \quad , \end{array}$$

where $t0$ and $t1$ are temporary local variables with fresh names. This example shows how to model procedure calls in expressions. (Alternatively, one can view this as the *definition* of procedure calls occurring in expressions.) Note that, since procedures P and Q may have side effects, so may the evaluation of the expression $y + P(z, Q(w))$.

Short-circuit operators are handled similarly. For example,

$$x := B \textbf{ cand } C \qquad ,$$

where **cand** denotes the conditional-and operator, is elaborated into (or, gets its definition from)

$$[\![\, t \bullet t := B \;;\; \textbf{if } t \textbf{ then } t := C \textbf{ fi} \;;\; x := t \,]\!] \qquad ,$$

where $t := B$ and $t := C$ may require further elaboration.

Some operators are not defined for all operands. For example, division is not defined for a second argument of 0, and the array indexing operator (introduced in Chapter 7) is not defined for indices outside the index set of the array. Expressions involving such operators are called *partial*.

Sometimes "not defined" means the value of the expression in unspecified; more frequently, "not defined" means evaluation of the expression results in a checked run-time error. In the case of the latter, a statement

$$x := a \textbf{ div } b$$

is elaborated into

$$\textbf{assert } b \neq 0 \;;\; x := a \textbf{ div } b \qquad ,$$

which incorporates the prescribed run-time check.

In the examples above, I have only shown expressions that occur in assignment statements. Expressions that occur elsewhere are handled in a similar way. For example,

```
WHILE B DO S END
```

is elaborated into

$$
\begin{array}{ll}
[\![& b \bullet b := B \\
; & \textbf{do } b \mapsto S \ ; \ b := B \textbf{ od} \\
]\!] & \quad ,
\end{array}
$$

where the two occurrences of $b := B$ may require further elaboration.

As a final concern, I discuss the order of evaluation of parameters. For example, a programming language may not specify the order of evaluation of the operands of $+$. The order makes a difference when the operands may have side effects. Thus, a statement like $x := A + B$ can be elaborated into

$$
\begin{array}{ll}
[\![& t0, t1 \\
\bullet & (t0 := A \ ; \ t1 := B \ \square \ t1 := B \ ; \ t0 := A) \\
; & x := t0 + t1 \\
]\!] & \quad .
\end{array}
$$

The price of this elaboration —exponentially longer formulas in its verification— compares unfavorably to the marginal value of this elaboration over one like

$$[\![\ t0, t1 \bullet t0 := A \ ; \ t1 := B \ ; \ x := t0 + t1 \ ]\!] \quad .$$

Instead, one can disallow expressions for which different permitted orders have an effect on the result of the computation. Ada, for example, defines such code to be *erroneous*, meaning that it causes an *unchecked* run-time error. It only seems fair that a verification process should catch such programming errors. To detect such possibilities, procedure specifications need to mention not only the variables a procedure writes (recall, these are given in the **modifies** clause), but also those it reads. Such information is facilitated, for example, by the **global in** construct in Penelope [62].

Part II

Data Structures

Data structures

In Chapter 1 of Part I, I described statements of an imperative programming notation and gave their semantics. Two of these statements, the assignment and specification statements, modify the state of a program, whereas the others deal only with the flow of control of the program. The particular types and internal structure of the variables are of no importance in Part I; I use only the fact that the variables are independent of each other, so that the update of one variable does not affect the values of other variables (see Section 1.1).

In practical settings, the state space of a program is more complicated. A program may update only some portion of a variable, *e.g.*, an element of an array or a field of a record. A program may also have several ways of referring to the same piece of data, *e.g.*, through two identical indices into an array or through references (pointers). Object-oriented languages provide an even fancier mechanism, known as *subtyping*, for organizing the data of a program.

In this Part, I describe the data structures most commonly provided by imperative languages: subranges, enumerations, arrays, records, sets, references, and objects. I show how these are modeled in the programming notation from Chapter 1.

Outline

In Chapter 7, I introduce types and describe how to model the most common data types using the constructs I define. In Chapter 8, I introduce into my programming notation the mechanisms necessary to deal with objects. Readers familiar with notions of subtyping such as [48, 56] may be in for a pleasant surprise: separation of concerns and thus simplicity. In Chapter 9, I show how to make sense of this simplicity when specifying and implementing objects. In that chapter, I present *data abstraction* and *data refinement* as these are known in the literature (*cf.* [38, 30, 43, 3, 11, 24]). As I show in Part III, this view of data refinement is not sound except in very restrictive modular programs. Nevertheless, Chapter 9 gives a flavor of the kinds of abstractions in which I am interested, and provides a nice coverage of the concepts used in Part III.

Chapter 7

Data types

In this chapter, I introduce types and global variables. I then explain how to model data types from common programming languages, like arrays and records, in the programming notation from Chapter 1. I also define a construct that introduces a new type into a program, and show how it can be used in the modeling of references.

I assume some familiarity with these data types from a language like Modula-3, Ada, or C.

7.0 Types

A variable may be of a certain *type*. A type is a possibly empty, possibly infinite set of values. Examples of types are **int**, the set of integers, and **bool**, the set $\{false, true\}$. Variables can only be declared to be of nonempty types.

7.0.0 Composite types

Types can be composed in two ways to form new types. These compositions are familiar from set theory.

For any types S and T, $S \times T$ is the type consisting of *pairs* of values, one from S and one from T.

Much more important is the *map* type $S \to T$. Type S in $S \to T$ is called the *index* type (or set), and T is called the *element* type. Values of map types, called *maps*, can be applied to values of their index type, called *indices*. Every map is *total*, *i.e.*, it can be applied to *any* element of its index set. Consider a variable a of type $S \to T$. Given an expression i of type S, $a[i]$ is an expression of type T. The value of this expression is the value of a applied to i. Because I often think of maps and arrays as being synonymous, I often say "a *at* index i" or "a *indexed* with i" instead of "a applied to i". I will also refer to this operation as *array dereferencing*.

A map can be updated at a particular index using the assignment statement

$$a[i] := E \qquad . \tag{7.0}$$

This is shorthand for

$$a := subst.a.(i : E) \qquad , \tag{7.1}$$

where $subst.a.(i : E)$ is the map that differs from a only in that, applied to i, $subst.a.(i : E)$ yields E [37, 40, 17].

> **Remark 7.0.** Fortunately, in spite of its shape, we know of an efficient implementation of (7.1), *viz.*, the one that (7.0) suggests: Just change the state of $a[i]$, element i of array a.

Formally, the meaning of *subst* is given by the following two axioms.

$$subst.a.(i : E)\ [j] \ = \ E \qquad \text{if } i = j \tag{7.2}$$
$$subst.a.(i : E)\ [j] \ = \ a[j] \qquad \text{if } i \neq j \tag{7.3}$$

Axiom (7.2) states that indexing $subst.a.(i : E)$ with i yields E. Axiom (7.3) states that indexing $subst.a.(i : E)$ with j, where $i \neq j$, yields the same result as indexing a with j. (See [69] on how these axioms are used in automatic theorem proving.)

7.0.1 Typed variables

When writing a variable, I will sometimes write its type following the variable identifier, separated by a colon. A variable x of type T is thus denoted

$$x : T \qquad .$$

This requires that T be nonempty.

I distinguish between *local* and *global* variables. Local variables are declared by block statements (see Section 1.3). They are created at the beginning of execution of the block and perish as control leaves the block.

Global variables, on the other hand, are created at the beginning of a program execution and survive all changes of the current control point in the execution. (In a modular program, variables may or may not be *visible* at the current control point, but that doesn't play a rôle until Part III.)

I introduce each global variable with a **var** declaration, as in

$$\mathbf{var}\, x : T \qquad .$$

Similarly, a local variable may be given a type at the point of declaration. As the semantics of blocks suggests, I assume the initial value of a variable to be any value of its type. (This matches Modula-3's definition, but not Ada's or C's, neither of which provides this guarantee.)

> **Remark 7.1.** The Ada language [1] does not require that the language implementation initialize variables, and defines programs that use an uninitialized variable as *erroneous*. The rationale behind this may be the fact that Ada is designed to support systems programming. This

means that a variable can be mapped to a register in a machine, and any write to such a variable may have a hardware side effect such as disabling interrupts or sending a packet. However, Ada *does* require that the language implementation initialize *pointers* to **nil**. So much for allowing variables to be mapped to registers.

The C programming language [45] contains three simple types: integers, reals, and pointers, each grouped into possibly different sizes. Since any values of the bits that represent an integer make up *some* integer value, an integer always contains a value of its type, and similarly for reals on many machines. The bits representing a pointer, however, may denote an invalid address, *i.e.*, an address that cannot be dereferenced, because of, for example, memory protection. C does not guarantee any particular initialization of pointer variables.

7.0.2 Narrowing

Let v be a variable of type S and E an expression of type T. If $T \subseteq S$, then an assignment

$$v := E \tag{7.4}$$

is always legal. If $S \subset T$, the assignment is also allowed; however, since some potential values of E are not assignable to v, a run-time check, called a *narrow* check, is necessitated. Assignment (7.4) is thus treated as

$$[\![\ t : T \bullet t := E \ ; \ \textbf{assert}\ t \in S \ ; \ v := t\ ]\!] \quad ,$$

where t denotes a temporary variable with a fresh name (*cf.* partial expressions, Section 6.3).

Remember that parameters and result values of procedures are defined via assignments (Section 6.0); thus, narrowing applies there, too. For example, for $S \subseteq T$,

$$\begin{array}{l}\textbf{spec}\ P(x : S)\ \textbf{is} \\ \quad \textbf{requires}\ Q\end{array}$$

is equivalent to

$$\begin{array}{l}\textbf{spec}\ P(x : T)\ \textbf{is} \\ \quad \textbf{requires}\ x \in S\ \wedge Q \qquad .\end{array}$$

7.1 Types in common programming languages

In this section, I describe the correspondence between the previous section and data structures in programming languages like Modula-3.

7.1.0 Subranges

In Modula-3, a *subrange* is a type like $[M..N]$, where M and N are integer constants. It contains the (possibly empty) inclusive range of integers from M to N.

Subranges need not be treated as separate types in my formalism. Consider a variable x of type $[M..N]$. Like all other variables, x is initialized to a value of its type. Values assigned to x (*i.e.*, E in $x := E$) are checked at compile-time to be integers. However, the restriction of E being in the correct range of the integers is checked at run-time by narrowing. Hence, $x := E$ is treated as

$$[\![\ t : \textbf{int} \bullet t := E\ ;\ \textbf{assert}\ M \le t \wedge t \le N\ ;\ x := t\ ]\!] \qquad .$$

Consequently, $M \le x\ \wedge\ x \le N$ is an invariant of the program. This fact can then be used in proofs, as required, for example, in showing that

$$[\![\ y : [2{\cdot}M..2{\cdot}N] \bullet y := 2{\cdot}x\ ]\!]$$

does not go wrong.

7.1.1 Enumerations

In Modula-3,

```
TYPE E = {egg, sugar, flour};
```

is an example of a declaration of an *enumeration* type E. This particular type has three elements, written $E.egg$, $E.sugar$, and $E.flour$. These names may have a meaning to a programmer. Mathematically, however, I treat them simply as an alternate notation for 0, 1, and 2, respectively. This renders unnecessary the introduction of a new domain of values and an order thereon. Hence, the type E is treated as the subrange $[0..2]$.

A programming language may impose additional restrictions regarding the use of these types. For example, a variable of type E cannot be assigned to a variable of type **int**. These restrictions enforce a disciplined use of the types and do not pose any problem in the theory.

7.1.2 Arrays

An *array* type

```
ARRAY S OF T
```

is treated as the map $S \to T$. The expression $a[i]$ is partial (Section 6.3) and can only be evaluated if i is in the index set of a, something that in general needs a narrow check at run-time (for example, if S is a subrange and i is an integer).

Commonly, programming languages like Modula-3 define

```
ARRAY S0, S1 OF T
```

to be a shorthand for

$$\texttt{ARRAY}\ S0\ \texttt{OF ARRAY}\ S1\ \texttt{OF}\ T \qquad ,$$

(or, in the theory, $S0 \to (S1 \to T)$), and similarly $a[i,j]$ for $a[i][j]$. Thus,

$$\begin{array}{ll} & a[i,j] := E \\ = & \quad \{ \ \text{Modula-3 shorthand} \ \} \\ & a[i][j] := E \\ = & \quad \{ \ (7.1)\text{: array update shorthand} \ \} \\ & a[i] := subst.(a[i]).(j : E) \\ = & \quad \{ \ (7.1)\text{: array update shorthand} \ \} \\ & a := subst.a.(i : subst.(a[i]).(j : E)) \qquad . \end{array}$$

This allows $a[i]$ to be treated as an array in its own right. If that feature is not needed, one may prefer to treat this array type as

$$(S0 \times S1) \to T \qquad .$$

Then, an index is a pair, and thus $a[i,j] := E$ is simply

$$a := subst.a.((i,j) : E) \qquad ,$$

that is, a gets a in which element (i,j) has been replaced by E.

7.1.3 Records

An example Modula-3 *record* type is

$$\texttt{TYPE}\ R = \texttt{RECORD}\ f0 : T0\ ;\ f1 : T1\ \texttt{END;} \qquad .$$

$f0$ and $f1$ are distinct identifiers known as the *fields* of R. They have the respective types $T0$ and $T1$. For a value $r : R$, $r.f0$, called a *field dereference*, denotes field $f0$ of r.

This record type can be thought of as an array type with index set $\{f0, f1\}$. Thus, $r.f0$ means $r[f0]$. By treating records as arrays, no additional theory is required; the array axioms suffice.

Note that $r[f0]$ and $r[f1]$ have types $T0$ and $T1$, respectively, and that these types may differ. This causes no problem, even in languages that do typing at compile-time, because the only way to index r is by the constants $f0$ and $f1$ themselves. The type of a field dereference is thus immediately available.

7.1.4 Sets

Modula-3 writes a *set* of elements of a type T as

`SET OF` T .

It, too, can be treated as a map, *viz.*,

$T \rightarrow$ **bool** .

Hence, a set operation like

t `IN` s ,

where s is of type `SET OF` T and t is of type T, is taken to be the boolean expression

$s[t]$.

7.2 Declaring new types

Just as variables can be declared by **var** declarations, my programming notation allows new types to be defined by **type** declarations, as in

type R .

This introduces a new name R, and declares it to be a type. R contains an infinite number of elements, all but one of which differ from the elements of other types declared by **type**. The one exception is a special constant called **nil**, which is part of every type declared by **type**.

I will use use the names *reference* type or *object* type when referring to a type declared by **type**. Similarly, I will call elements of such a type *references* or *objects*. The inspiration for these names is discussed below and in the next chapter.

7.2.0 References

In common imperative languages, a *reference* type is sometimes called a *pointer* type. (Ada calls them *access* types.) Every reference type has a *referent* type, that is, the type to which the reference type is a reference. A reference is usually implemented as the address of its *referent*, a piece of data residing in the heap of a program. In Modula-3,

`REF` T

is a reference type whose referent type is T.

A reference r is *dereferenced* by $r\wedge$. This maps r to its referent, a fact that reveals the need for a map of type $(\texttt{REF } T) \rightarrow T$ [40].

Let R be a unique name for a particular reference type with referent type T. R is then modeled by

$$\textbf{type}\ R\ ;\qquad \textbf{var}\ \mathit{map}{\star}R : R^{-} \rightarrow T \quad ,$$

where R^{-} denotes $R \setminus \{\textbf{nil}\}$ and $\star$ is a reserved character so that $\mathit{map}{\star}R$ is a name uniquely determined by R. $\mathit{map}{\star}R$ is called a *dereference map* (or *collection*). A dereference r^ is then simply the array dereference $\mathit{map}{\star}R[r]$. Note that the index type of $\mathit{map}{\star}R$ is R^{-}, so $\mathit{map}{\star}R[r]$ is a partial expression that can be evaluated only for references r other than **nil** (see Section 6.3).

> **Remark 7.2.** Dereference maps are explicitly declared and manipulated in the programming language Euclid [47] and in an early version of Pascal [79].

The crux with modeling pointers in a theory is the aliasing they introduce. Using maps (arrays) to model references reduces the problem to the aliasing problem of indices into arrays; this, in turn, is handled by the axioms (7.2) and (7.3). This assumes that all access of referents go via references, *i.e.*, there is no way other than ^ to alias a referent. This is true in Modula-3 and Ada, where all references point into the heap, but not in C or C++, where one can take the address of variables (and of just about everything else). Thus, only a disciplined subset of C and C++ can be modeled directly by the techniques I have described here.

7.2.1 Allocation and deallocation

Languages like Modula-3, Ada, and C++ that provide references also provide a mechanism to create new references and referents. In Modula-3,

$$\mathtt{NEW}(R) \quad ,$$

where R is a reference type, returns a new reference of type R. In effect, `NEW` also allocates a referent in the program heap to which the new reference points. This can be modeled by introducing a map

$$\textbf{var}\ \mathit{allocated}{\star}R : R^{-} \rightarrow \textbf{bool} \quad ,$$

initialized so that

$$\langle \forall\, r : R^{-} \ \triangleright\ \neg \mathit{allocated}{\star}R[r] \ \rangle \quad .$$

$\mathtt{NEW}(R)$ is the only construct that modifies $\mathit{allocated}{\star}R$. The specification of $\mathtt{NEW}(R)$ as a procedure is given as

$$\begin{array}{l} \textbf{spec}\ \ r : R^{-} \ := \ \mathtt{NEW}(R)\ \ \textbf{is} \\ \quad \textbf{modifies}\ \mathit{allocated}{\star}R \\ \quad \textbf{ensures}\ \neg \mathit{allocated}{\star}R_0[r] \wedge \mathit{allocated}{\star}R[r] \wedge \\ \qquad\qquad \langle \forall\, s : R^{-} \mid s \neq r \ \triangleright\ \mathit{allocated}{\star}R_0[s] = \mathit{allocated}{\star}R[s] \ \rangle \quad . \end{array}$$

Remark 7.3. Recall from Section 1.7 that a variable subscripted with 0 refers to the initial value of that variable. So, $allocated{\star}R_0$ refers to the value of $allocated{\star}R$ on entry to the procedure.

This assumes there is always enough memory for new referents.

Deallocation of a referent is modeled similarly. A map

$$\textbf{var}\ deallocated{\star}R : R^- \to \textbf{bool} \quad ,$$

initialized like $allocated{\star}R$, is introduced. Then, a procedure `FREE` is defined.

$\textbf{spec}\ \texttt{FREE}(r : R^-)\ \textbf{is}$
$\quad \textbf{modifies}\ deallocated{\star}R$
$\quad \textbf{requires}\ allocated{\star}R[r] \wedge \neg deallocated{\star}R[r]$
$\quad \textbf{ensures}\ deallocated{\star}R[r] \wedge$
$\qquad\qquad \langle\, \forall s : R^- \mid s \neq r \triangleright deallocated{\star}R_0[s] = deallocated{\star}R[s] \,\rangle$

This introduces another requirement on evaluating the expression $map{\star}R[r]$, *viz.*,

$$\neg deallocated{\star}R[r] \quad .$$

Because `NEW`(R) and `FREE` are the only procedures that modify $allocated{\star}R$ and $deallocated{\star}R$ —these map variables are not accessible like regular program variables—, we can prove

$$\langle\, \forall r : R^- \triangleright deallocated{\star}R[r] \Rightarrow allocated{\star}R[r] \,\rangle$$

to be an invariant of any program execution.

The reason for providing a procedure like `FREE` is to gain (if "gain" is really the right word) programmer-defined control of storage efficiency. A pleasant alternative is for the run-time system to assume control of this, an effort realized by a *garbage collector*. The run-time system then reclaims the storage of referents to which no reference exists. This choice is pursued by, for example, Modula-3, thus rendering $deallocated{\star}R$ and `FREE` unnecessary.

7.3 Maps and specifications

Consider a simple procedure, call it $Update$, whose effect is

$$a[i] := y$$

for some global map variable a and appropriately typed parameters i and y. Since $Update$ updates a,

$$\textbf{modifies}\ a$$

is part of $Update$'s specification. The final value of $a[i]$ is specified by

$$\textbf{ensures}\ a[i] = y \quad .$$

Since no other elements of a are updated, the clause

$$\textbf{ensures}\ \langle \forall k \mid k \neq i \triangleright a_0[k] = a[k] \rangle$$

is also part or *Update*'s specification.

Updating a map variable at one given index is quite common, as examples throughout the rest of this thesis show. Therefore, I introduce the shorthand

$$\textbf{modifies}\ a[i] \qquad .$$

Roughly speaking, this means

$$\textbf{modifies}\ a \quad \textbf{ensures}\ \langle \forall k \mid k \neq i \triangleright a_0[k] = a[k] \rangle \qquad .$$

I say "roughly" because of some subtle and messy details, described next.

Consider a procedure $Swap(i,j)$ that swaps $a[i]$ and $a[j]$. It would be convenient to be able to write its specification as

$$\begin{array}{l} \textbf{spec}\ Swap(i,j)\ \textbf{is} \\ \quad \textbf{modifies}\ a[i], a[j] \\ \quad \textbf{ensures}\ a[i] = a_0[j] \land a[j] = a_0[j] \qquad . \end{array}$$

However, applying the rough formulation of the shorthand, we get

$$\begin{array}{l} \textbf{spec}\ Swap(i,j)\ \textbf{is} \\ \quad \textbf{modifies}\ a, a \\ \quad \textbf{ensures}\ a[i] = a_0[j] \land a[j] = a_0[j] \land \\ \qquad\qquad \langle \forall k \mid k \neq i \triangleright a_0[k] = a[k] \rangle \land \\ \qquad\qquad \langle \forall k \mid k \neq j \triangleright a_0[k] = a[k] \rangle \qquad . \end{array}$$

For $i = j$, this **ensures** condition simplifies to

$$\textbf{ensures}\ a_0 = a \qquad ,$$

and for $i \neq j$, it simplifies to

$$\textbf{ensures}\ a_0 = a \land a_0[i] = a_0[j] \qquad ,$$

which makes the specification a partial command (Section 1.4)—if $i \neq j \land a[i] \neq a[j]$ holds initially, *Swap* needs a miracle to establish $a_0[i] = a_0[j]$. This is not the intended specification, a fact for which I blame the formulation of the shorthand. Instead,

$$\textbf{modifies}\ a \quad \textbf{ensures}\ \langle \forall k \mid k \neq i \land k \neq j \triangleright a_0[k] = a[k] \rangle$$

does the trick. I leave it to the reader to write the straightforward but messy generalization of this rule.

There is another detail to be discussed. In the examples above, i and j are expressions whose values are unchanged by the procedure. Consider a variation of procedure *Update* where i is not a parameter but a global variable. Then consider the specification

$$\begin{array}{l} \textbf{spec } \mathit{UpdateAndAdvance}(y) \textbf{ is} \\ \quad \textbf{modifies } a[i], i \\ \quad \textbf{ensures } a[i_0] = y \wedge i = i_0 + 1 \qquad . \end{array}$$

This specifies *UpdateAndAdvance* to write y at $a[i]$ and then to increment i by 1. Here, the intention is that the **modifies** clause be a shorthand for

$$\textbf{modifies } a, i \quad \textbf{ensures } \langle \forall k \mid k \neq i_0 \triangleright a_0[k] = a[k] \rangle$$

as opposed to

$$\textbf{modifies } a, i \quad \textbf{ensures } \langle \forall k \mid k \neq i \triangleright a_0[k] = a[k] \rangle \qquad . \tag{7.5}$$

The same holds true for many other examples, but it is conceivable that one may sometimes want (7.5). Yet another choice is to allow the shorthand only for constant indices.

Chapter 8

Objects

In this chapter, I discuss object types. These are similar to reference types, but cannot be dereferenced using ^, and thus lack the $map{\star}R$ map that every reference type R has. Instead of that one map, object types can have several maps, called *data fields*. In addition, object types feature *subtyping* and *methods*. I describe each of these features, and conclude by relating objects in my notation to those in common programming languages.

> **Remark 8.0.** Since C++ terminology of objects differs from the usual ones [28, 64], readers familiar with C++ but with no other object-oriented language may want to read Section 8.6.1 before reading this chapter from the start.

8.0 Subtypes

Every object type has an infinite number of elements, one of which is **nil** (see Section 7.2). S is called a *subtype* of an object type T just when S, too, is an object type and S is a subset of T. T is then called a *supertype* of S.

As described in the previous chapter, an object type T is declared by

type T .

I now define a way to declare one object type from another. For any object type T,

type $S <: T$

introduces a new name S, and declares it to be a subtype of T different from T itself, *i.e.*, a *proper* subtype of T. T is called the *immediate* supertype of S (see also Remark 8.1).

If $T0$ is a subtype of T, then any $T0$ object, t say, is also a T object (because $t \in T0 \subseteq T$). The smallest subtype Tn such that $t \in Tn$ is called the *dynamic* or *allocated* type of t. For any object type T, $\mathtt{NEW}(T)$ is guaranteed to return an object whose allocated type is T.

8.1 Data fields

For any object type T, I call a map

$$\mathbf{var}\, x : T^{-} \to X$$

a *data field* of type T (or of some T object). x (or x applied to some T object) can be thought of as an *attribute* or *property* of T (or of that T object).

Note that a subtype shares (or *inherits*) the properties of its supertypes: For $T0$ a subtype of T and t a (non- **nil**) $T0$ object, x can be applied to t, because t is in the index set of x ($x \in T0^{-} \subseteq T^{-}$).

8.2 Methods

Like values of any other type, objects can be passed as parameters to procedures. In addition, objects have special procedures, called *methods*, that can be applied to them. An object type T declares a method m by

$$\mathbf{method}\, t : T\ \mathbf{spec}\ r := m(x)\ \mathbf{is}\ spec \qquad . \tag{8.0}$$

This is similar to a procedure specification (6.2) except for the prefixed " $\mathbf{method}\, t : T$ ". The " $\mathbf{method} \ldots T$ " shows that the method is declared for object type T. T is called the *declaring type* of m. The " t : " introduces a name for a special parameter; this parameter is called the *receiver* and is often referred to as *self* [28, 64] or *this* [54, 23]. Consequently, t abides by the same rules as x, with regard to being mentioned in *spec* (see Section 6.0).

An invocation of a method like (8.0) is written

$$\mathbf{call}\ v := o.m(E) \qquad , \tag{8.1}$$

where v is a variable, o is an expression of type T, and E is a list of expressions (*cf.* (6.3)). This invokes method m on object o. The semantics of this method invocation (*cf.* (6.4)) is

$$[\![\ t, x, r \bullet t, x := o, E\ ;\ w, r : [Pre,\ (ePost, nPost)]\ ;\ v := r\ ]\!] \qquad ,$$

where $w, r : [Pre,\ (ePost, nPost)]$ is the method specification *spec* interpreted as a specification statement (*cf.* (6.1,6.5)).

8.3 Method implementations

Unlike procedures, which are restricted to one implementation, a method can have one implementation per subtype of the declaring type. Let T denote the allocated type of o. Then, the method invocation (8.1) is implemented as a subroutine call to T's implementation of m.

If no explicit implementation is given for a particular subtype in a program, the method implementation defaults to that of the immediate supertype. For simplicity, I assume this definition leads to some implementation for every method invocation of every execution of a program.

> **Remark 8.1.** This is the only place where the notion of an *immediate* supertype comes into play. Everywhere else,
>
> $$\textbf{type}\ S <: T$$
>
> can be interpreted as declaring S as *any* proper subtype of T—T need not be the *immediate* supertype. Before one can link and execute such a program, the name of the S's immediate supertype must be given, so that methods can be implemented as described above. Modula-3 features so-called *partially opaque types*, which take advantage of this.

Let $T0$ be a subtype of T. Then, the notation used for associating with $T0$ an implementation of method m (8.0) is

$$\textbf{method}\ t : T0\ \textbf{impl}\ r := m(x)\ \textbf{is}\ gc \qquad .$$

t, r, x are identifiers that may be used in the guarded command gc. For the implementation to meet its specification, the condition

$$w, r : [Pre,\ (ePost, nPost)]\ \sqsubseteq\ gc$$

must be established for t of type $T0$.

8.4 Object simplicity

This is all there is to objects. Note that the notion of a subtype is as simple as the notion of a subset. Unlike [48, 56], the data fields have nothing to do with the subtype relation. Each data field is introduced independently, and I never mention that the fields of a subtype *data refine* or *simulate* those of a subtype. These concepts have to do with abstraction (see next chapter), not subtyping.

And nowhere do I need to consider the complete set of methods of an object type (though, admittedly, unlike [56], I am only considering properties of sequential programs). Instead, methods are declared independently of each other. A method implementation for a particular subtype has very little to do with the subtype itself. Instead, refinement is what matters. Notice also that the refinement is always between an implementation and the (one and only) specification of a method, not between the implementation at one type and the implementation at the immediate supertype.

Finally, note that the semantics of a method invocation depends only on the method specification. Thus, this, too, is independent of subtypes.

Both [48] and [56] seem to treat the issue of what it means for a collection of data fields and methods to form a data type, and the relation between such types. Thus, they consider all properties of the behavior of such types. When reasoning about the correctness of a program (as opposed to reasoning about properties of a type), different properties of a type can be considered in isolation —for example, reasoning about the effect of one method is orthogonal to the effects of other methods—; in fact, considering them in isolation may be preferable, because it may simplify the verification conditions. Nevertheless, techniques like those in [48, 56] provide some utility in the design of object types and object-type hierarchies.

In summary, by separating the concepts of subtyping and abstraction, I achieve simplicity.

8.5 Language implementations of objects

The fact that data fields are introduced separately and are maps whose index sets are infinite poses no problems in language implementations. At any point in a program execution, only a finite number of objects have been allocated, and only the allocated portion of an index set will ever be used to dereference a map. Instead of placing a data field next in memory to the same data field for each object (as the map notation may suggest), all data fields for one object are placed next to each other. Thus, for a data field x and an object t, $x[t]$ is stored at some offset —a function of the name x— into t's data record, not at index t of array x as the notation may suggest.

Most object-oriented programming languages use a notation like $t.x$ instead of $x[t]$. This has appeal, especially to object-orientation buffs—"t's attribute x" puts the stress on t, whereas "x at t" puts x in the spotlight. My focus, however, is on the mathematical meaning of objects.

An implementation makes sure that accessible from an object is an identification of its allocated type. This type identification makes it possible to dispatch to the right method implementation in a method invocation.

8.6 Objects in common programming languages

In this section, I describe the correspondence between the object types presented here and those in common programming languages. I focus on Modula-3 and C++; other languages are similar.

8.6.0 Modula-3

In Modula-3, an object type T is declared by

TYPE $T = $ *SuperT* OBJECT
  fieldlist
 METHODS
  methodlist

```
OVERRIDES
  overridelist
END;
```

SuperT gives the name of the immediate supertype of T. Hence, in my notation, T is declared by

$$\textbf{type}\ T <: \mathit{SuperT} \quad .$$

fieldlist is a list of field declarations. In my notation, each such field x of type X is declared by

$$\textbf{var}\ x : T^{-} \to X \quad .$$

Similarly, every method in the method list is declared with a **method** T **spec** declaration. Note, though, that my notation requires a specification, whereas Modula-3 provides no way to state a specification (except, of course, informally as a comment).

In Modula-3, an implementation P for a method m is specified by appending "$:= P$" to m's declaration in *methodlist* for T, if T is the declaring type of m. If m is declared in a proper supertype of T, then $m := P$ is given as an element in *overridelist*. This P must name a procedure. For a method m specified by (8.0), the specification of P must have the form

$$\textbf{spec}\ r := P(t : T0',\ x)\ \textbf{is}\ \mathit{pspec} \quad ,$$

where $T0'$ names some supertype of $T0$. (The names r, t, and x are allowed to differ in *spec* and *pspec*, since they are local to each specification. For simplicity, I assume them to be the same.) For P to be a valid implementation of m, one must then prove, for t of type $T0$,

$$\mathit{spec}\ \sqsubseteq\ \textbf{call}\ r := P(t, x) \quad ,$$

or stated differently,

$$\mathit{spec}\ \sqsubseteq\ \mathit{pspec} \quad .$$

Since Modula-3 uses structural equivalence among types, a unique name for T should be used when translating to my notation. For a discussion of Modula-3's partially opaque types, see Remark 8.1.

8.6.1 C++

Modeling C++ in my notation is similar to modeling Modula-3. However, C++ uses some different terminology that is worth explaining in order to avoid confusion.

In C++, an object type, called a *class*, is a record type (a "`struct`") with extra features. Thus, what I consider an object is, in C++, a *pointer* to a C++ class. Then, my objects are like C++ class pointers as long as all such pointers are obtained via `new`. A class declared as a local variable is modeled as a record. But taking the address of such a variable, like taking the address of anything else, introduces aliasing that I don't handle (see Section 7.2.0).

So, as my objects correspond to pointers to records in C++, why do I not model objects and data fields that way, too? Subtypes may increase the number of data fields that an object has, and different subtypes may have different data fields. Using a map for each data field provides the flexibility required for this task—data fields can be added arbitrarily and can be added only to those subtypes for which they exist. Simplicity is another reason, because having the index set of the data record vary as a function of the object that is used to get to the data record becomes clumsy and awkward.

What I have called a method, C++ calls a *virtual* method. What C++ calls a (non-virtual) method, I simply view as a procedure, because it cannot be replaced in subtypes.

Finally, a note on protection levels. C++ features three data field protection levels, `private`, `protected`, and `public`. These levels enforce an access discipline and do not affect the semantics of the data fields and methods that can be accessed.

Chapter 9

Abstraction

In this chapter, I introduce the concept of *data abstraction*, first for general variables and later for data fields. In Section 9.1, I introduce *data refinement*, an introduction that stays close to the way it was done originally by Hoare [38]. In Part III, however, where programs consist of modules, the techniques discussed in Section 9.1 will not suffice. Nevertheless, this chapter provides a flavor of what data abstraction and refinement are all about and why they are of interest.

9.0 Abstract variables

So far, each variable we have seen has represented one coordinate in the program state space. Let us now consider functions over these variables. For example, if x and y are program variables, we may consider the function $x + 2 \cdot y$. We may name such a function by introducing an *abstract* (or *specification*) *variable*.

The declaration

spec var z

introduces an identifier z, and declares it to be an abstract variable.

> **Remark 9.0.** I only introduce *global* abstract variables, because they are the ones that are needed when writing modular specifications (see Part III). All concepts apply to local abstract variables as well—all that's needed is a notation for declaring such variables.

An abstract variable is not a coordinate in the state space that is changed independently of other variables (*cf.* Section 1.1). Rather, it is simply a function of regular variables (the latter hereinafter called *program* or *concrete* variables). When the values of these program variables change, so does the value of the abstract variable, and *vice versa*.

> **Remark 9.1.** A program variable, too, is a function. It abstracts a meaningful value from the bits that physical machines have. However,

since the program variables are independent of each other, one such bit represents part of only one program variable. (Each bit, also, is a function of more concrete entities such as the components of a virtual memory system, and they, in turn, are functions of voltages, and so on; in this thesis, I do not get more concrete than program variables.)

To specify what the value of z is, with respect to other variables, a **rep** declaration is used. It has the form

$$\textbf{rep}\ z\ \textbf{is}\ R \quad ,$$

where z names an abstract variable and R, called the *representation* of z, is a predicate involving z. The representation specifies the value of z. For example,

$$\textbf{rep}\ z\ \textbf{is}\ z = x + 2 \cdot y$$

defines z to be the function $x + 2 \cdot y$.

For expressiveness, R is a predicate rather than the function itself. This allows a declaration like

$$\textbf{rep}\ F\ \textbf{is}\ \ 9 \cdot (C + 40) = 5 \cdot (F + 40)$$

instead of forcing the formula to be written as

$$F = 9/5 \cdot C + 32 \quad .$$

Nevertheless, I require that R specify z uniquely from the program variables. The jargon is that R is an *abstraction function*, as opposed to an *abstraction relation* (or *nondeterministic function* [72]), which would only specify z down to a set of possible values. I restrict my attention to abstraction functions so as to avoid issues such as, "Does $z = z$ hold?".

Remark 9.2. Interestingly enough, the formal proof of soundness in Chapter 12 does not rely on representations being abstraction functions; abstraction relations work just as well. However, with abstraction relations, Chapter 11 needs revamping.

It is common, however, for several values of the concrete variables to yield the same value of the abstract variable. This justifies the name *abstract* variable, since it abstracts away from some details of particular states.

Remark 9.3. The names *specification* variable *vs. program* variable stem from the fact that the former typically exists only in specifications whereas the latter is compiled and takes up memory like the rest of the program.

I also assume that R is *total* in the variables that represent z. That is, I assume that for every state that ever occurs in an execution of a program, the value of z is defined.

An abstract variable need not be defined only in terms of program variables; it can also be defined in terms of other abstract variables. For example, for abstract variables a and b,

$$\textbf{rep}\ a\ \textbf{is}\ a = 3\cdot b + 2\cdot x\ ;$$
$$\textbf{rep}\ b\ \textbf{is}\ b = x + y$$

in effect defines a to satisfy

$$a = 5\cdot x + 3\cdot y \qquad .$$

9.1 Abstract variables and refinement

In this section, I treat refinements involving abstract variables. Consider the following example.

$$\textbf{var}\ c\ ;$$
$$\textbf{spec var}\ a\ ;$$
$$\textbf{rep}\ a\ \textbf{is}\ a = c^2$$

The statement $a := 9$ thus has the effect of setting c to either −3 or 3. Therefore, the statement $c := 3$, which always sets c to 3 and never to −3, is a refinement of $a := 9$ (*cf.* Section 1.8).

A refinement like this is called a *data refinement* [38]. I proceed to show how such a refinement is often proven (*cf.* [38, 30, 43, 3]). Then, I briefly mention why data abstraction and refinement are important, and why the classical view of data refinement is too restrictive for doing abstraction in modular programs.

9.1.0 Classical data refinement

The idea is to consider two state spaces, one containing the abstract variables and the other containing the concrete variables. Within each state space, the respective variables are considered to be independent coordinates. The two state spaces are related by the predicate that defines the representation of the abstract variables.

A new command, *SwitchToAbstract*, is introduced. It is defined by the following weakest precondition. (The presence of exceptions is tangential to this discussion, so I assume programs whose outcome is always normal.)

$$wp.SwitchToAbstract.Q \;=\; \langle \forall\, a \mid R \triangleright Q \rangle$$

where a is the list of abstract variables whose representation is prescribed by R, and Q is a predicate over the abstract variables.

Informally, *SwitchToAbstract* switches from the concrete partition to the abstract partition, and thus sets the values of the abstract variables according the values of the concrete variables and the representation predicate R. *SwitchToAbstract* establishes (abstract) postcondition Q from those (concrete) initial states in which, for each value of a that satisfies R, Q holds. Since I assume that R determines a uniquely, there is at most one such value for a. Moreover, since I assume R to be total in the representation of a, there is at least one value for a that satisfies R. Thus, the interpretation of *SwitchToAbstract* can be reformulated: *SwitchToAbstract* establishes Q from those initial states in which Q holds for the value of a that satisfies R.

A concrete program C is defined to data refine an abstract program A just when

$$\mathit{SwitchToAbstract} \ ; \ A \ \sqsubseteq \ C \ ; \ \mathit{SwitchToAbstract} \qquad ,$$

where $\sqsubseteq$ is defined as in Section 1.8. Both sides of $\sqsubseteq$ show a program that starts in the concrete state space and ends in the abstract state space. The refinement thus compares the left- and right-hand sides with respect to their outcomes in the abstract state space. In other words, the details of the concrete state space matter not; only the abstract representation of these states do.

> **Remark 9.4.** Unlike the data refinement found in the literature, the *SwitchToAbstract* command does not appear in Part III, despite the fact that (or maybe, because) a more general form of abstraction is considered there.

Let's now prove that $c := 3$ refines $a := 9$ in our example. We need to show

$$\mathit{SwitchToAbstract} \ ; \ a := 9 \ \sqsubseteq \ c := 3 \ ; \ \mathit{SwitchToAbstract} \qquad .$$

For any predicate Q, we calculate,

$$\begin{array}{ll}
& wp.(c := 3 \ ; \ \mathit{SwitchToAbstract}).Q \\
= & \quad \{ \ ; \text{ and } \mathit{SwitchToAbstract} \ \} \\
& wp.(c := 3).\langle \forall a \mid a = c^2 \triangleright Q \rangle \\
= & \quad \{ \text{ one-point rule } \} \\
& wp.(c := 3).(Q[a := c^2]) \\
= & \quad \{ \ := \ \} \\
& Q[a := c^2][c := 3] \\
= & \quad \{ \text{ substitution, since } Q \text{ is a predicate over the abstract} \\
& \quad\quad \text{variables and thus does not contain } c \ \} \\
& Q[a := 9] \qquad ,
\end{array}$$

and

$$\begin{array}{ll}
& wp.(\mathit{SwitchToAbstract} \ ; \ a := 9).Q \\
= & \quad \{ \ ; \text{ and } := \ \} \\
& wp.\mathit{SwitchToAbstract}.(Q[a := 9]) \\
= & \quad \{ \ \mathit{SwitchToAbstract} \ \}
\end{array}$$

$$\langle \forall a \mid a = c^2 \triangleright Q[a := 9] \rangle$$
$= \quad$ { one-point rule, since a does not occur free in $Q[a := 9]$ }
$$Q[a := 9] \quad .$$

This proves the data refinement.

9.1.1 Advantages and shortcomings

Data abstraction and refinement are important because they allow us to introduce abstract variables to describe the abstract behavior of the procedures and methods in a module. They also admit a way to represent the properties of objects abstractly, independently of the details of the particular subtypes (see Section 9.2 below).

For example, consider the procedures of a module that implements a file system. Abstract variables are introduced to be used in the procedure specifications that describe the intended behavior of the file system. Later, in the implementation, design decisions are made as to how to represent the file system. At that time, concrete variables are introduced and the representation for each abstract variable is given.

Notice that the abstract variables, which are functions of the concrete variables, are introduced before the exact function definitions are given. In fact, the exact functions of the concrete variables that the abstract variables make up are not known at the time the abstract variables are introduced. The exact mapping is given only later; nevertheless, the mere naming of the abstract variables allows specifications to be written in terms of them.

Also note that *any* mapping to concrete variables (that admits an implementation—and even those that do not [66]) will do, because the details of the concrete representation do not matter to the callers of the procedures. This allows an implementation to change without radiating a new abstract behavior.

Classical data refinement does not permit reasoning about programs where abstract and concrete variables exist together. For example, neither of the programs

$$a := 9 \ ; \ c := c + 1$$

and

$$a := 4 \cdot c$$

makes any sense in the classical data refinement model.

> **Remark 9.5.** When doing data refinement via so-called *auxiliary variables*, statements like these are allowed (see, *e.g.*, [65]).

Instead, the world is divided: either the abstract variables are present or the concrete variables are, but never both at the same time. This gives the view of the program as having exactly two modules, one with access only to abstract variables, the other with access only to concrete ones. My view of a modular program in Part III allows any number of modules. Furthermore, my modules are not separated from each other; rather, a module can include (*import*) others. (This corresponds to the use of *interfaces* and *modules* in Modula-3 and Modula-2, and to the use of *packages* in Ada.) This means that abstract and concrete variables do exist together, a situation that yearns for a solution.

Before entering Part III, let me discuss abstract data fields and partial representations.

9.2 Abstract data fields

In Section 9.0, I introduced abstract variables without saying anything about types. Now, I consider a specific kind of abstract variables—those whose types make them data fields (see Section 8.1).

Consider an object type T and an abstract data field declared by

$$\textbf{spec var}\, a : T^{-} \to A$$

for some type A. Let $T0$ and $T1$ be two subtypes of T, neither a subtype of the other. I allow the representation of a to be different for these two subtypes. For example, if

$$\textbf{var}\, p : T^{-} \to A$$

is a data field of T and A denotes some numeric type, then a can be represented by $a = 1.12 \cdot p$ for $T0$ objects and $a = 1.08 \cdot p$ for $T1$ objects. Each of these is called a *partial representation* of a. They are written

$$\begin{array}{l} \textbf{rep}\, a[t : T0]\ \textbf{is}\ a[t] = 1.12 \cdot p[t] \ ; \\ \textbf{rep}\, a[t : T1]\ \textbf{is}\ a[t] = 1.08 \cdot p[t] \qquad . \end{array} \tag{9.0}$$

The "$[\ldots T0]$" shows that the **rep** declaration only gives the representation of a for $T0$ objects, and the "$t :$" introduces a name (of type $T0$) that can be used in the representing expression, and similarly for $[t : T1]$.

> **Remark 9.6.** It would be more accurate to write "$[t : T0^{-}]$" instead of "$[t : T0]$", but since the index set of a is T^{-}, *i.e.*, $T \setminus \{\textbf{nil}\}$, I take "$[t : T0]$" to mean "$[t : T0^{-}]$", which simplifies the notation slightly.

> **Remark 9.7.** An alternative way of viewing the representation of a (9.0) is
>
> $$\textbf{rep}\, a\ \textbf{is}\ \langle\, \forall\, t : T^{-} \ \triangleright\ a[t] = \begin{cases} 1.12 \cdot p[t] & \text{, if } t \in T0 \\ 1.08 \cdot p[t] & \text{, if } t \in T1 \\ \quad \vdots & \end{cases} \quad \rangle \qquad .$$

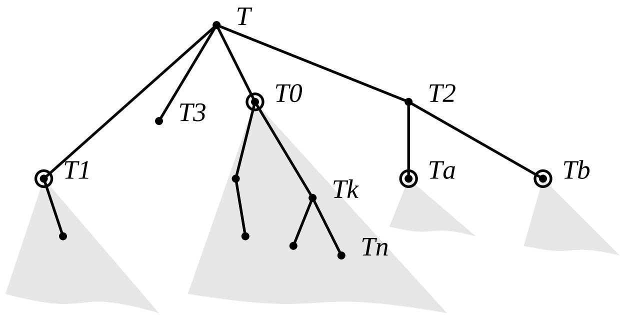

Figure 9.0: Example of partial representations in a type hierarchy

I say that $a[t] = 1.12 \cdot p[t]$ is the representation of a *at* type $T0$.

The representation of a can also be in terms of the data fields of the respective subtypes. For example, if

var $x : T0^{-} \to A$;
var $y : T1^{-} \to A$

are data fields of $T0$ and $T1$, respectively, and A denotes the integers, then a can be represented by $a = 2 \cdot x$ for $T0$ objects and $a = y - 7$ for $T1$ objects.

rep $a[t : T0]$ **is** $a[t] = 2 \cdot x[t]$;
rep $a[t : T1]$ **is** $a[t] = y[t] - 7$

The flexibility of stating different representations for different subtypes comes with a restriction. If a subtype $T0$ provides a representation, then no subtype of $T0$ may provide a different representation. That is, unlike methods, the subtypes of $T0$ may not override the representation provided at $T0$. This can be illustrated with a picture. In Figure 9.0, types are represented as nodes in a tree. Edges connect a subtype with its immediate supertype, and the convention is that supertypes are placed above their subtypes in the picture. Circled nodes indicate that a representation is given at this node. Thus, the shaded regions show subtypes that may not provide their own representation.

This restriction is pronounced in the interest of modular verification (see Part III), where all information about a program may not be available at the time of verification. The restriction then prevents the verification process (be it performed by a human or by a machine) from using the wrong representation. For example, if both $T0$ and Tk (*cf.* Figure 9.0) were to provide representations, then for a scope where types T, $T0$, and Tn are visible, and only $T0$'s (and not Tk's) representation is visible, the verification process might use the representation provided at $T0$ as the prevailing representation for Tn, whereas with full information about the program, the representation Tk would be used.

The object model in [56] does not have this restriction. There, a subtype provides all of its own data fields, *i.e.*, it does not inherit any data fields —and, in particular, inherits no abstract data fields— from its supertypes. The model herein comes closer to objects provided by languages like Modula-3 and C++.

The restriction does not so much restrict as it does provide a guiding methodology for the construction of object-type hierarchies. Consider a type T that declares a data field c. Some supertype of T declares an abstract data field a, and T gives its representation of a in terms of c. If, for some reason, it is expected that subtypes of T would want to provide different representations of a, then one can often split type T into two types, T' and $TImpl$, say, where $TImpl <: T'$. The representation of a previously given at T is now given at $TImpl$. This usually means that c should be declared at $TImpl$, but declaring it at T' is sometimes also a possibility. The aforementioned subtypes of T are now declared subtypes of T', which means they are free to give their own representations of a.

To recap, subtypes of a type that declares an abstract data field can provide their own representations of that field. Each such representation is called a partial representation. Usage of partial representations is restricted to avoid that an object would have more than one representation of some data field. This restriction provides guidance in the design of object-type hierarchies.

Part III

Modularity

Modules and modular verification

In Parts I and II, I treated the semantics of control structures of programs and of data structures that those programs manipulate. In this Part, I consider making the verification of large programs feasible.

The verification of a program consists of the verification of a set of refinements. Each such refinement can be verified independently. In what I have presented, it is always specifications that are refined. To make the mathematics work *for* us in a large program, it is therefore crucial that we know how to write specifications in a large program. For example, just because one procedure calls several others, need the specification of one involve all the details of the others being called? We hope not, because if this were the case, writing the specification for a procedure in a program with tens or hundreds of thousands of lines of code would be too complex a task for us to manage. Luckily, we are armed with the tool of abstraction (see Chapter 9), with which there is hope to hide the complexity of the implementation details at various levels.

A mechanism used as an aid in abstraction and data hiding is *modularity*. By organizing the code into separate *modules*, we are able to hide implementation details of that module from other parts of the program.

Modularity lends itself to separate compilation of modules. Not only can this be a time-saving device when a small change is made in a program, but it also allows code to be written by different groups or vendors and collected into *libraries* without these groups knowing the details of the code in the other groups' modules.

Similar to the concept of separate compilation of modules is the concept of *modular verification*—that each module can be verified in isolation from the verification of other modules. Modular verification not only saves time, but is essential to enable libraries to be verified without knowing the details of the programs into which they eventually will be linked. To perform modular verification, we need to learn how to write specifications for procedures that appear in modules.

In this Part, I deal with specifications and modular verification. Having these under control is vital to making the verification of large programs feasible.

Outline

In Chapter 10, I describe the basic problem with abstraction within a setting of an arbitrary number of modules when modular verification is important. I also give a solution to this problem, and discuss related work. In Chapter 11, I give a formal description of this solution for a simple language with modules. In Chapter 12, I prove that the solution in Chapter 11 is *sound* with respect to modular verification, *i.e.*, that a module verified to be correct using the detailed technique shown in Chapter 11 would also be verified to be correct had all information about the program been taken into account in the verification. In Chapter 13, I discuss some remaining problems.

Chapters 10 and 13 are of interest to those readers concerned with abstraction, multiple modules, and modular verification. Chapter 11 is of interest to those readers who want to apply the methods of Chapter 10 in practice, for example by building a formal verification system that is unforgiving when it comes to having all the details. Chapter 12 targets a smaller audience, and is provided for completeness. It is intended for semanticists with a thirst for detailed understanding, possibly because they are facing similar proof obligations, and for those who like formal proofs in their own right. However, even the readers with an interest in only Chapters 10 and 13 may be curious to read Section 12.5, which comments on the shape of the proof.

Chapter 10

Specifications in modular programs

In this chapter, I describe a specification problem that arises in the context of modular verification of programs with many modules. I proceed concretely by showing a programming example in which the problem surfaces. Since the goal in this Part is to make program verification feasible in practice, I draw my example from an attempt to write the specification for a real library of input/output streams [9], a library written in a modular, object-oriented style.

After some motivation, I describe the problem, introducing the necessary concepts along the way. Then, I describe a solution to the problem, followed by some discussion. The solution can be viewed as a generalization of classical data refinement [38]. I conclude this chapter by making a connection between the two, and by comparing my solution with other specification languages.

I try to keep the discussion at as high a level as possible while exposing enough details to reveal the problem. A precise description of the solution is given in the next chapter. I assume some familiarity with the concepts of modules and interfaces (or packages) from languages like Modula-3 and Modula-2 (or Ada, respectively).

10.0 Motivation

Programs written in a programming language like Modula-3 are divided into *modules* and *interfaces*. A module or interface *imports* another interface in order to gain access to the entities declared in that interface.

A procedure is declared in an interface, and its implementation is given in a module. This hides the private data of the module from the clients of the interface. The implementation of a procedure declared in an interface may have an effect on the private data in the module. I use *data abstraction* to combat this problem: The interface describes an abstract view of the behavior of the procedure; the module provides the implementation and prescribes the relation between the abstract view and the concrete one.

As described in the preface to Part III, we are interested in *modular verification.* That means that one should be able to verify an implementation given only its module and the module's imported interfaces. Having the entire program in view at one time simplifies verification, but is unreasonable to require, because, for example, then a library could not be verified until it were linked with a complete program.

Specifications play a central rôle in this chapter. Recall from Sections 6.0.0 (and 1.7) that a specification includes a *frame* (given by a **modifies** clause), which lists those variables that are allowed to be modified. Without **modifies** clauses, a procedure would be able to modify *anything*, just as long as the postcondition were met. For example, consider the specification given by $Q, Q0, Q1$ in Chapter 5. Without any notion of what the program to be developed is allowed to modify, a perfectly valid implementation would be

$$a[0,0], x, i, j, b := 0, 0, 0, 0, true \quad ,$$

or, if M and N were not given as constants,

$$M, N, b := 0, 0, false \quad .$$

The lack of a construct like **modifies** yields specifications that are too weak to be useful in the setting of an entire program. Hence, **modifies** clauses are important.

> **Remark 10.0.** The specification language Anna [57] does not feature **modifies** clauses. When specifications are interpreted by a person (rather than by an unforgiving machine), several conventions are understood (or misunderstood, as it may be). For example, maybe if *Swap* from Section7.3 were specified only by
>
> **ensures** $a[i] = a_0[j] \wedge a[j] = a_0[i] \quad ,$
>
> a programmer "knows" not to modify a at any index other than i and j. Thus, Anna is useful for formal documentation of a program. Anna annotations admit translation into checks that can be executed at run-time. These are an important and useful aid in finding errors already introduced into a program. They do not, however, establish the absence of errors in a program. For that, verification is needed, so for that, Anna is not, without further restrictions, suitable.

10.1 Problem

The problem I'm about to show arises in the presence of three things: data abstraction, *friends interfaces* —which I will describe below—, and modular verification. I won't discriminate between modules and interfaces, because their distinguishing characteristics are not material to the discussion. Instead, I will refer to either as a *unit.*

10.1.0 Writer example

The example consists of four units. The first, named *Wr*, shows the declaration of a *writer* class. A writer is an output stream. Examples of writers —that is, of writer subtypes— are *file writers*, which write their output stream to a file in a file system, and *text writers*, which write their output to a text string in memory.

Here's the first unit.

unit *Wr* **is**
  type *T*;
  spec var *target* : $T^{-} \to$ **seq**[**char**];

  spec *PutChar*(*wr* : *T* ; *ch* : **char**) **is**
    modifies *target*[*wr*]
    ensures $target[wr] = target_0[wr] +\!\!+\ ch$
end

Unit *Wr* introduces a type *T* (see Section 7.2), and declares a specification (abstract) variable *target* (see Chapter 9). *target* is of type $T^{-} \to$ **seq**[**char**] ; hence, it designates an (abstract) data field (see Sections 8.1 and 9.2).

seq[**char**] is a type whose values are sequences of characters. Readers familiar with Modula-3 may think of **seq**[**char**] as Modula-3's `TEXT` type, but the exact nature of **seq**[**char**] is not central to the discussion. I use $+\!\!+$ to denote concatenation.

Unit *Wr* also shows the specification of a procedure, *PutChar*, which takes as parameters a writer and a character. The procedure modifies the target of the writer and ensures that the target, upon termination of the procedure, equals the initial target extended with the given character.

Here's the second unit.

unit *WrFriends* **import** *Wr* **is**
  var *buff* : $Wr.T^{-} \to$ **seq**[**char**]
  (∗ *Wr.target*[*wr*] = flushed characters of $wr +\!\!+\ buff[wr]$ ∗)
end

Unit *WrFriends* is what is known as a *friends interface*. It gets that name from the fact that the unit is intended for import by other units with a close tie to the implementation of writers—a tie only "good friends" are thought to have. (In [9], *WrFriends* is called `WrClass`.)

Unit *WrFriends* imports unit *Wr* to make the declarations in *Wr* visible in *WrFriends*. The import relation is transitive—that is, by importing *Wr*, *WrFriends* also imports all units imported by *Wr* (if there were any). The *scope* of a unit is the set of declarations given in that unit and the units that it imports.

WrFriends declares a data field *buff*. Note that I have prefixed type *T* from unit *Wr* with "*Wr.*" since it is imported from another unit, a practice common among languages with modules. Note also that *buff* is not a specification variable, but a program variable.

The unit ends with a comment describing to programmers the intended usage of field *buff*. This comment is vital to the discussion; in a sense, the problem I am describing is the problem of formalizing this comment. The idea is the following. The target of each writer is made up of a *flushed* portion and a *buffered* portion. Different writers store their flushed portion in different ways. For example, a file writer keeps the flushed portion on disk, whereas a text writer keeps it in a string in memory. This is the essence of object-oriented programming; different subtypes define their own ways of dealing with this aspect of being a writer. All subtypes share a mechanism for the buffered portion. For that purpose, *buff* has been introduced. Procedure *PutChar*, then, simply adds the given character to the end of *buff* for the given writer. If *buff* becomes too large as the result of such an operation, *PutChar* calls out to the particular subtype to perform a flush, *i.e.*, to copy the value of *buff* into the flushed portion of the writer and clear *buff*. The call to the particular subtype is done via a method invocation to some *flush* method, not shown here.

Now, let's move on to a unit declaring a particular kind of writer: text writers.

$$
\begin{array}{l}
\textbf{unit}\ \mathit{TextWr}\ \textbf{import}\ \mathit{Wr}\ \textbf{is} \\
\quad \textbf{type}\ T <: \mathit{Wr.T}; \\
\quad \textbf{spec}\ \mathit{Init}(\mathit{wr} : T)\ \textbf{is} \\
\quad\quad \textbf{modifies}\ \mathit{Wr.target}[\mathit{wr}] \\
\quad\quad \textbf{ensures}\ \mathit{Wr.target}[\mathit{wr}] = \text{``''}\ ; \\
\quad \textbf{spec}\ \mathit{result} : \textbf{seq}[\textbf{char}]\ :=\ \mathit{Target}(\mathit{wr} : T)\ \textbf{is} \\
\quad\quad \textbf{ensures}\ \mathit{result} = \mathit{Wr.target}[\mathit{wr}] \\
\textbf{end}
\end{array}
$$

Unit *TextWr* declares a type T as a subtype of *Wr.T* (see Section 8.0). That is, objects of type *TextWr.T* make up some of the objects of type *Wr.T*.

Unit *TextWr* also declares two procedures, *Init* and *Target*. *Init* clears the target of a text writer. Its specification states that the target of the given text writer is modified to ensure that it equals the empty string upon termination. Procedure *Target* returns the target for a given text writer. It does not modify anything in the process.

The fourth and final unit shows the implementation of text writers. In a language like Modula-3, this unit would be a module. However, since I do not distinguish between modules and interfaces, this is simply another unit, whose name I choose to be *TextWrImpl*.

$$
\begin{array}{l}
\textbf{unit}\ \mathit{TextWrImpl}\ \textbf{import}\ \mathit{Wr}, \mathit{WrFriends}, \mathit{TextWr}\ \textbf{is} \\
\quad \textbf{var}\ \mathit{flushed} : \mathit{TextWr.T}^{-} \rightarrow \textbf{seq}[\textbf{char}]; \\
\quad \textbf{rep}\ \mathit{Wr.target}[\mathit{wr} : \mathit{TextWr.T}]\ \textbf{is} \\
\quad\quad \mathit{Wr.target}[\mathit{wr}] = \mathit{flushed}[\mathit{wr}] +\!\!\!+\ \mathit{WrFriends.buff}[\mathit{wr}]\ ; \\
\quad \textbf{impl}\ \mathit{TextWr.Init}(\mathit{wr} : \mathit{TextWr.T})\ \textbf{is} \\
\quad\quad \mathit{flushed}[\mathit{wr}] := \text{``''}\ ;\ \mathit{WrFriends.buff}[\mathit{wr}] := \text{``''}\ ;
\end{array}
$$

impl *result* : **seq**[**char**] := *TextWr.Target*(*wr* : *TextWr.T*) **is**
  result := *flushed*[*wr*] ++ *WrFriends.buff*[*wr*]
end

This unit imports all of the previously introduced units. It declares a variable *flushed*, whose purpose is to contain the flushed portion of text writers. Hence, the representation of *target* for text writers can be given, as stated by the **rep** clause (see Section 9.0): *target* of a text writer is the concatenation of *flushed* and *buff* for that writer.

Unit *TextWrImpl* also gives the implementation of procedures *Init* and *Target*. *Init*, which is supposed to set *target* to the empty string, sets both *flushed* and *buff* to the empty string for that writer, the concatenation of which is the empty string. Procedure *Target*, which is supposed to return *target* of the given text writer, simply returns the concatenation of *flushed* and *buff* for that writer.

10.1.1 Frames and abstract variables

Now that I've shown the example, I am able to raise a question. Why is it that procedure *Init*, which is specified to only modify *Wr.target*, is allowed to modify variables *flushed* and *WrFriends.buff*? From Chapter 9, we answer, "Because *flushed* and *WrFriends.buff* are part of the representation of *Wr.target*".

Having decided that, we find ourselves at the edge of the problem. Consider the following client unit, which imports *Wr*, *WrFriends*, and *TextWr*—that is, all the units seen so far except the text writer implementation.

unit *FaultyClient* **import** *Wr*, *WrFriends*, *TextWr* **is**
  ⋮
  TextWr.Init(*wr*); (∗ *Wr.target*[*wr*] = "" ∗)
  WrFriends.buff[*wr*] := . . . ;
  if *TextWr.Target*(*wr*) ≠ "" **then** *wrong* **fi**
  ⋮
end

This faulty client calls *TextWr.Init* for some text writer *wr*. After that call, we can conclude that the target of this writer is the empty string. To remind ourselves of that, I have shown this as a comment in the code. Although it is not visible in this scope, the fact that the representation of *target* equals, for text writers, the concatenation of *flushed* and *buff* means that *flushed* and *buff* each equals the empty string at this point in any execution.

The next statement mucks with this writer's *buff* field. Hence, this statement actually affects the value of *target*[*wr*]. But this fact goes unnoticed to the verification process, to which *target*'s **rep** clause is not visible. Therefore, the modular-verification process treats the update of *buff* as having no effect on *target*.

The last line of the code compares the result value of *TextWr.Target*(*wr*), *i.e.*, the value *target*[*wr*], with the empty string. If the update of *buff* has no effect on *target*, which, recall, is what a modular-verification process would conclude from the given information, then $target[wr]$ and "" will be equal, and the branch that goes wrong is not taken. However, full information about the program —in particular, having full information about the representation of $target$ for text writers— reveals that the update of $buff$ does affect the value of $target$, and thus the program will go wrong.

We have reached the climax of the exposition. The question we're facing is: What, in a modular verification of this program, should prevent the program from being verified to be correct?

10.2 Solution

Now that we understand what the problem is, let me move on to its solution.

The solution is to introduce a new specification construct called **depends**. This will allow dependencies between variables —that one variable is part of the representation of another— to be revealed. The clause

depends a **on** c

reveals that abstract variable a may be represented in terms of variable c.

The **depends** construct allows a programmer to give *part* of the representation of an abstract variable. **depends** does not state what the representation *is*, but reveals a variable on which it depends.

> **Remark 10.1.** Modula-3 programmers familiar with partially opaque types may find that the relation between partial and full type revelations is similar to that of **depends** and **rep** clauses.

The use of an abstract variable in a frame can now be defined. I define an abstract variable listed in a **modifies** clause to be a shorthand for also listing the variables on which the abstract variable depends. In other words, the actual frame is the reflexive transitive closure of the frame given by the programmer. It is sometimes convenient to call this closure the *downward* closure, to indicate that the closure goes toward the more concrete (or *down*-to-earth) representation.

I require that all variables on which an abstract variable's representation depends be given in **depends** clauses. Thus, in order to mention a variable c in the **rep** of an abstract variable a,

depends a **on** c

must be visible (or deducible by transitivity) in the scope in which the **rep** appears.

Finally, a predicate $Pred$ that mentions an abstract variable a (here written $Pred(a)$) is interpreted as the same predicate with a replaced by a function instantiation. The function, which here I shall call a', is a function of its dependencies. Thus, $Pred(a)$ is defined to mean

$$Pred(\ a'(c, \ldots)\ ) \qquad .$$

Remark 10.2. There is an important detail, coined *residues*, that plays a rôle here. I postpone discussing this detail until Chapter 11.

10.2.0 Correcting the example

Let me illustrate how **depends** solves the problem in the example I introduced earlier. Since the representation of *Wr.target* involves *WrFriends.buff*, *i.e.*, *Wr.target* depends on *WrFriends.buff*, we add the line

depends $Wr.target[t : Wr.T]$ **on** $buff[t]$

in unit *WrFriends*. This states that, for every writer t, $target[t]$ depends on $buff[t]$.

Similarly, in unit *TextWrImpl*, we add the line

depends $Wr.target[t : TextWr.T]$ **on** $flushed[t]$.

It discloses that, for every *text writer* t, $target[t]$ depends on $flushed[t]$.

As these dependencies are in the scope of *TextWrImpl*, the **rep** clause there is permitted to mention *buff* and *flushed*. Furthermore, because of the **depends** clause in *WrFriends*, the faulty client's update of *buff* is perceived as an update of *target*. The precise effect on *target* is, however, unknown in the scope of *TextWrImpl*—all that is known is that an update of *buff* may cause the value of *target* to change. Hence, the faulty client will no longer verify.

10.2.1 Visibility requirement

Let us consider some restrictions that apply in the use of **depends** clauses. Certainly, there must be *some* restriction, because otherwise all **depends** clauses could be written in some distant unit that almost never is imported, and then these clauses would do no good given our goal of modular verification.

To withstand this problem, I require that the dependency between two variables be visible wherever both of those two variables are. I call this rule the *visibility requirement*.

10.2.2 Benevolent side effects

Consider the following couple of units.

```
unit D is
   spec var valid ;
   spec var state ;
   spec P() is
      requires valid
      modifies state
end
```

```
unit DImpl import D is
   var hash_table, start, n, ... ;
   depends valid on hash_table, start, n, ... ;
   depends state on hash_table, start, n, ... ;
   rep ... ;
   impl P() is ...
end
```

These two units show a common paradigm (see also Chapter 13), *viz.*, using, in addition to specification variables describing the values provided by interface D (here, simply and generally called *state*), a specification variable *valid*. *valid* is *true* just when the values of the internal implementation are in a state that "makes sense", *i.e.*, a state which represents a value of *state*. An example of when this might not be the case is before that state is initialized. *valid* is set by some initialization procedure (not shown here) and is required as a precondition by all procedures in the interface.

> **Remark 10.3.** Some consider *valid* the *object* or *module invariant* and make it implicit (*cf.* [38, 64, 57, 56]). In the presence of operations that involve many objects, it is not always clear *when* the object invariant is supposed to hold. At the expense of verbosity, giving *valid* explicitly makes it clear at what points the invariant must hold.

Let's discuss the value of *valid* upon exit from P. The specification of P requires that *valid* hold upon entry to P; it also states that only *state*, not *valid*, is modified. Hence, we conclude that *valid* holds upon exit from P.

Very well, let's now focus on the implementation in *DImpl*. Here, the **modifies** clause of P is interpreted by taking the downward closure of *state*, which shows that the variables $hash\_table, start, n, \ldots$ are allowed to be modified. Note that these variables are the representation also of *valid*, and the value of *valid* is not allowed to be changed. This means that the implementation of P is constrained to modify $hash\_table, start, n, \ldots$ only in such ways that the value of *valid* is preserved. In the parlance, P's side effects must be *benevolent*. A way to view this is to add

ensures $valid_0 = valid$

to the specification of P.

10.2.3 Authentic abstractions and variables

Now that I've introduced benevolent side effects, consider the following program unit.

unit *AuthenticityProblem* **import** *Wr*, *WrFriends* **is**
  spec var *a*;
  depends *a* **on** *WrFriends.buff*;
    ⋮
  $(* \; a = A \; *)$ *Wr.PutChar*(wr, ch) $(* \; a = A \; ? \; *)$
    ⋮
end

This unit declares a specification variable called a, and reveals that a depends on *buff*. For simplicity, I have left subscripts off.

From the specification of *PutChar* (Section 10.1.0), we determine that any side effect on a is benevolent. That is, despite the fact that both *target* (which is in the

modifies clause of *Wr.PutChar*) and *a* (which is not) depend on *buff*, *PutChar* does not alter the value of *a*. But to ensure this, *a* (and its representation) must be available at the time the implementation of *Wr.PutChar* is verified. How is this to be guaranteed?

In general, such a guarantee cannot be made. However, if *a* is visible anywhere *buff* is, then the implementation (if it modifies *buff* at all) has both *buff* and *a* in scope. Due to the visibility requirement, the dependency of *a* on *buff* is thus also visible.

> **Remark 10.4.** We don't need to worry about whether or not *a*'s representation is in scope; if it is not, verification of code that modifies *buff* but that must not modify the value of *a* will not go through.

If *a* is indeed visible anywhere *buff* is, then I say that this dependency is an *authentic abstraction.* A specification variable is *authentic* just when all its (downward) dependencies are authentic abstractions. A variable that is authentic is also said to satisfy the *authenticity property.*

Hence, the condition of benevolent side effects can be guaranteed only for authentic variables. Distinguishing between authentic and unauthentic variables can be done (see next section); however, the distinction is maybe more easily detected by a machine than by a programmer. An option is therefore to simply rule out unauthentic variables, a rule I give the name *authenticity requirement.* That is, the authenticity requirement states that all variables satisfy the authenticity property.

> **Remark 10.5.** The soundness proof in Chapter 12 uses the authenticity requirement. However, the proof can easily be modified to also allow unauthentic variables. To do this, the definition of benevolent side effects in Section 11.1.4 is changed to project not to specification variables but to *authentic* specification variables. The proof is then modified in Sections 12.3.1 and 12.4.1 to use, instead of the authenticity requirement, the fact that all variables in *bb* satisfy the authenticity property.

10.3 Enforcing the requirements

Having posed two requirements on modular programs, a natural concern is: Can these requirements be enforced? The answer is simple: By following a simple convention, both of the requirements follow.

Let me recap the two requirements and state them with respect to two variables *a* and *c*.

Visibility requirement If *a* depends on *c*, then this dependency must be visible anywhere both *a* and *c* are.

Authenticity requirement If *a* depends on *c*, then *a* must be visible anywhere *c* is.

Observe that, if the dependency of a on c is declared either in the unit that declares a or in the unit that declares c, then the visibility requirement follows—then, any unit that imports both a's unit and c's unit also imports the dependency. If declared in a's unit, the dependency is *not* authentic; if declared in c's unit, the dependency *is* authentic. Thus, in conjunction, the two requirements can be stated as one.

> Declaration **depends** a **on** c is placed in the unit that declares c.

This is a condition that is easy to check using the declarations only of the unit being verified and its imports.

> **Remark 10.6.** The convention is a sufficient condition for the two requirements to hold. However, in languages where cyclic imports are allowed (this excludes, *e.g.*, Modula-3), the requirements can be satisfied without adhering to the convention.

> **Remark 10.7.** Not only does the convention capture the two requirements concisely, it is also easier to teach to a programmer than the two requirements are separately. However, I do not regret having introduced the two requirements separately, because of the now manifest opportunity to replace each independently (see, *e.g.*, Remark 10.8) and because of their separate rôles in the soundness proof (see Section 12.5 and Remark 10.5).

10.4 Soundness of modular verification

I introduced **depends** and motivated two requirements for its use. This invites the question: Are these two requirements enough? To answer that question positively, we must prove the *soundness* of modular verification with respect to these requirements. This means that if the verification of each unit goes through, then the verification of the whole program would go through, provided the program satisfies the two requirements.

Such a result allows the verification of a procedure implementation with respect to its specification to be performed in the scope of the unit in which the implementation occurs, *i.e.*, using only the information from that unit and its imports.

Since soundness does hold (see Chapter 12), one may wonder about completeness. Completeness means that if the program could be verified correct given *all* its units at once, then each unit can be verified correct by itself. There is no hope of achieving this. For example, in the faulty client example I showed, if the line that mucks with *buff* actually sets *buff* to the empty string, then *target* remains unchanged. But the only way to determine that *target* indeed remains unchanged is to have a's **rep** clause in scope, and requiring that violates the essence of data hiding.

So, it is not completeness in which we're interested. Instead, we're interested in *adequacy*. That is, we want to be able to specify and verify programs we care about. I revisit the issue of adequacy in Chapter 13.

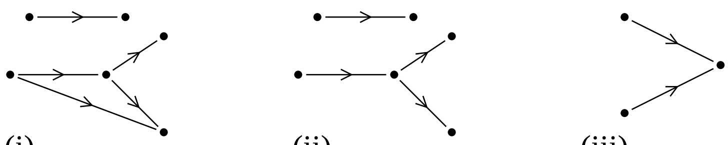

Figure 10.0: Example dependency graphs

Remark 10.8. For the record, this remark discusses a predecessor of the authenticity requirement that proved to be inadequate.

Before inventing the authenticity requirement, I was using what I called the *forest requirement*. Consider the graph whose vertices are the variables and whose directed edges correspond to the given dependencies among variables. Then, the forest requirement states that this graph is acyclic and that the transitive reduction of the graph is a forest. The transitive reduction of a directed graph G is the graph with the fewest edges among those graphs whose transitive closure equals the transitive closure of G.

> **Remark 10.9.** A transitive reduction of a graph is unique if the graph is acyclic.

A forest is a set of trees, and a tree is a directed graph in which every vertex has in-degree at most one (see, *e.g.*, [53]).

For example, Figure 10.0 shows three graphs. Graph (ii) is the transitive reduction of graph (i), and graph (iii) is its own transitive reduction. Graphs (i) and (ii) satisfy the forest requirement, whereas graph (iii) does not.

I had to give up the forest requirement, because it proved inadequate. For example, the dependencies of the common paradigm explained in Section 10.2.2 take the form of Figure 10.0(iii). (The specification language Aspect [41], in some sense, can only handle dependencies satisfying the forest requirement—see its Section 8.1.)

10.5 A generalization of classical data refinement

In this section, I compare my **depends** solution with classical data refinement [38] (see also Section 9.1).

10.5.0 Modeling classical data refinement

Any program specified and refined with the techniques of [38] can also be specified and refined in my model, because [38] is essentially just a restriction on my model:

There are just two units, one for the specification and one for the implementation. Call these S and M, respectively. S then declares all abstract variables, and gives the specifications in terms of those. M imports S, declares the concrete variables, and specifies the representation of the abstract variables. In this simple world, the dependencies can be inferred from the **rep** clause. Finally, the implementation only refers to concrete variables

10.5.1 Explicit functions

Like [38], I treat abstract variables as functions. However, an important distinction is that I prove refinements by making these functions explicit. Classical data refinement instead introduces the program *SwitchToAbstract* and does the dance with two state spaces. It is also crucial that the visible dependencies are shown explicitly. (The fact that dependencies that are not in scope need not be present is proven by the soundness proof in Chapter 12.)

For example, consider variables a and c, where a depends on c. If dependencies are not shown explicitly, then calculating the weakest precondition of a statement $c := 2$ with respect to a postcondition $a \geq 0$ is done as follows. (For simplicity, I leave off the exceptional postcondition in this discussion.)

$$
\begin{array}{ll}
 & wp.(c := 2).(a \geq 0) \\
= & \qquad \{ \;\; := \;\; \} \\
 & (a \geq 0)[c := 2] \\
= & \qquad \{ \;\; \text{substitution} \;\; \} \\
 & a \geq 0
\end{array}
$$

Notice that, without any further information, the substitution in the last step is performed on the basis that c does not appear the expression $a \geq 0$. The last line of the calculation is not the desired result, because an update of c affects the value of a. Thus, proving a refinement this way calls for some other measure, like the business with *SwitchToAbstract*.

In contrast, translating a into a function a' with dependencies shown explicitly yields the following calculation.

$$
\begin{array}{ll}
 & wp.(c := 2).(a'(c, \ldots) \geq 0) \\
= & \qquad \{ \;\; := \;\; \} \\
 & (a'(c, \ldots) \geq 0)[c := 2] \\
= & \qquad \{ \;\; \text{substitution} \;\; \} \\
 & a'(2, \ldots) \geq 0
\end{array}
$$

Note that not only is the last line the expression we want, but both the weakest precondition of an assignment and the rules for substitution are the same as those we would use to prove programs without abstraction.

So, in summary, instead of changing the notion of refinement, predicates are converted with respect to visible dependencies. Then, refinement, assignment, and substitution are those that we are used to.

10.5.2 Inferring dependencies

Languages like Modula-3, Ada, and Modula-2 provide a more flexible model of units than Simula [14], on which [38] is based. These languages allow the state of an implementation to be distributed across multiple units. Only when all of these units and the representation function of an abstract variable are visible can the dependencies of the abstract variable be inferred. Since a unit, in general, imports but a proper subset of these units, automatically inferring dependencies is no longer possible. The **depends** construct solves this problem.

10.6 Other specification languages

In this section, I discuss how some existing specification and programming languages deal with the problem I have described.

10.6.0 Anna

The Anna specification language [57] does not provide a construct like **modifies**; hence, it is not equipped, as is, to handle the kind of verification that I have described (see Remark 10.0).

Moreover, Anna allows the body of a procedure to use a different specification than the so-called *visible* declaration given at the procedure declaration. The motivation for this is that the procedure declaration appears in the package declaration and the procedure body appears in the package body. Since the scope of the body contains the private data of the package, one may want in the package body to extend the visible specification so that it also specifies the behavior in terms of the private data. Anna lets the conjunction of the postconditions appearing in these two specifications be the prevailing postcondition. This is sound, because the implementation is allowed to provide a more specific behavior than originally specified. Stated differently, this is sound because a specification, as a predicate transformer, is antimonotonic in its postcondition. However, Anna also takes the prevailing precondition to be the conjunction of the two given preconditions. That is not sound, because the specification is monotonic in its precondition, not antimonotonic—external clients of the procedure have no way to establish the strengthened postcondition, let alone know what it is.

Finally, Anna provides a mechanism called *package states*. It is used as an abstraction mechanism. Anna requires that package states be used only for packages that satisfy the *Hidden State Principle*, which essentially states that the package implementation cannot make use of the program state outside the package (this includes using, directly or indirectly, global variables declared in other packages). Thus, this kind of data abstraction is like that in classical data refinement.

Any *one* of these three shortcomings means that Anna does not provide a solution for the problem I describe in this chapter.

10.6.1 Penelope

Penelope [32, 62] is an interactive environment for developing and verifying Ada programs. It is based on Larch [33] but borrows much of its syntax from Anna. Penelope's verification is sound across modules. However, in addition to not supporting pointer types (and Ada doesn't feature object types), Penelope does not support abstraction and thus ducks the problem altogether.

10.6.2 CLU

Abstraction in CLU (see [55]) is done under the assumption that the implementation does not "share" its values (so-called *objects*) with clients of the interface. Such sharing is referred to as *rep exposure*. That phenomenon is easily produced, however, and then verification is no longer sound. Similar issues are treated in Section 13.2.

CLU, like Simula and classical data refinement, does not provide language features that allow an implementation to be distributed across modules; hence, friends interfaces are not even a possibility.

10.6.3 Summary

Both Anna and Penelope dodge the problem I have described, because they do not even provide the necessary specification features in the presence of which the problem arises. Both Anna and Penelope are, of course, useful in their own rights, but neither can claim to provide an answer to specifying and verifying modular programs.

What both Anna and Penelope provide, however, which I have not discussed, is the *theory packages* (see [33]) one needs to write elaborate specifications, *e.g.*, a library of axioms and theorems regarding tree structures. These issues, and the issues of how theorems are proven automatically in the presence of these, are orthogonal to my discussions. Both Anna and Penelope aim at achieving full specifications. With extended static checking as one's goal, theory packages play a diminished rôle.

Chapter 11

Generating verification conditions

In this chapter, I formalize the idea of **depends**. I do so by first introducing a simple programming notation for units (modules and interfaces). Then, I define the relation *Refine*, which is true just when a program implements (refines) its specification. The Prolog-style definition of *Refine* suggests a precise operational way to generate the formulas, known as *verification conditions*, that need to be proven in order to establish the correctness of the refinement.

The previous chapter provides an informal discussion of what this chapter formalizes, but with one notable exception: *residues*. Residues are swept under the rug in Chapter 10, because they would have diverted attention from the central ideas in that chapter. In this chapter, I do include residues; in fact, I conclude this chapter by showing the importance of residues. The next chapter, which proves the soundness of modular verification, builds on the definitions presented in this chapter.

The reader will notice that this chapter changes gears to more formality from the previous chapter. In that sense, this chapter also serves as a preparation for the next chapter, which contains the formal proof of soundness. Another change from the previous chapter is that, in order to focus on the relevant details, I have left out many bells and whistles from the richer notation used in Chapter 10.

11.0 A notation for modular programs

In this section, I present the syntax of a programming notation with units. I also give some definitions, many of which are review from previous chapters. The reason for showing the syntax of the notation is that this makes everything explicit and lends itself to inductive definitions of some of the relations defined in the rest of the chapter.

A program consists of a number of *units*.

> **Remark 11.0.** Programming languages like Modula-3 typically provide two kinds of units, modules and interfaces. Since I do not distinguish between the two, a Modula-3 program is a restriction of what I present.

The syntax of each unit is given by the grammar

<*Unit*> ::= **unit** <*id*> [**import** <*idlist*>] **is** {<*Decl*>} **end**

<*Decl*> ::=
  var <*idlist*>
| **spec var** <*idlist*>
| **spec** <*id*> **is** **modifies** <*idlist*> **requires** <*Pred*> **ensures** <*Pred*>
| **impl** <*id*> **is** <*Command*>
| **depends** <*id*> **on** <*idlist*>
| **rep** <*id*> **is** <*Pred*> ,

where {<*Decl*>} means any number of occurrences of <*Decl*>, separated by semicolons.

An <*id*> is an identifier, and an <*idlist*> is a nonempty, comma-delimited list of <*id*>'s. The declarations **unit**, **var**, **spec var**, and **spec** introduce new <*id*>'s. The declared <*id*>'s in a program must be unique.

> **Remark 11.1.** Modular programming languages typically require that top-level identifiers within one unit be unique. Identifiers declared in imported units are then prefixed by the name of the unit in which they are declared, like "*Wr.*" in the examples from the previous chapter. For local variables, variables are distinguished by scope rules. In this chapter, however, I assume that such resolutions have already been done, and that all identifiers are unique.

unit declares the succeeding identifier to be a unit. The <*idlist*> after **import** must list only units. The set of units reachable via imports from a unit A is called the *import closure* of A, denoted $\mathit{ImportClosure}(A)$. Note that $A \in \mathit{ImportClosure}(A)$. Every identifier mentioned in a unit A must be *visible* in A, meaning that it has its declaration in some unit in $\mathit{ImportClosure}(A)$.

Identifiers introduced by **var** or **spec var** are called *variables.* I distinguish between the two kinds of variables by referring to the former as *program* variables and the latter as *specification* variables (see Section 9.0). Identifiers introduced by **spec** are called *procedures.*

The <*idlists*> in **spec** and **depends** must list only variables. The <*id*> in the **impl** clause must be a procedure, and the <*id*>'s in the **depends** and **rep** clauses must be specification variables.

The <*idlist*> in the **spec** clause is called the *frame*, and the two <*Pred*>'s are called the *precondition* and *postcondition*, respectively. The postcondition may mention a variable subscripted with 0, an *initial-value* variable. This refers to the value of the variable upon entry to the procedure (see Section 1.7).

The **impl** declaration gives an implementation of a procedure. A procedure has exactly one implementation. Note that the name of a procedure is declared by a **spec** clause, whereas **impl** just associates an implementation with an already declared procedure identifier. Consequently, the specification of a procedure must be visible in the unit that gives the implementation.

The **depends** and **rep** declarations are described later. For each of these, <*id*> must be a specification variable. Only one **rep** clause per specification variable is allowed. Note that, as with **impl**, the (**spec var**) declaration of the identifier to which **depends** and **rep** pertain must be visible in the unit that declares the **depends** or **rep** clause.

As in Chapter 7, I reserve the character $\star$ in identifiers, so that new unique identifiers needed in the proof can easily be constructed.

Because exceptions are orthogonal to the present discussion, I omit them and consider programs with only one postcondition.

> **Remark 11.2.** Using as that postcondition a *partitioned predicate* (see [61] or Remark 3.3), the discussion also applies to exceptional programs. The addition of *raise* and $\lhd$ then only affects the proof of property (12.12), found in Section 12.4.2.

A guarded command has the following syntax.

<*Command*> ::=
  skip
| *wrong*
| <*id*> “ := ” <*expr*>
| **call** <*id*>
| <*Command*> “ ; ” <*Command*>
| <*Command*> “ □ ” <*Command*>
| <*bool-expr*> “ → ” <*Command*>
| “ ⟦ ” <*id*> “ • ” <*Command*> “ ⟧ ”

The <*id*> in the assignment statement and the identifiers mentioned in <*expr*> and <*bool-expr*> must be program variables. The <*id*> in the procedure call must denote a procedure. The <*id*> in the block statement introduces a new identifier that can be used in the subsequent <*Command*> . This <*id*> must be distinct from all other <*id*> 's in the program.

> **Remark 11.3.** One of the simplifications made from the richer notation previously presented is the absence of loops. Note that programmers can still declare and call tail-recursive procedures.

11.1 Definitions

I now give all the definitions necessary to define *Refine* . I start bottom-up, and will end with the definition of *Refine* itself.

I define an *environment* to be the set of declarations in a set of units closed under imports. Stated differently, an environment is the union of a set of units closed under imports. Most of the definitions take an environment as a parameter.

I treat sets and lists of variables to be synonymous, and do not distinguish between single variables and lists of variables. So, for example, for a list of variables w, the expression $w_0 = w$ is a shorthand for

$$\langle \forall v \mid v \in w \triangleright v_0 = v \rangle \qquad .$$

11.1.0 Dependencies

I start by defining the *dependency relation*, named $Depends$. For any variables a and c and environment E,

$$Depends(a, c, E)$$

holds just when a *depends on* c in E. This means that E contains enough **depends** declarations so that the dependency of a on c is deducible by reflexivity and transitivity.

Note that every variable depends on itself. Also, note that $Depends$ is monotonic in its last argument, *i.e.*, for any a, c, E, E',

$$E \subseteq E' \;\Rightarrow\; (Depends(a, c, E) \Rightarrow Depends(a, c, E')) \qquad . \tag{11.0}$$

11.1.1 *Resolve*

$Resolve$ of a list of variables w and an environment E is the *downward closure* (or *resolve set*) of the variables in w, as is visible in E.

$$\begin{aligned} &Resolve(w, E) = \\ &\qquad \{ v, x \mid v \in w \land Depends(v, x, E) \triangleright x \} \cup \\ &\qquad \{ v, x \mid v_0 \in w \land Depends(v, x, E) \triangleright x_0 \} \end{aligned}$$

It will be convenient to allow $Resolve(w_0 = w, E)$ as a shorthand for

$$Resolve(w_0, E) = Resolve(w, E) \qquad ,$$

i.e., the formula that states that the value of each variable in $Resolve(w, E)$ is unchanged.

Properties of *Resolve*

From the monotonicity of $Depends$ (11.0), we have that $Resolve$ is monotonic in both arguments, *i.e.*, for any w, w', E, E',

$$E \subseteq E' \land w \subseteq w' \;\Rightarrow\; Resolve(w, E) \subseteq Resolve(w', E') \qquad . \tag{11.1}$$

In rewriting an expression describing an element of a resolve set, the following property will come in handy. For any c, w, E,

$$(c \in Resolve(w, E)) \;=\; \langle \exists a \mid a \in w \triangleright Depends(a, c, E) \rangle \quad . \tag{11.2}$$

We also have, for any w visible in some environment E,

$$w \subseteq \mathit{Resolve}(w, E) \subseteq E \qquad . \tag{11.3}$$

The fact that *Resolve* is a closure is reflected in the following property. For any a, c, w, E,

$$\mathit{Depends}(a, c, E) \wedge a \in \mathit{Resolve}(w, E) \Rightarrow c \in \mathit{Resolve}(w, E) \qquad . \tag{11.4}$$

11.1.2 *Functionalize*

I now formally explain the interpretation of specification variables. A specification variable is a function on the program variables. For every specification variable a, I introduce a function $f{\star}a$. (Recall that $\star$ is a special character, so that $f{\star}a$ is just an identifier. This identifier is distinct from all programmer-declared identifiers and is uniquely determined from the name a.)

Example

Let me start with an example. Consider the following declarations.

$$\textbf{spec var}\ a \quad ; \quad \textbf{var}\ c \quad ; \quad \textbf{depends}\ a\ \textbf{on}\ c \quad ; \quad \textbf{rep}\ a\ \textbf{is}\ a = c^2$$

Let's ponder the interpretation of expression $a = 9$. The a in this expression is *functionalized* into

$$f{\star}a(a, c)$$

and a's **rep** declaration is functionalized into

$$\langle \forall a, c \triangleright f{\star}a(a, c) = c^2 \rangle \qquad .$$

Thus, the expression $a = 9$ is shorthand for $f{\star}a(a, c) = 9$, which, using the functionalized representation of a, simplifies to $c^2 = 9$. Hence, a guarded command $c := 3$ will establish $a = 9$, and so will $c := -3$.

Notice the occurrence of the symbol a as a parameter in the instantiation of $f{\star}a$. This a is called the *residue* of the specification variable a. The inclusion of residues is important for soundness; they must not be left out, as I show in Section 11.3.

Definition

Functionalize distributes over all connectives. I show its effect on atomic formulas, quantifiers, and substitution functions.

Functionalize has no effect on constants or program variables. Hence, for any constant c and (possibly initial-value) program variable v, we have the following rules.

$$\begin{array}{l} \mathit{Functionalize}(c, E) = \\ \quad c \end{array}$$

$$Functionalize(v, E) = \\ \quad v$$

For specification variables a, $Functionalize$ is more interesting.

$$Functionalize(a, E) = \\ \quad f{\star}a(Resolve(a, E))$$

$$Functionalize(a_0, E) = \\ \quad f{\star}a(Resolve(a_0, E))$$

The order of the arguments in $f{\star}a$'s argument list is a function of E. That is, if specification variable a is functionalized more than once with respect to the same environment, the order of the arguments of function $f{\star}a$ remains the same. For a different environment, however, the order may be different, and the number of arguments may also differ since different environments provide different visibility of dependencies. In the next chapter, I consider the functionalization with respect to two environments, one of which is a subset of the other. I then assume that the common dependencies are listed first, and are listed in the same order for both environments.

Now for quantifiers.

$$Functionalize(\langle\, \mathbf{Q}\, w \mid R \triangleright T \,\rangle, E) = \\ \quad \langle\, \mathbf{Q}\, Resolve(w, E) \mid Functionalize(R, E) \triangleright Functionalize(T, E) \,\rangle$$

Finally, for the substitution function,

$$Functionalize(Q[x := y], E) = \\ \quad Functionalize(Q, E)[Resolve(x, E) := Resolve(y, E)] \qquad ,$$

where the first of the two occurrences of $Resolve$ takes a list of variables x and produces x in which each variable is replaced by its list of dependencies, and similarly for $Resolve$ of the list of expressions y. Thus, for example, given

$$\textbf{spec var}\ a, b \quad ; \quad \textbf{var}\ c, d, e \quad ; \quad \textbf{depends}\ a\ \textbf{on}\ c, d \quad ; \quad \textbf{depends}\ b\ \textbf{on}\ d, e$$

in E, $Functionalize(Q[a_0, b_0 := a, b], E)$ simplifies to

$$Functionalize(Q)[a_0, c_0, d_0, b_0, d_0, e_0 := a, c, d, b, d, e] \qquad .$$

Note that this expression contains two occurrences of $d_0 := d$. Instead of complicating the definition to avoid this, I simply take its meaning to be

$$Functionalize(Q)[a_0, c_0, b_0, d_0, e_0 := a, c, b, d, e] \qquad ,$$

that is, duplicate substitutions are ignored. Note that, provided the original substitution contains no duplicate symbols on the left-hand side, the functionalized substitution will contain no conflicting substitutions (by a conflicting substitution, I mean, *e.g.*, $x, x := y, z$, where y and z differ).

Functionalize and substitutions

Recall from the meaning of units and imports, that if a unit A imports another unit B, then A may reference identifiers declared in B. Conversely, if A does not import B, then A cannot reference any identifiers declared in B. I use this fact implicitly.

In a similar way, I implicitly use the fact that the representation of a specification variable a refers only to those variables on which a depends. It is this fact that justifies the correctness of the definition of *Functionalize*: Since a specification variable a is replaced by an expression that explicitly shows all visible variables on which a depends, substitution in functionalized expressions works properly.

To illustrate my point, let me offer an example. Consider the expression $a = 9$ appearing in the example earlier in this section. The functionalization of that expression is $f{\star}a(a,c) = 9$. Applying the substitution $c := 3$ (*i.e.*, calculating the weakest precondition of $c := 3$ with respect to postcondition $f{\star}a(a,c) = 9$), we then get $f{\star}a(a,3) = 9$. Subsequently applying the representation of a, we get $3^2 = 9$, which simplifies to *true*.

Now consider what happens if, hypothetically, the representation of a also mentioned a variable d, *e.g.*,

$$\textbf{rep}\ a\ \textbf{is}\ a = c^2 + d \qquad ,$$

where a is *not* declared to depend on d. Then, the functionalization of $a = 9$ would still be $f{\star}a(a,c) = 9$. Thus, a substitution $d := d+1$ has no effect on this expression, since d does not occur in it. However, the update of d *does* have an effect on the value of a, so, here, substitution in the functionalized expression does not work properly.

In short, the restriction that a **rep** clause for a variable a can only refer to those variables on which a depends, justifies why performing substitutions in functionalized expressions is correct.

Leibniz

Note that for any environment E that contains a representation of a specification variable a,

$$[\mathit{Functionalize}(a_0 = a, E) \Leftarrow \mathit{Resolve}(a_0 = a, E)] \qquad . \tag{11.5}$$

That is, if all variables on which a depends are unchanged, then the value of a functionalized is unchanged, too. Applied to the example above, this line reads

$$f{\star}a(a_0, c_0) = f{\star}a(a,c) \Leftarrow a_0 = a \wedge c_0 = c \qquad .$$

This phenomenon is known as Leibniz's Rule.

11.1.3 *RepPreds*

RepPreds of an environment E is the conjunction of "axioms" formed from the **rep** declarations in E. For each

$$\textbf{rep}\ a\ \textbf{is}\ P$$

occurring in E, the corresponding axiom is

$$\langle \forall\, w \triangleright \mathit{Functionalize}(P, E) \rangle \qquad ,$$

where $w = \mathit{Resolve}(a, E)$.

We can rewrite this expression as follows.

$$\begin{array}{ll} & \langle \forall\, \mathit{Resolve}(a, E) \triangleright \mathit{Functionalize}(P, E) \rangle \\ = & \qquad \{ \ \mathit{Functionalize} \text{ over quantification } \} \\ & \mathit{Functionalize}(\langle \forall\, a \triangleright P \rangle, E) \end{array}$$

Let the first argument to *Functionalize* in the last line be denoted by $\mathit{RepAxiom}(a, E)$, and let $\mathit{RepAxioms}(E)$ be the conjunction of $\mathit{RepAxiom}(a, E)$ for each a for which a **rep** is declared in E. Since *Functionalize* distributes over conjunction, we thus have

$$\begin{array}{l} \mathit{RepPreds}(E) = \\ \qquad \mathit{Functionalize}(\mathit{RepAxioms}(E), E) \qquad . \end{array}$$

11.1.4 *GetSpec*

Given a procedure id and an environment E, $\mathit{GetSpec}(id, E)$ denotes the specification of id, desugared according to E.

$$\begin{array}{l} \mathit{GetSpec}(id, E) = \\ \qquad w : [\mathit{Pre},\ \mathit{Post}] \\ \qquad\qquad \text{where} \\ \qquad (\textbf{modifies}\ \mathit{frame}\ \textbf{requires}\ P\ \textbf{ensures}\ Q) = \mathit{Lookup}(id, E)\ \wedge \\ \qquad w = \mathit{Resolve}(\mathit{frame}, E)\ \wedge \\ \qquad \mathit{Pre} = \mathit{Functionalize}(P, E)\ \wedge \\ \qquad \mathit{Post} = \mathit{Functionalize}(Q \wedge \mathit{BenSideEffects}(w, E), E) \end{array}$$

For a procedure id, $\mathit{Lookup}(id, E)$ returns the specification of id as declared in E.

BenSideEffects is the conjunct added to the postcondition when desugaring a specification. This ensures that the procedure's effect on specification variables not listed in the resolved frame can be described as *benevolent side effects* (see Section 10.2.2), meaning their values do not change.

Before giving the exact definition of *BenSideEffects*, let me illustrate the idea with an example. Consider the declarations

$$\textbf{spec var}\ a, b \quad ; \quad \textbf{var}\ c \quad ; \quad \textbf{depends}\ a\ \textbf{on}\ c \quad ; \quad \textbf{depends}\ b\ \textbf{on}\ c$$

and the procedure specification

$$\textbf{spec}\ p\ \textbf{is modifies}\ a\ \textbf{requires}\ \mathit{true}\ \textbf{ensures}\ a = 9 \qquad .$$

GetSpec of p in this environment desugars the specification of p into

$$a, c : [\mathit{true},\ \mathit{Functionalize}(a = 9 \wedge b_0 = b, E)] \qquad ,$$

which simplifies to

$$a, c : [\mathit{true},\ \ f{\star}a(a, c) = 9\ \ \wedge\ \ f{\star}b(b_0, c_0) = f{\star}b(b, c)] \qquad .$$

Here, a has been replaced by its resolve set a, c, and the pre- and postconditions have been functionalized. Moreover, the postcondition has an extra conjunct which, in effect, states that the procedure may only modify c in such a way as to preserve the value of b, that is, preserve the value of $f{\star}b(b, c)$.

Now for the definition of *BenSideEffects*.

$$\begin{aligned}
&\mathit{BenSideEffects}(w, E) = \\
&\qquad b_0 = b \\
&\qquad\qquad \text{where} \\
&\qquad b = \mathit{SpecVarProjection}(\mathit{BenSideSet}(w, E))
\end{aligned}$$

In words, *BenSideEffects* states that b is unchanged, where b denotes the list of variables prescribed by *BenSideSet*. In the following, v ranges over variables.

$$\begin{aligned}
&\mathit{BenSideSet}(w, E) = \\
&\qquad \{\, v \mid v \in E \setminus w \triangleright\ v \,\}
\end{aligned}$$

Stated differently, if V is the set of variables in E, then $\mathit{BenSideSet}(w, E)$ is simply $V \setminus w$.

> **Remark 11.4.** The set *BenSideSet* can be quite large. Its size can be reduced by using (11.9). Then, $\mathit{BenSideSet}(w, E)$ can be simply those variables whose downward closure overlaps with w. For an already resolved list w, the definition is then written
>
> $$\begin{aligned}
&\mathit{BenSideSet}(w, E) = \\
&\qquad \{\, v \mid v \in E \setminus w\ \ \wedge\ \ w \cap rv \neq \{\,\} \triangleright\ v \,\} \\
&\qquad\qquad \text{where} \\
&\qquad rv = \mathit{Resolve}(v, E) \qquad .
\end{aligned}$$

Finally, *SpecVarProjection* of a set s is the set of specification variables in s.

$$\begin{aligned}
&\mathit{SpecVarProjection}(s) = \\
&\qquad \{\, a \mid a \in s \wedge a \text{ is a specification variable} \triangleright\ a \,\}
\end{aligned}$$

Note that *SpecVarProjection*, being a projection, is monotonic in its argument, *i.e.*, for any sets of variables s and s',

$$s \subseteq s' \Rightarrow \mathit{SpecVarProjection}(s) \subseteq \mathit{SpecVarProjection}(s') \qquad .$$

11.1.5 Weakest liberal precondition

Commands are modeled by their weakest liberal preconditions. The variables mentioned in program statements are all program variables, so their interpretation does not depend on the environment. However, procedure calls are defined in terms of their specifications, which may refer to specification variables. Therefore, the interpretation of procedure calls is affected by the environment.

For any command gc and predicate Q written in an environment E, $wlp(gc, Q, E)$ gives the weakest liberal precondition of the command interpreted in E.

$$
\begin{aligned}
&wlp(gc, Q, E) = \\
&\qquad wlp.Env(gc, E).Q
\end{aligned}
$$

where the second occurrence of wlp denotes the weakest liberal precondition from Chapter 1 but dropping the exceptional postcondition (*i.e.*, implicitly letting it be $false$; *cf.* Section 6.2).

$Env(gc, E)$ is the interpretation of a command gc in an environment E. $Env(gc, E)$ is defined as gc in which all occurrences of procedure calls $\mathbf{call}\, p$ are replaced by the specification statement $w : [P,\ Q]$, where $w : [P,\ Q] = GetSpec(p, E)$.

From the definition of wlp for each statement, we can show that wlp is positively finitely conjunctive (in its second argument), *i.e.*, for any predicates P and Q and environment E,

$$wlp(gc, P \land Q, E) \;=\; wlp(gc, P, E) \land wlp(gc, Q, E) \qquad . \tag{11.6}$$

(wlp is actually universally conjunctive in general (see also Remark 6.0), but we only need conjunctivity for nonempty finite bags of predicates.) Note that conjunctivity implies monotonicity [21].

11.1.6 *Target*

The set of targets of a command is the set of all variables that, according to a naïve syntactic inspection of the command, are potentially updated by an execution of the command.

In $Target(gc, E)$, gc is a guarded command and E is an environment. It equals the resolve set of $RawTarget(gc, E)$.

$$
\begin{aligned}
&Target(gc, E) = \\
&\qquad Resolve(RawTarget(gc, E), E)
\end{aligned}
$$

On account of (11.3), we thus have

$$RawTarget(gc, E) \subseteq Target(gc, E) \qquad . \tag{11.7}$$

$RawTarget$ is defined inductively over the syntax of the command. Only two commands update variables: the assignment statement and the procedure call.

RawTarget is the unfunctionalized set of variables that are possibly updated by a command.

$$\begin{array}{l} \mathit{RawTarget}(v := e, E) = \\ \quad \{v\} \end{array}$$

$$\begin{array}{l} \mathit{RawTarget}(\mathbf{call}\ p, E) = \\ \quad \mathit{fr} \\ \qquad \text{where} \\ \quad (\mathbf{modifies}\ \mathit{fr}\ \mathbf{requires}\ P\ \mathbf{ensures}\ Q) = \mathit{Lookup}(p, E) \end{array}$$

I also show the definition of *RawTarget* for a block statement.

$$\begin{array}{l} \mathit{RawTarget}([\![\ x \bullet s\ ]\!], E) = \\ \quad \mathit{RawTarget}(s, E) \setminus \{x\} \end{array}$$

For all other primitive statements, *RawTarget* equals the empty set. *RawTarget* of other statement compositions equals the union of *RawTarget* for the constituent statements.

Note that since *RawTarget* does not perform functionalization (the environment is used only to retrieve the raw specification of procedures), $\mathit{RawTarget}(gc, E)$ is the same for all environments E in which gc can be written.

11.1.7 *Refine*

Finally, I get to the definition of *Refine*.

$$\begin{array}{l} \mathit{Refine}(\mathbf{impl}\ \mathit{id}\ \mathbf{is}\ gc, E) = \\ \quad \mathit{Rep} \Rightarrow (w : [\mathit{Pre},\ \mathit{Post}] \sqsubseteq \mathit{impl}) \\ \qquad \text{where} \\ \quad w : [\mathit{Pre},\ \mathit{Post}] = \mathit{GetSpec}(\mathit{id}, E) \land \\ \quad \mathit{impl} = \mathit{Env}(gc, E) \land \\ \quad \mathit{Rep} = \mathit{RepPreds}(E) \end{array}$$

Remember from (1.18) that $s \sqsubseteq t$ (here for *wlp*) is defined as

$$\langle \forall R \triangleright [\mathit{wlp}.s.R \Rightarrow \mathit{wlp}.t.R] \rangle \qquad ,$$

where R ranges over all predicates. This formula may look a bit intimidating, since it involves a quantification over all predicates. Using a result from the next section, I now transform this expression into an equivalent one. This alternative formulation may be preferable to work with in certain circumstances, *e.g.*, in automated theorem proving. It is also the formulation I use in the proof in Chapter 12.

First, note that *Rep* has no free variables, *i.e.*, it is a *boolean scalar*. We calculate,

$$\begin{array}{ll} & \mathit{Rep} \Rightarrow (w : [\mathit{Pre},\ \mathit{Post}] \sqsubseteq \mathit{impl}) \\ = & \quad \{\ (1.18)\text{: def. of } \sqsubseteq\ \} \end{array}$$

$$
\begin{array}{ll}
& Rep \ \Rightarrow \ \langle \forall R \triangleright \ [wlp.(w:[Pre,\ Post]).R \ \Rightarrow \ wlp.impl.R] \rangle \\
= & \quad \{ \ (1.16)\text{: } wlp \text{ of specification statement, where } v_0 \text{ denotes the} \\
& \quad\ \ \text{initial-value variables in } Post \ \} \\
& Rep \ \Rightarrow \ \langle \forall R \triangleright \ [Pre \ \wedge \ \langle \forall w \mid Post \triangleright \ R \rangle [v_0 := v] \ \Rightarrow \ wlp.impl.R] \rangle \\
= & \quad \{ \ \ \Rightarrow \ \text{ over } \forall\text{, since } Rep \text{ is a boolean scalar } \} \\
& \langle \forall R \triangleright \ Rep \ \Rightarrow \ [Pre \ \wedge \ \langle \forall w \mid Post \triangleright \ R \rangle [v_0 := v] \ \Rightarrow \ wlp.impl.R] \rangle \\
= & \quad \{ \ \ \Rightarrow \ \text{ over } [\,]\text{, since } Rep \text{ is a boolean scalar } \} \\
& \langle \forall R \triangleright \ [Rep \ \Rightarrow \ (Pre \ \wedge \ \langle \forall w \mid Post \triangleright \ R \rangle [v_0 := v] \ \Rightarrow \ wlp.impl.R)] \rangle \\
= & \quad \{ \ \text{pred. calc. } \} \\
& \langle \forall R \triangleright \ [Rep \ \wedge \ Pre \ \wedge \ \langle \forall w \mid Post \triangleright \ R \rangle [v_0 := v] \ \Rightarrow \ wlp.impl.R] \rangle \\
= & \quad \{ \ (1.16)\text{: } wlp \text{ of specification statement } \} \\
& \langle \forall R \triangleright \ [wlp.(w:[Rep \ \wedge \ Pre,\ Post]).R \ \Rightarrow \ wlp.impl.R] \rangle \\
= & \quad \{ \ (1.18)\text{: def. of } \sqsubseteq \ \} \\
& w:[Rep \ \wedge \ Pre,\ Post] \sqsubseteq impl \qquad .
\end{array}
$$

Define t and m by

$$t = Target(impl, E) \ \wedge \ m = t \setminus w \qquad ,$$

and let z denote the list of all variables. m is the list of variables that, from a syntactic inspection of gc, are possible targets of gc, but which, according to the procedure specification, are not allowed to be modified. We then have,

$$
\begin{array}{ll}
& w:[Rep \ \wedge \ Pre,\ Post] \sqsubseteq impl \\
= & \quad \{ \ (11.9)\text{, since } w \text{ and } m \text{ are disjoint } \} \\
& w, m:[Rep \ \wedge \ Pre,\ Post \ \wedge \ m_0 = m] \sqsubseteq impl \\
= & \quad \{ \ \ w \text{ and } m \text{ partition } t \ \} \\
& t:[Rep \ \wedge \ Pre,\ Post \ \wedge \ m_0 = m] \sqsubseteq impl \\
= & \quad \{ \ \ t \text{ is the set of targets of } impl\text{, } i.e.\text{, } t \text{ is a conservative} \\
& \quad\ \ \text{estimate of the variables that are possibly modified by } impl \ \} \\
& z:[Rep \ \wedge \ Pre,\ Post \ \wedge \ m_0 = m] \sqsubseteq impl \qquad .
\end{array}
$$

Let v_0 denote a list of initial-value variables, such that v_0 is a superset of the initial-value variables in $Post \ \wedge \ m_0 = m$. Then, from the above and (11.10), we have

$$Refine(\mathbf{impl}\ id\ \mathbf{is}\ gc, E) \quad = \quad [Rep \wedge Pre \wedge v_0 = v \Rightarrow wlp.impl.(Post \wedge m_0 = m)] \ . \tag{11.8}$$

In the above calculation, I used the fact that, for any disjoint lists of variables w and m, and predicates P and Q,

$$w:[P,\ Q] \quad = \quad w, m:[P,\ Q \wedge m_0 = m] \qquad . \tag{11.9}$$

I end this section by discharging that proof obligation.

Recall that the weakest (liberal) precondition of a specification $w:[P,\ Q]$ with respect to a postcondition R is

$$P \wedge \langle \forall w \mid Q \triangleright \ R \rangle [v_0 := v] \qquad ,$$

where v_0 is the list of initial-value variables that occur in Q. Since neither w nor R contain initial-value variables —initial-value variables can only be written in the postcondition of specifications—, this expression equals

$$P \wedge \langle \forall w \mid Q \triangleright R \rangle [z_0 := z] \qquad ,$$

where z is any superset of v (*cf.* Section 1.7).

Let z denote any superset of $v \cup m$ (which, of course, is also a superset of v). Then, we calculate,

$$\begin{array}{ll}
 & wlp.(w, m : [P,\ Q \wedge m_0 = m]).R \\
= & \qquad \{ \ (1.16)\text{: } wlp \text{ of specification, using } z \text{ instead of } v \ \} \\
 & P \wedge \langle \forall w, m \mid Q \wedge m_0 = m \triangleright R \rangle [z_0 := z] \\
= & \qquad \{ \ \text{one-point rule, since } m \text{ and } m_0 \text{ are disjoint} \ \} \\
 & P \wedge \langle \forall w \mid Q[m := m_0] \triangleright R[m := m_0] \rangle [z_0 := z] \\
= & \qquad \{ \ \text{substitution distributes over } \forall \text{, since } m \text{ and } w \text{ are disjoint} \ \} \\
 & P \wedge \langle \forall w \mid Q \triangleright R \rangle [m := m_0][z_0 := z] \\
= & \qquad \{ \ \text{substitution, since } m \subseteq z \text{ and thus } m_0 \subseteq z_0 \text{, and since} \\
 & \qquad \quad m \text{ and } z_0 \text{ are disjoint} \ \} \\
 & P \wedge \langle \forall w \mid Q \triangleright R \rangle [m := m][z_0 := z] \\
= & \qquad \{ \ [m := m] \text{ is the identity function} \ \} \\
 & P \wedge \langle \forall w \mid Q \triangleright R \rangle [z_0 := z] \\
= & \qquad \{ \ (1.16)\text{: } wlp \text{ of specification, using } z \text{ instead of } v \ \} \\
 & wlp.(w : [P,\ Q]).R \qquad .
\end{array}$$

11.2 Proving refinements

In this section, I give a proof of

$$(z : [Pre,\ Post] \sqsubseteq S) \ = \ [Pre \wedge v_0 = v \Rightarrow wlp.S.Post] \qquad , \tag{11.10}$$

where z denotes the list of all variables and v_0 is a list of initial-value variables such that v_0 is a superset of the set of initial-value variables in $Post$. This property allows a refinement of this form to be proven using a simple implication.

I define a z-*predicate* to be a predicate over a state space whose variables are z. For the S in (11.10), $wlp.S$ is thus a z-predicate transformer, *i.e.*, a function from one z-predicate to another. Hence, for any z-predicate R, $wlp.S.R$ is a z-predicate.

We calculate,

$$\begin{array}{ll}
 & z : [Pre,\ Post] \sqsubseteq S \\
= & \qquad \{ \ (1.18)\text{: def. of } \sqsubseteq \text{, and } (1.16)\text{: } wlp \text{ of specification statement} \ \} \\
 & \langle \forall R \triangleright [Pre \wedge \langle \forall z \mid Post \triangleright R \rangle [v_0 := v] \Rightarrow wlp.S.R] \rangle \\
= & \qquad \{ \ \text{substitution, since } v_0 \text{ does not occur free in } Pre \text{ or } wlp.S.R \ \} \\
 & \langle \forall R \triangleright [(Pre \wedge \langle \forall z \mid Post \triangleright R \rangle \Rightarrow wlp.S.R)[v_0 := v]] \rangle \\
= & \qquad \{ \ \text{below} \ \}
\end{array}$$

$$
\begin{array}{ll}
& [(Pre \;\Rightarrow\; wlp.S.Post)[v_0 := v]] \\
= & \quad \{ \;\text{substitution, since } v_0 \text{ does not occur free in } Pre \;\} \\
& [Pre \;\Rightarrow\; wlp.S.Post[v_0 := v]] \\
= & \quad \{ \;\text{one-point rule, since } v_0 \text{ does not appear in} \\
& \quad\quad Pre \;\Rightarrow\; wlp.S.Post[v_0 := v] \;\} \\
& [Pre \wedge v_0 = v \;\Rightarrow\; wlp.S.Post[v_0 := v]] \\
= & \quad \{ \;\text{equality and substitution} \;\} \\
& [Pre \wedge v_0 = v \;\Rightarrow\; wlp.S.Post] \qquad .
\end{array}
$$

The deferred step is proved by mutual implication. For one direction,

$$
\begin{array}{ll}
& \langle \forall R \triangleright [(Pre \wedge \langle \forall z \mid Post \triangleright R \rangle \;\Rightarrow\; wlp.S.R)[v_0 := v]] \rangle \\
\Leftarrow & \quad \{ \;\text{monotonicity, since } \forall z \text{ quantifies over all free variables of } R, \\
& \quad\quad \text{and } wlp.S \text{ is a } z\text{-predicate transformer} \;\} \\
& \langle \forall R \triangleright [(Pre \wedge \langle \forall z \mid Post \triangleright R \rangle \;\Rightarrow\; wlp.S.Post)[v_0 := v]] \rangle \\
\Leftarrow & \quad \{ \;\text{monotonicity} \;\} \\
& \langle \forall R \triangleright [(Pre \;\Rightarrow\; wlp.S.Post)[v_0 := v]] \rangle \\
= & \quad \{ \;\text{range in nonempty} \;\} \\
& [(Pre \;\Rightarrow\; wlp.S.Post)[v_0 := v]] \qquad ,
\end{array}
$$

and the other,

$$
\begin{array}{ll}
& \langle \forall R \triangleright [(Pre \wedge \langle \forall z \mid Post \triangleright R \rangle \;\Rightarrow\; wlp.S.R)[v_0 := v]] \rangle \\
= & \quad \{ \;[\,] \text{ over } \forall \;\} \\
& [\langle \forall R \triangleright (Pre \wedge \langle \forall z \mid Post \triangleright R \rangle \;\Rightarrow\; wlp.S.R)[v_0 := v] \rangle] \\
\Rightarrow & \quad \{ \;\text{instantiate with } R := Post \;\} \\
& [(Pre \wedge \langle \forall z \mid Post \triangleright Post \rangle \;\Rightarrow\; wlp.S.Post)[v_0 := v]] \\
= & \quad \{ \;\text{pred. calc.} \;\} \\
& [(Pre \;\Rightarrow\; wlp.S.Post)[v_0 := v]] \qquad .
\end{array}
$$

Remark 11.5. A proof of (11.10) appears in [67]. However, that proof contains a step whose hint is rather vague. In my notation, using symbols similar to those above, and with *wlp* instead of *wp* (but remember that *wp* and *wlp* coincide for the specification statement—see Sections 1.7 and 6.2), that step is

$$
\begin{array}{ll}
& Pre \;\Rightarrow\; wlp.S.Post \\
\Rightarrow & \quad \{ \;\text{by distributivity of } \Rightarrow \text{ over weakest preconditions} \;\} \\
& Pre \wedge \langle \forall z \triangleright Post \;\Rightarrow\; R \rangle \;\Rightarrow\; wlp.S.R \qquad ,
\end{array}
$$

where R is an arbitrary predicate. The step is correct, but the hint seems to suggest that a step like

$$
\begin{array}{ll}
& Pre \;\Rightarrow\; wlp.S.Post \\
\Rightarrow & \quad \{ \;\text{by distributivity of } \Rightarrow \text{ over weakest preconditions} \;\} \\
& Pre \wedge (Post \;\Rightarrow\; R) \;\Rightarrow\; wlp.S.R \qquad ,
\end{array}
$$

would also be correct, which it is not! $wlp.S$ is a monotonic function from z-predicates to z-predicates. This monotonicity is used in the proof step. However, in order to apply monotonicity, the two predicates involved ($Post$ and R above) must be ordered by implication at *every* value of z. This subtle point is not visible from the hint in [67].

In my first proof of the above theorem, which I constructed with Paul Sivilotti not knowing of the proof in [67], we fell into the trap of writing a vague hint like the one in [67]. Only later did I realize the subtlety described here.

As another remark on this step, $\langle \forall z \mid Post \triangleright R \rangle$ can be written with everywhere brackets as $[Post \Rightarrow R]$, where the everywhere brackets quantify over all values of z. Then the hint in my proof can be stated as

$$[Post \Rightarrow R] \Rightarrow [wlp.S.Post \Rightarrow wlp.S.R] \qquad ,$$

which clearly expresses the property being used. However, there is another set of everywhere brackets in the proof, and those everywhere brackets quantify over all values of z and z_0. To avoid confusion between the two different pairs of everywhere brackets, I kept the explicit quantification over z.

11.3 The importance of residues

I conclude this chapter by giving an example that shows the importance of the presence of residues. Consider the following unit.

unit A **is**
  spec var a ; **var** c ; **depends** a **on** c ;
  spec pc **is modifies** c **requires** $true$ **ensures** $true$;
  spec pa **is modifies** a **requires** $true$ **ensures** $c_0 = c$;
end

Procedure pc is specified to play havoc with c, but with no other variable and, in particular, without affecting the value of a. Procedure pa plays havoc with a (*i.e.*, it plays havoc with the resolve set of a), but ensures that the value of c is unchanged.

Contemplate the following (incorrect) argument.

> pc is specified to only modify c, whereas pa can modify a. In the scope of A, the downward closure of a contains only one concrete variable, *viz.*, c. Thus, both pc and pa are constrained to modifying only c. Furthermore, pc can change c to any value, whereas pa must leave its value unchanged. Hence, pa refines pc.

As I show in this section, pa does not refine pc, so the above argument is incorrect. It is incorrect because the downward closure of a may contain concrete variables other than c—concrete variables that are not visible in A. Residues are what catch the erroneous reasoning demonstrated in the above argument. Let's take a look at how this works.

Consider another unit.

$$
\begin{array}{l}
\textbf{unit } B \textbf{ import } A \textbf{ is} \\
\quad \textbf{var } d \;\; ; \quad \textbf{depends } a \textbf{ on } d \; ; \\
\quad \textbf{impl } pa \textbf{ is } d := d + 1 \; ; \\
\quad \vdots \\
\qquad \ldots \; d := 0 \; ; \; \textbf{call } pc \; ; \; \textbf{assert } d = 0 \; \ldots \\
\textbf{end}
\end{array}
\tag{11.11}
$$

This unit introduces another of a's dependencies, d. The unit defines the implementation of pa, which I will show to be a correct refinement of its specification. In the unit, I show another little code segment, (11.11) I will demonstrate that this code segment does not go wrong.

First, the refinement of pa.

$$
\begin{array}{ll}
 & Refine(\textbf{impl } pa \textbf{ is } d := d + 1, \; B) \\
= & \qquad \{ \;\; (11.8){:}\;\; Refine \;\; \} \\
 & [c_0 = c \;\Rightarrow\; wlp.(d := d + 1).(c_0 = c)] \\
= & \qquad \{ \;\; (1.0){:}\;\; wlp \text{ of } := \;\; \} \\
 & [c_0 = c \;\Rightarrow\; c_0 = c] \\
= & \qquad \{ \;\; \text{pred. calc.} \;\; \} \\
 & true
\end{array}
$$

This shows that pa is indeed implemented correctly. Note that, as suggested immediately following the incorrect argument above, this shows that pa really does have an effect on a concrete variable other than c. (Thus far, we have not used the presence of residues.)

Let's prove that the code segment (11.11) in unit B does not go wrong.

$$
\begin{array}{ll}
 & wlp(d := 0 \; ; \; \textbf{call } pc \; ; \; \textbf{assert } d = 0, \; true, \; B) \\
= & \qquad \{ \;\; (1.3{,}6.10){:}\;\; wlp \text{ of } ; \text{ and } \textbf{assert} \;\; \} \\
 & wlp.(d := 0).(wlp.Env(\textbf{call } pc, B).(d = 0)) \\
= & \qquad \{ \;\; Env\text{, using Remark 11.4} \;\; \} \\
 & wlp.(d := 0).(wlp.(c : [true, \; true]).(d = 0)) \\
= & \qquad \{ \;\; (1.14){:}\;\; wlp \text{ of specification statement} \;\; \} \\
 & wlp.(d := 0).\langle \forall c \triangleright d = 0 \rangle \\
= & \qquad \{ \;\; \text{pred. calc.} \;\; \} \\
 & wlp.(d := 0).(d = 0) \\
= & \qquad \{ \;\; (1.0){:}\;\; wlp \text{ of } :=\text{, and pred. calc.} \;\; \} \\
 & true
\end{array}
$$

(This calculation, too, is independent of the presence of residues.) We conclude that pa is correctly implemented and that (11.11) does not go wrong regardless of whether or not residues are included.

Now, let's look at one more unit. This unit provides the implementation of pc.

> **unit** C **import** A **is**
>   **impl** pc **is call** pa
> **end**

Does this implementation of pc meet pc's specification? It would if pa were a refinement of pc. Let's find out.

$$\begin{array}{ll} & Refine(\textbf{impl}\ pc\ \textbf{is call}\ pa,\ C) \\ = & \quad \{\ (11.8)\text{: } Refine\ \} \\ & [a_0 = a\ \wedge\ c_0 = c\ \Rightarrow\ wlp.Env(\textbf{call}\ pa, C).(f{\star}a(a_0, c_0) = f{\star}a(a, c))] \\ = & \quad \{\ Env\ \} \\ & [a_0 = a\ \wedge\ c_0 = c\ \Rightarrow\ wlp.(a, c : [true,\ c_0 = c]).(f{\star}a(a_0, c_0) = f{\star}a(a, c))] \\ = & \quad \{\ (11.9)\text{, with } w, m := a, c\ \} \\ & [a_0 = a\ \wedge\ c_0 = c\ \Rightarrow\ wlp.(a : [true,\ true]).(f{\star}a(a_0, c_0) = f{\star}a(a, c))] \\ = & \quad \{\ (1.14)\text{: } wlp \text{ of specification statement}\ \} \\ & [a_0 = a\ \wedge\ c_0 = c\ \Rightarrow\ \langle \forall a \triangleright f{\star}a(a_0, c_0) = f{\star}a(a, c)\ \rangle] \end{array} \tag{11.12}$$

Since nothing is known about function $f{\star}a$, we say it is *uninterpreted*; thus, we understand $f{\star}a$ as being universally quantified over all functions. By instantiating $f{\star}a$ with $+$, *i.e.*, $f{\star}a(x, y) = x + y$ for all x and y, the right-hand side of $\Rightarrow$ reads

$$\langle \forall a \triangleright a_0 + c_0 = a + c\ \rangle \quad ,$$

which simplifies to $false$. We conclude that, with residues, we are not able to establish that unit C provides a correct implementation of pc. This is good, because by tracing the procedure call in that implementation, we find that pc has the effect of incrementing d, even though its specification constrains it to modify only c. Hence, the code segment (11.11) *will* go wrong, and we are pleased that we were not able to conclude otherwise.

Let's then compare what happens if residues are not used. Not using residues means that the downward closure of a does not contain the residue a. The calculation embarked on at (11.12) then arrives at the expression

$$[c_0 = c \Rightarrow f{\star}a(c_0) = f{\star}a(c)] \quad ,$$

which simplifies to $true$ on account of Leibniz. So, without residues, a verification process would incorrectly verify the implementation of pc to be correct.

In modular verification, one unit can never be certain that it has information about all dependencies. Therefore, the downward closure of a variable includes a residue that represents all dependencies that are not in scope.

Only if the representation of an abstract variable is in scope is there a way to eliminate residues from expressions. For example, the representation

$$\textbf{rep}\ a\ \textbf{is}\ a = c + d$$

is functionalized (see Section 11.1.3) into the axiom

$$\langle\, \forall\, a, c, d \,\triangleright\, f{\star}a(a, c, d) = c + d \,\rangle \qquad .$$

Note that, despite the presence of a's **rep** clause, a may have dependencies that are not in scope—nothing prevents another unit from declaring other (unused) dependencies. Since such other dependencies do not affect the value of abstract variable a, it is sound to use the above axiom.

In conclusion, I showed in this section that without residues, modular verification would not be sound. In the next chapter, I show that with residues, and with the visibility and authenticity requirements, modular verification *is* sound.

Chapter 12

Soundness of modular verification

In this chapter, I give a formal proof of the soundness of modular verification with respect to my **depends** solution. Using the notation and definitions from the previous chapter, I define the visibility and authenticity requirements formally. Then, I state and prove the soundness theorem. I close the chapter with some remarks on the proof.

12.0 Requirements

An environment $Prog$ satisfies the *visibility requirement* exactly when for every environment E that is a subset of $Prog$, and for every two variables a and c,

$$\langle \exists E' \mid E' \subseteq Prog \triangleright \; Depends(a, c, E') \rangle \wedge a \in E \wedge c \in E \;\Rightarrow\; Depends(a, c, E) \qquad .$$

In words, the visibility requirement states that if a depends on c, and both a and c are visible in E, then the dependency of a on c is visible in E.

An environment $Prog$ satisfies the *authenticity requirement* exactly when for every environment E that is a subset of $Prog$, and for every two variables a and c,

$$\langle \exists E' \mid E' \subseteq Prog \triangleright \; Depends(a, c, E') \rangle \wedge c \in E \;\Rightarrow\; a \in E \qquad .$$

That is, a variable a that depends on a variable c is visible anywhere c is.

12.1 Soundness

I now state the soundness theorem.

$$\begin{array}{ll} & Refine(\textbf{impl}\ id\ \textbf{is}\ gc,\ E) \Rightarrow Refine(\textbf{impl}\ id\ \textbf{is}\ gc,\ Prog) \\ \Leftarrow & \\ & (\textbf{impl}\ id\ \textbf{is}\ gc) \in E \wedge E \subseteq Prog \wedge \\ & E \text{ is an environment} \wedge \\ & Prog \text{ is an environment that satisfies the} \\ & \quad \text{visibility and authenticity requirements} \end{array}$$

The theorem states that $\textbf{impl}\ \mathit{id}\ \textbf{is}\ \mathit{gc}$ is deemed a correct implementation (of procedure id with respect to its specification) in the environment E only if it would be deemed a correct implementation in the entire program Prog. That is, the implementation of id is verified to be correct in light of the information contained in E, a subset of Prog, only if it would be verified to be correct in light of *all* information in Prog.

This theorem is what allows a verification process to make use of only the information in the scope of a unit U, *i.e.*, $E := \mathit{ImportClosure}(U)$, when verifying that an implementation in U refines its specification.

Stated differently, to show that a procedure implementation meets its specification, one needs to establish

$$\mathit{Refine}(\textbf{impl}\ \mathit{id}\ \textbf{is}\ \mathit{gc},\ \mathit{Prog}) \quad . \tag{12.0}$$

Since this involves the entire program Prog, and the entire program may not be available at the time the procedure is to be verified, this proof obligation seems difficult to meet. The soundness theorem states that Refine is monotonic with respect to its second argument, the environment. Hence, to show (12.0), it suffices to show

$$\mathit{Refine}(\textbf{impl}\ \mathit{id}\ \textbf{is}\ \mathit{gc},\ E) \tag{12.1}$$

for any subset E of Prog. E can then be picked as the import closure of the unit that contains the procedure implementation, and the proof of (12.1) can be carried out as a modular verification.

In the next few sections, I give the proof of this theorem.

12.2 Proof outline

Let $\mathit{id}, \mathit{gc}, E, \mathit{Prog}$ satisfy the antecedent of the theorem.

$(\textbf{impl}\ \mathit{id}\ \textbf{is}\ \mathit{gc}) \in E \land E \subseteq \mathit{Prog} \land$
E is an environment $\land$
Prog is an environment that satisfies the
visibility and authenticity requirements

We need to prove

$$\mathit{Refine}(\textbf{impl}\ \mathit{id}\ \textbf{is}\ \mathit{gc}, E) \Rightarrow \mathit{Refine}(\textbf{impl}\ \mathit{id}\ \textbf{is}\ \mathit{gc}, \mathit{Prog}) \quad .$$

I introduce symbols $\mathit{frame}, \mathit{pre}, \mathit{post}, b, b', m, m', \mathit{rep}, \mathit{reprep}$, satisfying (*cf.* Section 11.1)

$$(\textbf{modifies}\ \mathit{frame}\ \textbf{requires}\ \mathit{pre}\ \textbf{ensures}\ \mathit{post}) = \mathit{Lookup}(\mathit{id}, E)$$

$$\begin{aligned}
b &= \mathit{SpecVarProjection}(\mathit{BenSideSet}(\mathit{Resolve}(\mathit{frame}, E), E)) \\
b' &= \mathit{SpecVarProjection}(\mathit{BenSideSet}(\mathit{Resolve}(\mathit{frame}, \mathit{Prog}), \mathit{Prog})) \\
m &= \mathit{Target}(\mathit{gc}, E) \setminus \mathit{Resolve}(\mathit{frame}, E) \\
m' &= \mathit{Target}(\mathit{gc}, \mathit{Prog}) \setminus \mathit{Resolve}(\mathit{frame}, \mathit{Prog})
\end{aligned}$$

$$
\begin{aligned}
&rep = RepAxioms(E)\\
&reprep = RepAxioms(Prog \setminus E) \qquad .
\end{aligned}
$$

We thus have

$$
\begin{aligned}
RepPreds(E) &= Functionalize(rep, E)\\
RepPreds(Prog) &= Functionalize(rep, Prog) \wedge Functionalize(reprep, Prog) \qquad .
\end{aligned}
$$

Let rp be a shorthand for $rep \wedge pre$.

For brevity, I will use R synonymously with *Resolve* and define F and F' as follows.

$$
\begin{aligned}
&F(Q) = Functionalize(Q, E)\\
&F'(Q) = Functionalize(Q, Prog)
\end{aligned}
$$

In the sequel, I will implicitly use the fact that F and F' distribute over logical connectives.

Let z denote the list of all variables in $Prog$. In terms of the above shorthands, we now have (*cf.* 11.8),

$$
\begin{aligned}
&Refine(\textbf{impl}\ id\ \textbf{is}\ gc, E) = \\
&\qquad [F(rep) \wedge F(pre) \wedge z_0 = z \Rightarrow\\
&\qquad\quad wlp(gc,\ F(post \wedge b_0 = b) \wedge m_0 = m,\ E)]
\end{aligned} \tag{12.2}
$$

$$
\begin{aligned}
&Refine(\textbf{impl}\ id\ \textbf{is}\ gc, Prog) = \\
&\qquad [F'(rep) \wedge F'(reprep) \wedge F'(pre) \wedge z_0 = z \Rightarrow\\
&\qquad\quad wlp(gc,\ F'(post \wedge b'_0 = b') \wedge m'_0 = m',\ Prog)] \qquad .
\end{aligned} \tag{12.3}
$$

I will make use of the property

$$
b \subseteq b' \qquad , \tag{12.4}
$$

which I will prove later. In light of this, I let bb denote $b' \setminus b$, and partition bb into bx and by. The exact nature of this partition won't be revealed until Section 12.3.1.

We calculate,

$$
\begin{aligned}
& Refine(\textbf{impl}\ id\ \textbf{is}\ gc, Prog)\\
= &\qquad \{\ (12.3)\ \}\\
& [F'(rep)\ \wedge\ F'(reprep)\ \wedge\ F'(pre)\ \wedge\ z_0 = z\ \Rightarrow\\
&\qquad wlp(gc,\ F'(post\ \wedge\ b'_0 = b')\ \wedge\ m'_0 = m',\ Prog)]\\
\Leftarrow &\qquad \{\ \text{strengthen, } rp\text{, and distribute } F'\ \}\\
& [F'(rp)\ \wedge\ z_0 = z\ \Rightarrow\ wlp(gc,\ F'(post\ \wedge\ b'_0 = b')\ \wedge\ m'_0 = m',\ Prog)]\\
= &\qquad \{\ \text{partition } b' \text{ into } b, bx, by \text{ (see (12.4) above)}\ \}\\
& [F'(rp)\ \wedge\ z_0 = z\ \Rightarrow\ wlp(gc,\ F'(post\ \wedge\ b_0 = b)\ \wedge\ F'(bx_0 = bx)\ \wedge\\
&\qquad\qquad\qquad\qquad\qquad\qquad F'(by_0 = by)\ \wedge\ m'_0 = m',\ Prog)]\\
= &\qquad \{\ (11.6)\text{: } wlp \text{ is conjunctive, and pred. calc.}\ \}\\
& [F'(rp)\ \wedge\ z_0 = z\ \Rightarrow\ wlp(gc, F'(post\ \wedge\ b_0 = b)\ \wedge\ F'(bx_0 = bx)\ \wedge\ m'_0 = m', Prog)]\ \wedge\\
& [F'(rp)\ \wedge\ z_0 = z\ \Rightarrow\ wlp(gc, F'(by_0 = by), Prog)] \qquad .
\end{aligned}
$$

The latter of these conjuncts follows from

$$[z_0 = z \Rightarrow wlp(gc,\ F'(by_0 = by),\ Prog)] \qquad , \tag{12.5}$$

which I will prove later.

For the former, I will use a function X. The idea is that X syntactically translates from E-expressions to $Prog$-expressions. Applied to a predicate, X provides a way to re-functionalize the predicate with respect to a larger environment (see (12.6)); applied to a list of variables (*cf.* (12.11)), X provides a way to re-resolve that list. For any predicate Q over the variables in E, and any guarded command S in E, X enjoys the following properties.

$$\begin{array}{ll}
F'(Q) = X(F(Q)) & (12.6)\\
z = X(z) & (12.7)\\
X \text{ distributes over logical connectives} & (12.8)\\
[X(Q)] \Leftarrow [Q] & (12.9)\\
Resolve(bx, Prog) \subseteq X(m) & (12.10)\\
m' \subseteq X(m) & (12.11)\\
wlp(S, X(Q), Prog) \Leftarrow X(wlp(S, Q, E)) & (12.12)
\end{array}$$

I postpone the precise definition of X and the proofs that show that X does have these properties.

Note that X is monotonic: For any predicates $Q0$ and $Q1$ over the variables of E,

$$\begin{array}{ll}
 & [X(Q0) \Rightarrow X(Q1)]\\
= & \quad \{\ (12.8)\ \}\\
 & [X(Q0 \Rightarrow Q1)]\\
\Leftarrow & \quad \{\ (12.9)\ \}\\
 & [Q0 \Rightarrow Q1] \qquad .
\end{array}$$

We calculate,

$$\begin{array}{ll}
 & F'(post \wedge b_0 = b) \wedge F'(bx_0 = bx) \wedge m'_0 = m'\\
\Leftarrow & \quad \{\ (12.11)\ \}\\
 & F'(post \wedge b_0 = b) \wedge F'(bx_0 = bx) \wedge X(m_0 = m)\\
\Leftarrow & \quad \{\ \text{Leibniz (11.5):}\ F'(bx_0 = bx) \Leftarrow R(bx_0 = bx, Prog)\ \}\\
 & F'(post \wedge b_0 = b) \wedge R(bx_0 = bx, Prog) \wedge X(m_0 = m)\\
= & \quad \{\ (12.10)\ \}\\
 & F'(post \wedge b_0 = b) \wedge X(m_0 = m) \qquad .
\end{array}$$

Now, we have,

$$\begin{array}{ll}
 & [F'(rp) \wedge z_0 = z \Rightarrow wlp(gc,\ F'(post \wedge b_0 = b) \wedge F'(bx_0 = bx) \wedge m'_0 = m',\ Prog)]\\
\Leftarrow & \quad \{\ \text{above calculation, and } wlp \text{ is monotonic}\ \}\\
 & [F'(rp) \wedge z_0 = z \Rightarrow wlp(gc,\ F'(post \wedge b_0 = b) \wedge X(m_0 = m),\ Prog)]\\
= & \quad \{\ (12.6)\text{, and } (12.7)\text{, and } (12.8)\ \}
\end{array}$$

$$
\begin{array}{ll}
& [X(F(rp) \ \wedge \ z_0 = z) \ \Rightarrow \ wlp(gc,\ X(F(post \ \wedge \ b_0 = b) \ \wedge \ m_0 = m),\ Prog)] \\
\Leftarrow & \qquad \{ \quad (12.12) \quad \} \\
& [X(F(rp) \ \wedge \ z_0 = z) \ \Rightarrow \ X(wlp(gc,\ F(post \ \wedge \ b_0 = b) \ \wedge \ m_0 = m,\ E))] \\
\Leftarrow & \qquad \{ \quad X \text{ is monotonic} \quad \} \\
& [F(rp) \ \wedge \ z_0 = z \ \Rightarrow \ wlp(gc,\ F(post \ \wedge \ b_0 = b) \ \wedge \ m_0 = m,\ E)] \\
= & \qquad \{ \quad (12.2) \quad \} \\
& Refine(\mathbf{impl}\ id\ \mathbf{is}\ gc, E) \qquad .
\end{array}
$$

That was the soundness proof! Well, only an outline thereof—I still need to produce the proofs of the properties (12.4)–(12.12) that were used in this proof outline. I will do these in order.

12.3 All side effects are benevolent

In this section, I discharge proof obligations (12.4) and (12.5).

12.3.0 $b \subseteq b'$

I now prove (12.4), that is, $b \subseteq b'$.

$$
\begin{array}{ll}
& b \subseteq b' \\
= & \qquad \{ \quad b \text{ and } b' \quad \} \\
& \quad SpecVarProjection(BenSideSet(R(frame, E), E)) \\
& \subseteq \ SpecVarProjection(BenSideSet(R(frame, Prog), Prog)) \\
\Leftarrow & \qquad \{ \quad SpecVarProjection \text{ is monotonic} \quad \} \\
& BenSideSet(R(frame, E), E) \subseteq BenSideSet(R(frame, Prog), Prog) \\
\Leftarrow & \qquad \{ \quad E \subseteq Prog \quad \} \\
& BenSideSet(R(frame, E), E) \text{ is monotonic is } E
\end{array} \tag{12.13}
$$

I prove this monotonicity. Let E and E' be two environments such that $E \subseteq E'$, and let w be any list of variables visible in E. Letting v range over variables, we calculate,

$$
\begin{array}{ll}
& BenSideSet(R(w, E), E) \subseteq BenSideSet(R(w, E'), E') \\
= & \qquad \{ \quad BenSideSet \quad \} \\
& \{ v \mid v \in E \setminus R(w, E) \triangleright v \} \subseteq \{ v \mid v \in E' \setminus R(w, E') \triangleright v \} \\
= & \qquad \{ \quad \text{sets} \quad \} \\
& \langle \forall v \mid v \in E \ \wedge \ v \notin R(w, E) \triangleright \ v \in E' \ \wedge \ v \notin R(w, E') \rangle \\
= & \qquad \{ \quad E \subseteq E' \quad \} \\
& \langle \forall v \mid v \in E \ \wedge \ v \notin R(w, E) \triangleright \ v \notin R(w, E') \rangle \\
= & \qquad \{ \quad \text{trading} \quad \} \\
& \langle \forall v \mid v \in E \ \wedge \ v \in R(w, E') \triangleright \ v \in R(w, E) \rangle \qquad .
\end{array}
$$

For any variable v, we calculate,

$$
\begin{array}{ll}
 & v \in E \ \wedge \ v \in \mathit{Resolve}(w, E') \\
= & \qquad \{ \ \ \text{(11.2): property of } \mathit{Resolve} \ \ \} \\
 & v \in E \ \wedge \ \langle \exists a \mid a \in w \triangleright \mathit{Depends}(a, v, E') \rangle \\
\Rightarrow & \qquad \{ \ \ \text{visibility requirement,} \\
 & \qquad \qquad \text{since } a \text{ is visible in } E \ (a \in w \subseteq E), \\
 & \qquad \qquad \text{and } v \text{ is visible in } E \ (v \in E) \ \ \} \\
 & v \in E \ \wedge \ \langle \exists a \mid a \in w \triangleright \mathit{Depends}(a, v, E) \rangle \\
\Rightarrow & \qquad \{ \ \ \text{(11.2): property of } \mathit{Resolve} \ \ \} \\
 & v \in \mathit{Resolve}(w, E) \qquad ,
\end{array}
$$

which concludes the proof.

12.3.1 *bx* AND *by*

In this section, I define bx and by. As advertised before, bx and by partition bb. From the definition of bx, I calculate the necessary ingredient in the upcoming proof of (12.10). From the definition of by, I prove (12.5).

For any specification variable d, we calculate,

$$
\begin{array}{ll}
 & d \in bb \\
= & \qquad \{ \ \ bb\,,\ \setminus \ \ \} \\
 & d \in b \ \wedge \ b \notin b' \\
= & \qquad \{ \ \ b\,,\ b' \ \ \} \\
 & d \in \mathit{Prog} \setminus R(\mathit{frame}, \mathit{Prog}) \ \wedge \ d \notin E \setminus R(\mathit{frame}, E) \\
\Rightarrow & \qquad \{ \ \ \setminus\text{, twice} \ \ \} \\
 & d \notin R(\mathit{frame}, \mathit{Prog}) \ \wedge \ (d \notin E \ \vee \ d \in R(\mathit{frame}, E)) \\
= & \qquad \{ \ \ \text{(11.1): } \mathit{Resolve} \text{ is monotonic, since } E \subseteq \mathit{Prog} \ \ \} \\
 & d \notin R(\mathit{frame}, \mathit{Prog}) \ \wedge \ d \notin E \qquad .
\end{array}
\tag{12.14}
$$

Treatment of *bx*

I now define bx, a subset of bb: For every d in bb, $d \in bx$ exactly when there is a procedure call $\mathbf{call}\, p$ in gc and a g in the frame of the specification of p such that

$$
\mathit{Depends}(g, d, \mathit{Prog}) \qquad .
$$

Thus,

$$
\begin{array}{ll}
 & \mathit{true} \\
= & \qquad \{ \ \ g \text{ is in frame of specification of } p \ \ \} \\
 & g \in \mathit{RawTarget}(\mathbf{call}\, p, E) \\
\Rightarrow & \qquad \{ \ \ \mathbf{call}\, p \text{ is a substatement of } gc \ \ \} \\
 & g \in \mathit{RawTarget}(gc, E) \\
\Rightarrow & \qquad \{ \ \ \text{(11.7): } \mathit{RawTarget}(gc, E) \subseteq \mathit{Target}(gc, E) \ \ \} \\
 & g \in \mathit{Target}(gc, E) \qquad .
\end{array}
$$

We calculate,

$$
\begin{array}{ll}
& Depends(g, d, Prog) \\
= & \quad \{ \ (12.14) \ \} \\
& Depends(g, d, Prog) \ \wedge \ d \notin R(frame, Prog) \\
\Rightarrow & \quad \{ \ (11.4)\text{: } Depends(g, d, Prog) \ \wedge \ g \in R(frame, Prog) \ \Rightarrow \ d \in R(frame, Prog) \ \} \\
& g \notin R(frame, Prog) \\
\Rightarrow & \quad \{ \ (11.1)\text{: } Resolve \text{ is monotonic, since } E \subseteq Prog \ \} \\
& g \notin R(frame, E) \qquad .
\end{array}
$$

From the last two calculations, we conclude,

$$g \in Target(gc, E) \setminus R(frame, E) \qquad ,$$

i.e., $g \in m$. From this and from (12.14), every element d of bx enjoys the property

$$d \notin E \wedge \langle \exists g \mid g \in m \triangleright \ Depends(g, d, Prog) \rangle \qquad . \tag{12.15}$$

Treatment of by

I define by as $bb \setminus bx$. Hence, bx and by do indeed partition bb. For every d in by, no procedure call in gc has a frame that mentions a g such that $Depends(g, d, Prog)$. Thus, no procedure call in gc modifies any variable which depends on d; stated differently, d does not appear in the downward closure of the frame of the specification of any procedure call in gc. So, no procedure call in gc has any effect on the value of d.

Let y be a variable updated by an assignment statement in gc. y is visible in E, because gc is contained in E. We calculate,

$$
\begin{array}{ll}
& y \in E \\
= & \quad \{ \ (12.14) \ \} \\
& y \in E \ \wedge \ d \notin E \\
\Rightarrow & \quad \{ \ \text{authenticity requirement} \ \} \\
& \neg Depends(d, y, Prog) \qquad .
\end{array}
$$

Since the only statements that may have an effect on the state are assignment statements and procedure calls, we conclude that no statement in gc has an effect on the value of d. This is so for every d in by, and thus,

$$wlp(gc, \ F'(by_0 = by), \ Prog) \ = \ F'(by_0 = by) \qquad ,$$

the right-hand side of which, by Leibniz, follows from

$$R(by_0 = by, Prog) \qquad .$$

Since $R(by, Prog)$ is a subset of z, the set of all variables, (12.5) follows.

12.4 X

I define function X, which implicitly is a function of E and $Prog$.

$X(Q) =$
Q in which every occurrence (not just free occurrences) of x is replaced by y
where
x is the list of specification variables in E and their initial-value forms, and
for each a in x, the corresponding term in y is:
a and the symbols in $Resolve(a, Prog) \setminus Resolve(a, E)$

This substitution is a rather curious one in that one identifier may be replaced by a *list* of identifiers. This causes no problem textually, but one may question the meaning of the resulting expression. If Q is a functionalized expression, then all occurrences in Q of the identifiers in list x appear as residues (I assume that identifiers are not renamed from the way they were produced as per Chapter 11). This means that they appear as parameters to $f\star$ functions or as dummies of quantifications. Replacing an identifier of the first kind with a *list* of identifiers thus alters the arity of the function. It is important that *every* occurrence of x, not just free occurrences, be replaced, so that the arity of each $f\star$ function is changed consistently; also replacing the identifiers of the second kind takes care of making y bound whenever x was.

Let me illustrate with an example. Let E be the environment

spec var a ; **var** c ; **depends** a **on** c ,

and $Prog$ the environment E extended with

var d ; **depends** a **on** d .

Then,

$$\begin{array}{ll} & X(Functionalize(\langle \forall a \mid a \leq 0 \triangleright a = 0 \rangle, E)) \\ = & \quad \{ \; Functionalize \text{ with respect to } E \; \} \\ & X(\langle \forall a, c \mid f\star a(a, c) \leq 0 \triangleright f\star a(a, c) = 0 \rangle) \\ = & \quad \{ \; X \; \} \\ & (\langle \forall a, c \mid f\star a(a, c) \leq 0 \triangleright f\star a(a, c) = 0 \rangle)[a := (a, d)] \\ = & \quad \{ \; \text{substitution} \; \} \\ & \langle \forall a, c, d \mid f\star a(a, c, d) \leq 0 \triangleright f\star a(a, c, d) = 0 \rangle \quad . \end{array} \tag{12.16}$$

I assume the order of the symbols returned by $Resolve$ ensures that $Resolve(a, E)$ is a prefix of $Resolve(a, Prog)$. I also assume operator $\setminus$ to preserve ordering. If the insertion of new symbols is then always done appropriately at the end of lists (as the example shows), then, for any list of variables w visible in E, we have

$$Resolve(w, Prog) = X(Resolve(w, E)) \quad . \tag{12.17}$$

Consequently, we have, for any Q built from symbols visible in E,

$$Functionalize(Q, Prog) = X(Functionalize(Q, E)) \quad .$$

This shows (12.6). For example, note that (12.16) equals

$$Functionalize(\langle \forall a \mid a \leq 0 \triangleright a = 0 \rangle, Prog) \quad .$$

As another consequence of (12.17), we have, for any variable v and list of variables w,

$$\langle \exists a \mid a \in w \triangleright v \in R(a, Prog) \setminus R(a, E) \rangle \Rightarrow v \in X(w) \quad . \tag{12.18}$$

For any expression e in which every variable is a program variable, $e = X(e)$. X does not introduce identifiers not found in $Prog$; hence, for the set z of all variables, $z = X(z)$, justifying (12.7). Note that X, like regular substitution, distributes over logical connectives, justifying (12.8). For quantifications, we have

$$X(\langle \mathbf{Q}\, w \mid R \triangleright T \rangle) \quad = \quad \langle \mathbf{Q}\, X(w) \mid X(R) \triangleright X(T) \rangle \quad . \tag{12.19}$$

For the substitution function,

$$X(Q[w_0 := w]) \quad = \quad X(Q)[X(w_0) := X(w)] \quad , \tag{12.20}$$

provided Q does not contain any occurrences of $X(w_0) \setminus w_0$.

12.4.0 Substitution theorem

A well-known rule in the predicate calculus, referred to as *instantiation*, is

$$[T] \Rightarrow [T[s := t]] \quad ,$$

where s denotes a list of program variables and t denotes a list of expressions. The substitution function $[s := t]$ only replaces *free* occurrences of s. If we instead let it replace bound occurrences, too, we will still find the implication shown above.

So what about the substitution function X? We could apply the above observations to X, if it weren't for the fact that X may replace one symbol with a *list* of other symbols. Hence, the following argument.

Focusing back on (12.9) and the definition of X, note that no symbol in $y \setminus x$ appears in Q. Moreover, since x is a list of residues, Q contains no information about the values of the identifiers in x; the identifiers in x occur *only* as parameters to $f\star$ functions or as bound variables in Q. Thus, the value of $[Q]$ does not depend on the exact values of x; the variables in x can be thought of as just place holders. In the same way, y will just be place holders in $[X(Q)]$, place holders appearing in the same places in $[X(Q)]$ that x does in $[Q]$.

It thus follows that $[X(Q)]$ is *true* if $[Q]$ is, proving (12.9).

12.4.1 X AND m

In this section, I prove (12.10) and (12.11).

Treatment of $Resolve(bx, Prog) \subseteq X(m)$

I prove (12.10) from (12.15). We calculate, for any x,

$$
\begin{array}{ll}
 & x \in Resolve(bx, Prog) \\
\Rightarrow & \quad \{ \quad (11.2)\text{: property of } Resolve \quad \} \\
 & \langle \exists\, d \mid d \in bx \triangleright Depends(d, x, Prog) \rangle \\
= & \quad \{ \quad (12.15)\text{: property of elements of } bx \quad \} \\
 & \langle \exists\, d \mid d \in bx \wedge d \notin E \triangleright Depends(d, x, Prog) \rangle \\
\Rightarrow & \quad \{ \quad \text{by authenticity requirement, } Depends(d, x, Prog) \wedge x \in E \Rightarrow d \in E \quad \} \\
 & \langle \exists\, d \mid d \in bx \triangleright Depends(d, x, Prog) \wedge x \notin E \rangle \\
= & \quad \{ \quad (12.15)\text{: property of elements of } bx \quad \} \\
 & \langle \exists\, d, g \mid d \in bx \wedge g \in m \triangleright Depends(g, d, Prog) \wedge Depends(d, x, Prog) \wedge x \notin E \rangle \\
\Rightarrow & \quad \{ \quad \text{transitivity of } Depends \quad \} \\
 & \langle \exists\, g \mid g \in m \triangleright Depends(g, x, Prog) \wedge x \notin E \rangle \\
= & \quad \{ \quad (11.2)\text{: } Depends(g, x, Prog) \equiv x \in Resolve(g, Prog) \quad \} \\
 & \langle \exists\, g \mid g \in m \triangleright x \in R(g, Prog) \wedge x \notin E \rangle \\
\Rightarrow & \quad \{ \quad Resolve(g, E) \subseteq E \quad \} \\
 & \langle \exists\, g \mid g \in m \triangleright x \in R(g, Prog) \setminus R(g, E) \rangle \\
\Rightarrow & \quad \{ \quad (12.18)\text{: property of } X \quad \} \\
 & x \in X(m) \qquad .
\end{array}
$$

Treatment of $m' \subseteq X(m)$

I let rt stand for $RawTarget(gc, E)$; then, rt also equals $RawTarget(gc, Prog)$. Thus, m and m' can be written as

$$
\begin{array}{l}
m = R(rt, E) \setminus R(frame, E) \\
m' = R(rt, Prog) \setminus R(frame, Prog) \qquad .
\end{array}
$$

Since we have $m \in X(m)$, consider any $v \in m' \setminus m$; I will show $v \in X(m)$. We have

$$
\begin{array}{lll}
 & v \in m' & \\
\Rightarrow & \quad \{ \quad m' \text{ and } \setminus \quad \} & \\
 & v \notin R(frame, Prog) & (12.21) \\
\Rightarrow & \quad \{ \quad (11.1)\text{: } Resolve \text{ is monotonic, since } E \subseteq Prog \quad \} & \\
 & v \notin R(frame, E) & (12.22)
\end{array}
$$

and

$$
\begin{array}{lll}
 & v \notin m & \\
= & \quad \{ \quad m \text{ and } \setminus \quad \} & \\
 & v \notin R(rt, E) \vee v \in R(frame, E) & \\
= & \quad \{ \quad (12.22) \quad \} & \\
 & v \notin R(rt, E) \qquad . & (12.23)
\end{array}
$$

I complete the proof with the following calculation.

$$
\begin{array}{ll}
 & v \in m' \\
\Rightarrow & \quad \{ \quad m' \text{ and } \setminus \quad \} \\
 & v \in R(rt, Prog) \\
= & \quad \{ \quad (11.2)\text{: property of } Resolve \quad \} \\
 & \langle \exists a \mid a \in rt \triangleright Depends(a, v, Prog) \rangle \\
= & \quad \{ \quad (11.3)\text{: } rt \subseteq R(rt, E) \quad \} \\
 & \langle \exists a \mid a \in rt \triangleright a \in R(rt, E) \wedge Depends(a, v, Prog) \rangle \\
= & \quad \{ \quad (12.24) \text{ below: } Depends(a, v, Prog) \Rightarrow a \notin R(frame, Prog) \quad \} \\
 & \langle \exists a \mid a \in rt \triangleright a \in R(rt, E) \wedge a \notin R(frame, Prog) \wedge Depends(a, v, Prog) \rangle \\
\Rightarrow & \quad \{ \quad (11.1)\text{: } Resolve \text{ is monotonic} \quad \} \\
 & \langle \exists a \mid a \in rt \triangleright a \in R(rt, E) \wedge a \notin R(frame, E) \wedge Depends(a, v, Prog) \rangle \\
= & \quad \{ \quad m \quad \} \\
 & \langle \exists a \mid a \in rt \triangleright a \in m \wedge Depends(a, v, Prog) \rangle \\
\Rightarrow & \quad \{ \quad (12.25) \text{ below: } a \in rt \wedge Depends(a, v, Prog) \Rightarrow v \in R(a, Prog) \setminus R(a, E) \quad \} \\
 & \langle \exists a \triangleright a \in m \wedge v \in R(a, Prog) \setminus R(a, E) \rangle \\
\Rightarrow & \quad \{ \quad (12.18)\text{: property of } X \quad \} \\
 & v \in X(m)
\end{array}
$$

The deferred proof obligations are shown by

$$
\begin{array}{lll}
 & true & \\
= & \quad \{ \quad (11.4)\text{: closure property of } Resolve \quad \} & \\
 & Depends(a, v, Prog) \wedge a \in R(frame, Prog) \Rightarrow v \in R(frame, Prog) & \\
= & \quad \{ \quad \text{shunting} \quad \} & \\
 & Depends(a, v, Prog) \Rightarrow a \notin R(frame, Prog) \vee v \in R(frame, Prog) & \\
= & \quad \{ \quad (12.21) \quad \} & \\
 & Depends(a, v, Prog) \Rightarrow a \notin R(frame, Prog) & (12.24)
\end{array}
$$

and

$$
\begin{array}{lll}
 & v \in R(a, Prog) \setminus R(a, E) & (12.25) \\
= & \quad \{ \quad \setminus \quad \} & \\
 & v \in R(a, Prog) \wedge v \notin R(a, E) & \\
= & \quad \{ \quad (11.2)\text{: } v \in Resolve(a, Prog) \equiv Depends(a, v, Prog) \quad \} & \\
 & Depends(a, v, Prog) \wedge v \notin R(a, E) & \\
\Leftarrow & \quad \{ \quad (11.1)\text{: } Resolve \text{ is monotonic} \quad \} & \\
 & Depends(a, v, Prog) \wedge a \in rt \wedge v \notin R(rt, E) & \\
= & \quad \{ \quad (12.23) \quad \} & \\
 & Depends(a, v, Prog) \wedge a \in rt \quad . &
\end{array}
$$

Remark 12.0. For the record, here is an example that shows that equality between m' and $X(m)$ does not hold. Let rt consist of specification variables a, b, and let $frame$ be b. Let these be the only things visible in E. In $Prog$, let both a and b depend on c.

Then, $Resolve(rt, E) \setminus Resolve(frame, E)$ is a, and $X(a)$ is a, c. However, $Resolve(rt, Prog) \setminus Resolve(frame, Prog)$ is simply a.

12.4.2 X AND *wlp*

Coping with the last proof obligation, I will now show (12.12), that is,

$$wlp(gc, X(Q), Prog) \Leftarrow X(wlp(gc, Q, E))$$

by induction over the structure of gc.

For $skip$,

$$\begin{array}{ll} & wlp(skip, X(Q), Prog) \\ = & \quad \{ \quad skip \quad \} \\ & X(Q) \\ = & \quad \{ \quad skip \quad \} \\ & X(wlp(skip, Q, E)) \qquad , \end{array}$$

and for $wrong$,

$$\begin{array}{ll} & wlp(wrong, X(Q), Prog) \\ = & \quad \{ \quad wrong \quad \} \\ & false \\ = & \quad \{ \quad X \quad \} \\ & X(false) \\ = & \quad \{ \quad wrong \quad \} \\ & X(wlp(wrong, Q, E)) \qquad . \end{array}$$

For assignment,

$$\begin{array}{ll} & wlp(v := expr, X(Q), Prog) \\ = & \quad \{ \quad := \quad \} \\ & X(Q)[v := expr] \\ = & \quad \{ \quad \text{substitution involving only program variables distributes} \\ & \qquad \text{over } X\text{, since } v := expr \text{ can be written in } E \quad \} \\ & X(Q[v := expr]) \\ = & \quad \{ \quad := \quad \} \\ & X(wlp(v := expr, Q, E)) \qquad . \end{array}$$

Next, for the hardest one, the procedure call. Let p denote a procedure declared (in E) by

spec p **is modifies** fr **requires** Pr **ensures** Po .

Let t, t', u, u', B, B' satisfy

$$\begin{array}{l}
t = Resolve(fr, E) \\
t' = Resolve(fr, Prog) \\
u = \text{list of initial-value variables from } F(Po) \\
u' = \text{list of initial-value variables from } F'(Po) \\
bs = BenSideEffects(t, E) \\
bs' = BenSideEffects(t', Prog) \qquad .
\end{array}$$

Then, realize that $t' = X(t)$ (from (12.17)), $u' = X(u)$ (from (12.6)), and $bs \Leftarrow bs'$ (from (12.13)).

$$\begin{array}{ll}
 & wlp(\mathbf{call}\, p, X(Q), Prog) \\
= & \quad \{ \quad \mathbf{call}\, p \quad \} \\
 & F'(Pr) \ \wedge \ \langle \forall\, Resolve(fr, Prog) \mid F'(bs' \ \wedge \ Po) \ \triangleright \ X(Q) \ \rangle[u'_0 := u'] \\
\Leftarrow & \quad \{ \quad bs \ \Leftarrow \ bs' \text{, and monotonicity} \quad \} \\
 & F'(Pr) \ \wedge \ \langle \forall\, Resolve(fr, Prog) \mid F'(bs \ \wedge \ Po) \ \triangleright \ X(Q) \ \rangle[u'_0 := u'] \\
= & \quad \{ \quad (12.6)\text{, and } t' \quad \} \\
 & X(F(Pr)) \ \wedge \ \langle \forall\, t' \mid X(F(bs \ \wedge \ Po)) \ \triangleright \ X(Q) \ \rangle[u'_0 := u'] \\
= & \quad \{ \quad \text{realizations about } t \text{ and } u \quad \} \\
 & X(F(Pr)) \ \wedge \ \langle \forall\, X(t) \mid X(F(bs \ \wedge \ Po)) \ \triangleright \ X(Q) \ \rangle[X(u_0) := X(u)] \\
= & \quad \{ \quad (12.19)\text{: distribution of } X \text{ over quantification} \quad \} \\
 & X(F(Pr)) \ \wedge \ X(\langle \forall\, t \mid F(bs \ \wedge \ Po) \ \triangleright \ Q \ \rangle)[X(u_0) := X(u)] \\
= & \quad \{ \quad \text{within the quantification, only } F(bs \ \wedge \ Po) \text{ can contain initial-value} \\
 & \qquad\quad \text{variables, and} \\
 & \quad\quad F(bs \ \wedge \ Po) \text{ contains no reference to any variable in } u' \setminus u\text{, and} \\
 & \quad\quad (12.20)\text{: distribution of } X \text{ over substitution} \quad \} \\
 & X(F(Pr)) \ \wedge \ X(\langle \forall\, t \mid F(bs \ \wedge \ Po) \ \triangleright \ Q \ \rangle[u_0 := u]) \\
= & \quad \{ \quad \text{distribution of } X \text{ over } \ \wedge \ \text{, and } t \quad \} \\
 & X(F(Pr) \ \wedge \ \langle \forall\, Resolve(fr, E) \mid F(bs \ \wedge \ Po) \ \triangleright \ Q \ \rangle[u_0 := u]) \\
= & \quad \{ \quad \mathbf{call}\, p \quad \} \\
 & X(wlp(\mathbf{call}\, p, Q, E))
\end{array}$$

Now for sequential composition,

$$\begin{array}{ll}
 & wlp(s; t, \ X(Q), \ Prog) \\
= & \quad \{ \quad ; \quad \} \\
 & wlp(s, wlp(t, X(Q), Prog), Prog) \\
\Leftarrow & \quad \{ \quad \text{induction hypothesis, and } wlp \text{ is monotonic} \quad \} \\
 & wlp(s, X(wlp(t, Q, E)), Prog) \\
\Leftarrow & \quad \{ \quad \text{induction hypothesis, with } Q := wlp(t, Q, E) \quad \} \\
 & X(wlp(s, wlp(t, Q, E), E) \\
= & \quad \{ \quad ; \quad \} \\
 & X(wlp(s; t, \ Q, \ E)) \qquad .
\end{array}$$

For choice,

$$
\begin{array}{ll}
 & wlp(s \mathbin{\square} t, X(Q), Prog) \\
= & \quad \{ \ \square \ \} \\
 & wlp(s, X(Q), Prog) \ \wedge \ wlp(t, X(Q), Prog) \\
\Leftarrow & \quad \{ \ \text{induction hypothesis, twice} \ \} \\
 & X(wlp(s, Q, E)) \ \wedge \ X(wlp(t, Q, E)) \\
= & \quad \{ \ (12.8)\text{:} \ X \text{ distributes over } \wedge \ \} \\
 & X(wlp(s, Q, E) \ \wedge \ wlp(t, Q, E)) \\
= & \quad \{ \ \square \ \} \\
 & X(wlp(s \mathbin{\square} t, Q, E)) \qquad .
\end{array}
$$

For the guard statement,

$$
\begin{array}{ll}
 & wlp(g \rightarrow s, X(Q), Prog) \\
= & \quad \{ \ \rightarrow \ \} \\
 & g \ \Rightarrow \ wlp(s, X(Q), Prog) \\
\Leftarrow & \quad \{ \ \text{induction hypothesis} \ \} \\
 & g \ \Rightarrow \ X(wlp(s, Q, E)) \\
= & \quad \{ \ X \text{ distributes over logical connectives, and} \\
 & \quad \phantom{\{} \ g \text{ only mentions program variables} \ \} \\
 & X(g \ \Rightarrow \ wlp(s, Q, E)) \\
= & \quad \{ \ \rightarrow \ \} \\
 & X(wlp(g \rightarrow s, Q, E)) \qquad .
\end{array}
$$

Finally, for block,

$$
\begin{array}{ll}
 & wlp([\![\ y \bullet s \ ]\!], X(Q), Prog) \\
= & \quad \{ \ [\![\bullet]\!] \ \} \\
 & \langle \forall y \triangleright wlp(s, X(Q), Prog) \ \rangle \\
\Leftarrow & \quad \{ \ \text{induction hypothesis} \ \} \\
 & \langle \forall y \triangleright X(wlp(s, Q, E)) \ \rangle \\
= & \quad \{ \ X \text{ distributes over universal quantification where dummy is} \\
 & \quad \phantom{\{} \ \text{not name of specification variable} \ \} \\
 & X(\langle \forall y \triangleright wlp(s, Q, E) \ \rangle) \\
= & \quad \{ \ [\![\bullet]\!] \ \} \\
 & X(wlp([\![\ y \bullet s \ ]\!], Q, E)) \qquad .
\end{array}
$$

It's a wrap.

12.5 Epilogue

Now that I have proven the soundness theorem, let's look back at the proof to see what is used where.

Firstly, note that residues are used with every occurrence of X, and thus appear everywhere in the proof. If instead of residues some extra requirements were used, the proof that those requirements ensure soundness would require quite a different proof.

Secondly, as foretold by Remark 9.2, the proof does not make use of the structure of representations of specification variables. In particular, the choice of abstraction functions *vs.* abstraction relations does not surface as being important to the proof. Realize that the soundness proof just provides a bridge between a "one"-module proof and an all-modules proof; hence, Remark 9.2 does not imply anything positive about the effects of using abstraction relations, thus letting $f\star$ functions be nondeterministic, in a "one"-module proof.

Thirdly, the proof does not rely on whether or not dependencies are cyclic. It is not clear, however, for what purpose cyclic dependencies are useful.

> **Remark 12.1.** If imports are acyclic, then the visibility and authenticity requirements dictate that all cycles among dependencies be placed in one unit.

Finally, the visibility requirement plays a rôle in proving the monotonicity property (12.13). This property is used in the proofs of $b \subseteq b'$ (12.4) and in the distribution of X over *wlp* for procedure calls (12.12).

The authenticity requirement comes into play in showing that program variables updated by assignment statements in *gc* have only benevolent side effects, *i.e.*, do not alter the value of, any specification variable in $b' \setminus b$. The authenticity requirement is applied twice in the proof, once for each partition of $b' \setminus b$, *by* and *bx*.

Notice that the visibility and authenticity requirements are used in separate places in the proof. Hence, for the proof, it is preferable to have a division between visibility and authenticity requirements, rather than making use of the convention presented in Section 10.3.

Chapter 13

depends in perspective

In this chapter, I give a flavor of the successes and shortcomings of **depends**. I start by showing how **depends** can be used to specify *consumer* objects. Then, I discuss some problems for which **depends** is not the whole solution. I don't solve these problems here, but I conclude the chapter with some insights into the problems.

13.0 Specification of a consumer

In this section, I describe a difficult specification problem for which I then show a solution in my formalism.

13.0.0 Consumers

Consider the following unit, which provides a type of so-called *consumer* objects.

```
unit Consumer is
   type T ;
   spec Consume(s : seq[char] ; t : T) is
      (* for each character in s, invoke t.consume *) ;
   method t : T spec consume(ch : char) is
      (* ... *)
end
```

The idea is to call *Consume* with a character sequence and an object t of some subtype of T. This invokes *t.consume* for each character of the given string. (In functional programming, an operation like *Consume* is called a *map* (see, *e.g.*, [6]).)

Another example of a consumer takes a *reader*, *i.e.*, an input stream (*cf.* [9] and Chapter 10), and passes character sequences read from the reader to *t.consume* in arbitrary-size pieces.

The task is to write the specification of procedure *Consume* and method *consume*. The problem is —and it is for this reason that I earlier referred to this problem as being difficult— doing so without knowing what *t.consume* requires, modifies, or ensures for all subtypes of T.

Remark 13.0. In his thesis, Jackson says for Aspect that "it is not clear how to solve this problem" [41, Sec. 8.3], and leaves the problem without a solution.

The specification must admit a provably correct implementation and must be flexible enough to be useful for subtypes of T.

13.0.1 A SOLUTION

I present a solution to the problem. For simplicity, I let the postcondition of the consume method be *true*. Other postconditions can be handled similarly to the way I handle the precondition.

Remark 13.1. For the purposes of extended static checking (see Section 0.0), using a postcondition of *true* in cases like this is quite common. The precondition, and the fact that some invariant properties are preserved, are more important.

I introduce two properties of consumers, *valid* and *state*.

spec var $valid : T^- \rightarrow$ **bool** ;
spec var $state : T^- \rightarrow$ **any**

The type **any** indicates that the particular value of $state[t]$ is not relevant. Consequently, no **rep** is given for $state[t]$, but **depends** clauses are still used to declare the concrete variables on which $state[t]$ depends. This allows subtypes of T to declare the variables that make up their states. These concrete variables can then be modified when *state* is allowed to be modified. This important mechanism is not available in classical data refinement [38].

I write a specification of the *consume* method in terms of the properties *valid* and *state*.

method $t : T$ **spec** $consume(ch :$ **char**$)$ **is**
  requires $valid[t]$
  modifies $state[t]$

An invocation of *consume* requires that t be valid. This essentially means that t satisfies its *object invariant* (*cf.* [64] or *type constraints* in [57]). Unlike [64, 57], I make it explicit when the invariant should hold by explicitly stating $valid[t]$ as a precondition. Since $valid[t]$ is not mentioned in the **modifies** clause, the value of $valid[t]$ must not be changed by the method; hence, the object will be valid upon exit, too. This makes object invariants a convention rather than a canned feature of the specification language (see also Remark 10.3).

The specification of *consume* allows the state of object t to be modified. The only restriction on this modification is that the values of properties like *valid* that are not listed in the **modifies** clause are unchanged.

The specification of *Consume* can now be written. As it turns out, this specification coincides with that of method *consume*.

> **spec** *Consume*$(s : \textbf{seq}[\textbf{char}]\ ;\ t : T)$ **is**
> **requires** *valid*$[t]$
> **modifies** *state*$[t]$

These specifications allow an implementation of *Consume* to iterate over the characters of s, passing each one, in sequence, to *t.consume*.

> **impl** *Consume*$(s : \textbf{seq}[\textbf{char}]\ ;\ t : T)$ **is**
> **if** $s \neq$ “” **then** $t.consume(first(s))\ ;\ Consume(rest(s), t)$ **fi**

Without stating a stronger postcondition, this particular specification is not strong enough for full verification. For example, an implementation of *Consume* meets its specification even if it invokes the method twice for some characters of s, never for others, and the invocations of *consume* for different characters of s are not in the same order as the order in which they appear in s. Other correct implementations may invoke *consume* with characters not in s, or may never call *consume* at all. However, for extended static checking, this specification may still be strong enough, depending on the assumptions made about the consumer after the call to *Consume*.

Each subtype defines for itself what it means for one of its object to be valid and what the state of one of its objects is. Hence, the solution works for any consumer subtype.

13.1 Shortcomings of **depends**

From what we have seen, **depends** is promising, but it is not the end of the story. There are more problems to be solved, because the visibility and authenticity requirements are too strong for the solution to be adequate.

The solution is adequate as long as there is only one level of specification variables, *i.e.*, so long as no specification variable depends on another specification variable. However, when one module is implemented in terms of another, there is usually more than one level.

Let me give an example of a problem where the visibility and authenticity requirements are too strict. The *TextWrImpl* unit in Chapter 10 declares the *flushed* data field of each text writer to be of type **seq**[**char**]. The **seq**[**char**] type available through the programming language at hand (*e.g.*, `TEXT` in Modula-3) may not provide the most efficient implementation for the task in *TextWrImpl*, or the language may not directly provide this type at all. We may then want to use a custom-programmed implementation of sequences of characters.

Consider a unit *FastSeq*, declared as follows.

> **unit** *FastSeq* **is**
>   **type** T;
>   **spec** *contents* : $T^- \to$ **seq**[**char**];
>
>   **spec** *Init*$(t : T)$ **is**
>   **modifies** *contents*$[t]$
>   **ensures** *contents*$[t] =$ “” ;
>
>   **spec** *result* : **seq**[**char**] := *Contents*$(t : T)$ **is**
>   **ensures** *result* = *contents*$[t]$;
>
>   **spec** *Append*$(t : T \; ; \; ch :$ **char**$)$ **is**
>   **modifies** *contents*$[t]$
>   **ensures** $contents[t] = contents_0[t] + ch$
> **end**

Such a unit allows a change of unit *TextWrImpl* to

> **unit** *TextWrImpl* **import** *Wr*, *WrFriends*, *TextWr*, *FastSeq* **is**
>   **var** *flushed* : $TextWr.T^- \to FastSeq.T$;
>   **depends** $Wr.target[t : TextWr.T]$ **on** $FastSeq.contents[flushed[t]]$;
>   **rep** $Wr.target[t : TextWr.T]$ **is**
> $Wr.target[t] = FastSeq.contents[flushed[t]] + Wr.buff[t]$;
>   ⋮
> **end**

Here, *Wr.target* is declared to depend on *FastSeq.contents*. But according to the (convention suggested by the) two requirements, this dependency must be declared in unit *FastSeq*. That means that *FastSeq* needs to import *Wr* and give the dependency. The same holds for any other similar client of *FastSeq*. Thus, *FastSeq* needs to import all of these clients! That is unreasonable, because the implementor of *FastSeq* cannot anticipate all of *FastSeq*'s clients, so with every new client, *FastSeq* needs to be updated. This makes it practically impossible to put *FastSeq* in a library.

Furthermore, in order for *FastSeq* to be able to declare the dependency shown in *TextWrImpl* above, the data field *flushed* must be visible in *FastSeq*. This goes beyond the notion of “friends” interfaces and violates the notion of data hiding.

The problem described also surfaces, for example, in the consumer example in Section 13.0 if a consumer subtype depends on a data field like *Wr.target* or *FastSeq.contents*.

Let us take a closer look at the shape of the **depends** clause that's causing problems. In examples, I have freely used **depends** clauses of the form

> **depends** $a[t : T]$ **on** $c[t]$.

I call this a *pointwise dependency*, because a at a particular index depends on c at some particular index. Here, the two indices are the same. Chapters 11 and 12 explore dependencies only between entire variables, as in

> **depends** a **on** c .

Understanding pointwise dependencies and developing a methodology for their use are on the path to a solution to the problem I've described in this section.

Another closer look at the shape of the **depends** clause that presents the challenge shows

$$\textbf{depends}\ a[t : T]\ \textbf{on}\ c[b[t]] \qquad .$$

That is, a depends on c pointwise, but not at the same points. Instead, b serves as a function from the point of a to the dependent point of c. All three of a, b, c need to be visible in order to state the dependency. This reminds us of the visibility requirements. Furthermore, the intermediary variable b is declared in the same unit as the dependency itself. This leaves us with a scent of the authenticity requirement. These observations give us hope that there may be a way to modify the visibility and authenticity requirements to fit the needs described here. This is an open problem. In the next section, I discuss some considerations of its solution.

13.2 Private values

In the example presented in the previous section, the $\mathit{FastSeq.T}$ value of $\mathit{flushed}[t]$ can be thought of as being "private" to the unit (or to the object t). In this section, I develop the idea of *private values* further. I do not present a solution to the problem of writing specifications involving private values, but I discuss some issues that a solution should take into account.

Values may be "private" not just to modules, but also to procedures (as I show later in this chapter) and possibly also to objects or object types (not shown here).

13.2.0 A given specification

Consider a unit A and its friends interface $\mathit{AFriends}$.

unit *A* **is**
  type *T* ;
  spec var *valid* : $T^- \to$ **bool** ;
  spec var *state* : $T^- \to$ **any** ;

  spec *Init*($t : T$) **is**
    modifies *state*[*t*], *valid*[*t*]
    ensures *valid*[*t*] ;

  spec *Update*($t : T$) **is**
    requires *valid*[*t*]
    modifies *state*[*t*] ;

  spec *Destroy*($t : T$) **is**
    modifies *state*[*t*], *valid*[*t*]
end

$$
\begin{array}{l}
\textbf{unit } \mathit{AFriends} \textbf{ import } A \textbf{ is} \\
\quad \textbf{var } \mathit{previous} : A.T \ ; \\
\quad \textbf{depends } A.\mathit{state} \textbf{ on } \mathit{previous} \ ; \\
\quad \textbf{depends } A.\mathit{valid} \textbf{ on } \mathit{previous} \\
\textbf{end}
\end{array}
$$

Also consider B, a unit declared as follows.

$$
\begin{array}{l}
\textbf{unit } B \textbf{ is} \\
\quad \textbf{type } T \ ; \\
\quad \textbf{spec var } \mathit{valid} : T^{-} \rightarrow \textbf{bool} \ ; \\
\quad \textbf{spec var } \mathit{state} : T^{-} \rightarrow \textbf{any} \ ; \\
\quad \textbf{spec } \mathit{Init}(t : T) \textbf{ is} \\
\quad\quad \textbf{modifies } \mathit{state}[t], \mathit{valid}[t] \\
\quad\quad \textbf{ensures } \mathit{valid}[t] \ ; \\
\quad \textbf{spec } \mathit{Update}(t : T) \textbf{ is} \\
\quad\quad \textbf{requires } \mathit{valid}[t] \\
\quad\quad \textbf{modifies } \mathit{state}[t] \\
\textbf{end}
\end{array}
$$

The implementation of B reveals one of type $B.T$'s data fields.

$$
\begin{array}{l}
\textbf{unit } \mathit{BImpl} \textbf{ import } B, A \textbf{ is} \\
\quad \textbf{private var } x : B.T^{-} \rightarrow A.T \ ; \\
\quad\quad \vdots \\
\textbf{end}
\end{array}
$$

Notice that B is a regular client of A that does not import $\mathit{AFriends}$.

The keyword **private** is intended to indicate that field x is never "imported" into or "leaked" from this module. (I intentionally omit the exact definitions of those terms—the inclusion of these words is just supposed to convey a flavor of the full meaning.) The motivation for this is as follows. The state and validity of a $B.T$ object depends on the state and validity of the fields of a $B.T$. Thus, BImpl gives the following information about its specification variables.

$$
\begin{array}{l}
\textbf{depends } B.\mathit{valid}[t : B.T] \textbf{ on } A.\mathit{valid}[x[t]] \ ; \\
\textbf{depends } B.\mathit{state}[t : B.T] \textbf{ on } A.\mathit{valid}[x[t]] \ ; \\
\textbf{rep } B.\mathit{valid}[t : B.T] \textbf{ is } \mathit{valid}[t] = A.\mathit{valid}[x[t]]
\end{array}
$$

But this violates the visibility and authenticity requirements. Our first reaction is that the violation is for silly reasons; if $B.T$ happens to use an $A.T$ in its private implementation, why does this need to be advertised in A, which knows nothing about B? The **private** keyword used in conjunction with the declaration of field x is an attempt at allowing the implementation to depart from the visibility and authenticity requirements. The details of **private** are still under experimentation.

Unit *BImpl* also includes the implementation of the procedures declared in B.

impl $B.Init(t : B.T)$ **is**
  call $A.init(x[t])$;
impl $B.Update(t : B.T)$ **is**
  call $A.Update(x[t])$

13.2.1 A violation of soundness

Let's take a look at how a client may use A and B.

unit *Client* **import** *A*, *AFriends*, *B* **is**
  spec $P()$ **is**
    modifies *A.state*, *A.valid*, *B.valid*, *B.state* ;
  impl $P()$ **is**
    $[\![$ $b : B.T \bullet b :=$ **new**$(B.T)$
    ; $B.Init(b)$
    ; **if** *AFriends.previous* $\neq$ **nil then** *A.Destroy*(*AFriends.previous*) **fi**
    ; $B.Update(b)$
    $]\!]$
end

The question here is: Does P's call to *B.Update* meet its required precondition? From the information given and the rules from Chapter 11, the verification would indeed verify P as having the required precondition of *B.Update*. But consider an implementation of $A.init(t)$ that, in addition to initializing t, sets *AFriends.previous* to t. Then we do *not* want the verification process to deem the implementation of procedure P correct, because there is no guarantee that the call to *B.Update* actually meets its precondition.

The problem seems to be that although *BImpl* vowed (by using the keyword **private**) not to leak a value of an x field, *BImpl* passed such a value to the procedures of A, and A never made a promise not to leak such values. Hence, we don't want *BImpl* to get away with declaring x as **private** unless A guarantees that instances of $A.T$ are "*privatizable*".

To recap, I am discussing the circumstances under which the declaration of dependencies of a specification variable are allowed to depart from the visibility and authenticity requirements. In the above, in order for $B.valid[t]$ to be declared to depend on $A.valid[x[t]]$, *A.valid* must have been declared to be privatizable. If $A.valid[t]$ is privatizable, its potential dependencies are restricted. A research goal is to find a good set of rules for this. For example, one may require that $A.valid[t]$ must only depend on entities of the form $w[t]$ where w is also privatizable. As a base case of the definition of privatizable, all program (*i.e.*, non-abstract) data fields of a type are privatizable.

13.2.2 Private values and **new**

I present an issue related to the one above. Consider a unit S and its implementation.

unit *S* **is**	**unit** *SImpl* **import** *S* **is**
spec *kip*( ) **is**	**impl** *kip*( ) **is**
modifies (∗ nothing ∗)	$[\![\ a \bullet a := \mathbf{new}(A.T)\ ;\ A.Init(a)\ ]\!]$
end	**end**

A question to ask is: Does *SImpl* contain a correct refinement of *S.kip*? Since it has an effect on *A.state* and *A.valid*, the implementation would not be considered correct by the rules from Chapter 11. Nevertheless, one may be inclined to answer, "Yes, this is a correct refinement, because the value of a is conceived within this procedure (*i.e.*, it is not imported, and since the procedure has no return value, a is also not leaked from the procedure) and should thus be considered a private value of the procedure".

In response to this answer, let me define another unit.

```
unit M import S, A, AFriends is
   spec P() is
      ensures true ;
   impl P() is
      [[  p : A.T • p := AFriends.previous
      ;   S.kip()
      ;   if p ≠ AFriends.previous then wrong fi
      ]]
end
```

Using the rules from Chapter 11, the implementation of *M.P* is considered correct. Thus, also validating *S.kip* would not be sound, because then a trace of *M.P* would actually go wrong.

Therefore, the quoted claim above is flawed. The problem, as with B and *BImpl* above, is that *A.state* and *A.valid* are not guaranteed to be privatizable.

13.2.3 Summary

In this section, I have argued for a notion of "privatizable". If w is privatizable and t is a private value of a procedure, then modification of $w[t]$ is allowed, even if w is not mentioned in the frame. Similarly, if t is a private value of a unit, then dependencies on $w[t]$ declared in that unit are allowed, even if w is declared elsewhere. The exact details of "private values" and "privatizable" are still under experimentation, and soundness will be a function of these details.

Part IV

Epilogue

Chapter 14

Summary

In summary, this thesis is about the correctness of programs, and is in the direction of making the specification and verification of large programs —of the kinds that are written in practice— feasible. In particular, I deal with sequential programs that can raise and handle exceptions; programs whose expressions can be partial, can contain short-circuit boolean operators, and can have side effects; programs whose data structures include arrays, sets, records, references (pointers), and objects with methods and inheritance; and programs that achieve data hiding by being organized into modules.

The thesis explores several aspects of the semantics of programs with exceptions, and unveils the algebraic cosmos on which the foundation of this semantics rests. The thesis also takes a fresh look at object-oriented programs, and proposes a simple notion of their mathematical meaning.

Striving toward making large programs more reliable, I confront the specification and verification of modular programs, with the goal of achieving sound modular verification. This thesis solves one of the problems and reports on others that are still open.

There is more work to be done before practical tools that aid in the construction of large programs that are correct are used routinely by programmers. Type-checking compilers have become commonplace, and it seems only natural that a compiler or other tool for extended static checking will be next. Through something like extended static checking, it is my hope that it won't be too long before the science of program correctness will become a practical aid in the everyday life of programmers, so that we all will write correct programs more often.

Bibliography

[0] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[1] American National Standards Institute, Inc. *The Programming Language Ada Reference Manual*, volume 155 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, Germany, 1983. ANSI/MIL-STD-1815A-1983.

[2] R.-J.R. Back. *On the Correctness of Refinement Steps in Program Development*. Ph.D. thesis, University of Helsinki, 1978. Report A-1978-4.

[3] R.-J.R. Back. Data refinement in the refinement calculus. In *Proceedings 22nd Hawaii International Conference of System Sciences*, Kailua-Kona, January 1989.

[4] R.-J.R. Back and M. Karttunen. A predicate transformer semantics for statements with multiple exits. University of Helsinki, unpublished manuscript.

[5] R.C. Backhouse. *Program Construction and Verification*. Series in Computer Science. Prentice-Hall International, 1986.

[6] R.J. Bird and P. Wadler. *Introduction to Functional Programming*. Series in Computer Science. Prentice-Hall International, 1988.

[7] G. Birkhoff. *Lattice Theory*. Colloquium Publications, Volume 25. American Mathematical Society, 1967.

[8] F. Bourdoncle. Interprocedural abstract interpretation of block structured languages with nested procedures, aliasing and recursivity. In P. Deransart and J. Małuszyński, editors, *Programming Language Implementations and Logic Programming: Proceedings / International Workshop PLILP '90, Linköping, Sweden, August 20–22, 1990*, volume 456 of *Lecture Notes in Computer Science*, pages 307–323. Springer-Verlag, 1990.

[9] M.R. Brown and G. Nelson. I/O streams: Abstract types, real programs. In G. Nelson, editor, *Systems Programming with Modula-3*, Series in Innovative Technology, pages 130–169. Prentice-Hall, Englewood Cliffs, NJ, 1991.

[10] M. Burrows and G. Nelson. LIM. Private communications, 1993.

[11] W. Chen and J.T. Udding. Towards a calculus of data refinement. In J.L.A. van de Snepscheut, editor, *Mathematics of Program Construction*, volume 375 of *Lecture Notes in Computer Science*. Springer-Verlag, June 1989.

[12] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the 4th ACM Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, 1977.

[13] F. Cristian. Correct and robust programs. *IEEE Transactions on Software Engineering*, 10:163–174, 1984.

[14] O.-J. Dahl, B. Myhrhaug, and K. Nygaard. The Simula 67 common base language. Technical Report S-22, Norwegian Computing Centre, Oslo, 1970.

[15] D. Detlefs and G. Nelson. The BOLT specification language. Internal note, DEC SRC, 1994.

[16] E.W. Dijkstra. A constructive approach to the problem of program correctness. *BIT*, 8:174–186, 1968.

[17] E.W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, Englewood Cliffs, NJ, 1976.

[18] E.W. Dijkstra. *Selected Writings on Computing: A Personal Perspective*. Texts and Monographs in Computer Science. Springer-Verlag, 1982.

[19] E.W. Dijkstra. The unification of three calculi. In M. Broy, editor, *Program Design Calculi*, NATO ASI Series F. Springer-Verlag, 1993.

[20] E.W. Dijkstra and W.H.J. Feijen. *A Method of Programming*. Addison-Wesley, 1988.

[21] E.W. Dijkstra and C.S. Scholten. *Predicate Calculus and Program Semantics*. Texts and Monographs in Computer Science. Springer-Verlag, 1990.

[22] R.M. Dijkstra. DUALITY: A simple formalism for the analysis of UNITY. Technical Report CS-R9404, University of Groningen, April 1994.

[23] M.A. Ellis and B. Stroustrup. *The Annotated C++ Reference Manual*. Addison-Wesley Publishing Company, 1990.

[24] P.H.B. Gardiner and C.C. Morgan. Data refinement of predicate transformers. *Theoretical Computer Science*, 87:143–162, 1991.

[25] A.J.M. van Gasteren. *On the Shape of Mathematical Arguments*, volume 445 of *Lecture Notes in Computer Science*. Springer-Verlag, 1990.

[26] S.M. German. *Verifying the Absence of Common Runtime Errors in Computer Programs*. Ph.D. thesis, Harvard University, 1981. Also available as Technical Report STAN-CS-81-866, Stanford University, 1981.

[27] W.W. Gibbs. Software's chronic crisis. *Scientific American*, pages 86–95, September 1994.

[28] A. Goldberg and D. Robson. *Smalltalk-80: The Language and Its Implementation*. Addison-Wesley Series in Computer Science. Addison-Wesley, Reading, MA, 1983.

[29] D. Gries. *The Science of Programming*. Texts and Monographs in Computer Science. Springer-Verlag, 1981.

[30] D. Gries and J. Prins. A new notion of encapsulation. In *Proceedings of the ACM SIGPLAN 85 Symposium on Language Issues in Programming Environments, Seattle, WA, 25–28 June 1985*. ACM SIGPLAN Notices, 20(7):131–139, July 1985.

[31] D. Gries and F.B. Schneider. *A Logical Approach to Discrete Math*. Texts and Monographs in Computer Science. Springer-Verlag, 1994.

[32] D. Guaspari, C. Marceau, and W. Polak. Formal verification of Ada programs. *IEEE Transactions on Software Engineering*, 16(9):1058–1075, September 1990.

[33] J.V. Guttag, J.J. Horning, and J.M. Wing. The Larch family of specification languages. *IEEE Software*, 2(5):24–36, September 1985.

[34] E.C.R. Hehner. *A Practical Theory of Programming*. Texts and Monographs in Computer Science. Springer-Verlag, 1993.

[35] A. Heydon and G. Nelson. The Juno-2 constraint-based drawing editor. Research Report 131a, DEC SRC, 130 Lytton Ave., Palo Alto, CA 94301, U.S.A., December 1994.

[36] C.A.R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580,583, October 1969.

[37] C.A.R. Hoare. Notes on data structuring. In O.-J. Dahl, E.W. Dijkstra, and C.A.R. Hoare, editors, *Structured Programming*, pages 83–174. Academic Press, 1972.

[38] C.A.R. Hoare. Proof of correctness of data representations. *Acta Informatica*, 1(4):271–81, 1972.

[39] C.A.R. Hoare and J. He. The weakest prespecification. *Fundamenta Informaticae*, IX:51–84, 1986.

[40] C.A.R. Hoare and N. Wirth. An axiomatic definition the programming language PASCAL. *Acta Informatica*, 2(4):335–355, 1973.

[41] D. Jackson. *Aspect: A Formal Specification Language for Detecting Bugs.* Ph.D. thesis, Massachusetts Institute of Technology, 1992. Technical Report MIT/LCS/TR-543.

[42] K. Jensen, N. Wirth, A.B. Mickel, and J.F. Miner. *PASCAL: User Manual and Report.* Springer-Verlag, 4th edition, 1991. ISO Pascal Standard.

[43] C.B. Jones. *Systematic Software Development Using VDM.* Prentice-Hall, 1986.

[44] D.H. Kemp and G. Goodfellow. AT&T crash, 15 Jan 90: The official report. *ACM SIGSOFT Software Engineering Notes*, 15(2):11–12, April 1990.

[45] B.W. Kernighan and D.M. Ritchie. *The C Programming Language.* Prentice-Hall, 1978.

[46] S. King and C.C. Morgan. Exits in the refinement calculus. Technical Report PRG-TR-6-93, Programming Research Group, Oxford, 1993.

[47] B.W. Lampson, J.J. Horning, R.L. London, J.G. Mitchell, and G.J. Popek. Report on the programming language Euclid. Technical Report CSL-81-12, Xerox PARC, 3333 Coyote Hill Rd., Palo Alto, CA 94304, U.S.A., October 1981. An earlier version of this report appeared in ACM SIGPLAN Notices, 12(2), February 1977.

[48] G.T. Leavens. Modular specification and verification of object-oriented programs. *IEEE Software*, pages 72–80, July 1991.

[49] K.R.M. Leino. Multicomputer programming with Modula-3D. Technical Report Caltech-CS-TR-93-15, California Institute of Technology, 1993. Master thesis.

[50] K.R.M. Leino. Constructing a program with exceptions. *Information Processing Letters*, 53(3), 1995.

[51] K.R.M. Leino and J.L.A. van de Snepscheut. Semantics of exceptions. In E.-R. Olderog, editor, *Proceedings of the IFIP WG2.1/WG2.2/WG2.3 Working Conference on Programming Concepts, Methods, and Calculi, San Miniato, Italy, 6–10 June 1994*, pages 447–466. Elsevier, 1994.

[52] N.G. Leveson and C.S. Turner. An investigation of the Therac-25 accidents. *IEEE Computer*, 26(7):18–41, July 1993.

[53] J.H. van Lint and R.M. Wilson. *A Course in Combinatorics.* Cambridge University Press, New York, 1992.

[54] B. Liskov, R. Atkinson, T. Bloom, E. Moss, J.C. Schaffert, R. Scheifler, and A. Snyder. *CLU Reference Manual.* Springer-Verlag, 1981.

[55] B. Liskov and J. Guttag. *Abstraction and Specification in Program Development.* The MIT Electrical Engineering and Computer Science Series. MIT Press, Cambridge, MA, 1986.

[56] B.H. Liskov and J.M. Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems*, To appear.

[57] D.C. Luckham. *Programming with Specifications: An Introduction to Anna, a Language for Specifying Ada Programs.* Texts and Monographs in Computer Science. Springer-Verlag, 1990.

[58] D.C. Luckham, S.M. German, F.W. von Henke, R.A. Karp, P.W. Milne, D.C. Oppen, W. Polak, and W.L. Scherlis. Stanford Pascal Verifier user manual. Technical Report STAN-CS-79-731, Stanford University, 1979.

[59] J.J. Lukkien. An operational semantics for the guarded command language. In R.S. Bird, C.C. Morgan, and J.C.P. Woodcock, editors, *Mathematics of Program Construction*, volume 669 of *Lecture Notes in Computer Science*, pages 233–249. Springer-Verlag, 1993.

[60] M.S. Manasse and C.G. Nelson. Correct compilation of control structures. Technical report, AT&T Bell Laboratories, September 1984.

[61] R. Manohar and K.R.M. Leino. Theory and use of conditional composition. Technical Report Caltech-CS-TR-94-12, California Institute of Technology, 1994.

[62] C. Marceau. Penelope reference manual, version 3-3. Technical Report TM-94-0040, Odyssey Research Associates, Inc., 301 Dates Drive, Ithaca, NY 14850-1326, U.S.A., July 1994.

[63] A.J. Martin and J.L.A. van de Snepscheut. Design of synchronization algorithms. In M. Broy, editor, *Constructive Methods in Computing Science*, volume 55 of *NATO ASI Series F: Computers and Systems Sciences*, pages 447–478. Springer-Verlag, 1989.

[64] B. Meyer. *Object-oriented Software Construction.* Series in Computer Science. Prentice-Hall International, New York, 1988.

[65] C.C. Morgan. Auxiliary variables in data refinement. *Information Processing Letters*, 29(6):293–296, December 1988.

[66] C.C. Morgan. Data refinement using miracles. *Information Processing Letters*, 26(5):243–246, January 1988.

[67] C.C. Morgan. The specification statement. *ACM Transactions on Programming Languages and Systems*, 10(3):403–419, July 1988.

[68] C.C. Morgan and K.A. Robinson. Specification statements and refinement. *IBM Journal of Research and Development*, 31(5):546–555, September 1987.

[69] G. Nelson. Techniques for program verification. Technical Report CSL-81-10, Xerox PARC, 3333 Coyote Hill Rd., Palo Alto, CA 94304, U.S.A., June 1981. Ph.D. thesis.

[70] G. Nelson. A generalization of Dijkstra's calculus. *ACM Transactions on Programming Languages and Systems*, 11(4):517–561, 1989.

[71] G. Nelson, editor. *Systems Programming with Modula-3.* Series in Innovative Technology. Prentice-Hall, Englewood Cliffs, NJ, 1991.

[72] T. Nipkow. Non-deterministic data types. *Acta Informatica*, 22:629–661, 1986.

[73] F. Rubin. "GOTO considered harmful" considered harmful. *Communications of the ACM*, 30(3):195–196, March 1987. See also responses to this letter in the May through August, November, and December issues.

[74] N. Shankar, S. Owre, and J.M. Rushby. *The PVS Proof Checker: A Reference Manual (Draft).* Computer Science Laboratory, SRI International, Menlo Park, CA, February 1993.

[75] J.L.A. van de Snepscheut. *Trace Theory and VLSI Design.* Ph.D. thesis, Technische Hogeschool Eindhoven, 1983.

[76] J.L.A. van de Snepscheut. On lattice theory and program semantics. Technical Report Caltech-CS-TR-93-19, California Institute of Technology, 1993.

[77] J.L.A. van de Snepscheut. *What Computing Is All About.* Texts and Monographs in Computer Science. Springer-Verlag, 1993.

[78] M. T. Vandevoorde. *Exploiting Specifications to Improve Program Performance.* Ph.D. thesis, Massachusetts Institute of Technology, 1994. Technical Report MIT/LCS/TR-598.

[79] N. Wirth. The programming language PASCAL. *Acta Informatica*, 1(1):35–63, 1970.

[80] N. Wirth. Program development by stepwise refinement. *Communications of the ACM*, 14:221–227, 1971.

[81] N. Wirth. *Programming in Modula-2.* Texts and Monographs in Computer Science. Springer-Verlag, 1983.

[82] J. von Wright. *A Lattice-Theoretical Basis for Program Refinement.* Ph.D. thesis, Åbo Akademi, 1990.

Index

;, *see* sequential composition
[:=], *see* substitution
[ ], *see* everywhere operator
$\wedge$ (conjunction), 5
□, *see* box operator
$+\!\!+$ (concatenation), 98
⊠, *see* else operator
$\cdot$ (multiplication), 7
$\neg$ (negation), 4
$\vee$ (disjunction), 5
$\langle\ \mid\ \triangleright\ \rangle$, *see* quantifications
$\sqsubseteq$, *see* refinement
$\{\ \mid\ \triangleright\ \}$, *see* set constructor
<: (subtype relation), 78
$\langle\!\circ\!\rangle$, *see* double composition
$\mapsto$, 21
$\lceil\ \rceil$, *see* ceiling
$\langle\!\circ$, *see* left composition
$\lfloor\ \rfloor$, *see* floor
$\circ\!\rangle$, *see* right composition
$\sim$, *see* transposition
$\times$, 68
$\rightarrow$ (guard), 18
$\rightarrow$ (map), 68
$\lhd$, *see* exceptional composition
$\star$, 74, 112, 114
$^-$ shorthand, 74
0, subscripting a variable by, *see* initial-value variables

abort, 22, 42
abstract interpretation, 22
abstract variables, 84, 111
    and assignment (:=), 107
    and refinement, 86
    definition of usage in frames, 101
    interpretation of, in predicates, 101
    interpreted as explicit functions, 107
    local, 84
    mixed with program variables, 88
    origin of name, 85
    representation of, 85
abstraction, *see* data abstraction
abstraction functions, 85, 142
    and subtyping, 89
    axioms resulting from, 117
    in scope *vs.* out of scope, 104
    partial, 89
    totality of, 86
    vs. abstraction relations, 85, 142
abstraction relations, 85, 142
acknowledgements, v
Ada [1], 8, 12, 56, 59, 60, 68, 69, 74, 89, 96, 108
    initialization of variables in, 69
    references in, 69, 74
algebra over functions of two arguments, 12, 17, 26–35
    applications of, 17, 35
    generalization of, 35
    motivation for, 17
    unit elements in, 27
aliasing, 74, 83
allocated type, *see* dynamic type
allocation, 74, 150, 151
alternative statements (`IF THEN`), 59
Anna [57], 97, 108, 109
arity
    change of, 135
array dereferencing, 68
array index out-of-bounds errors, *see*

run-time errors
array types, 71
arrays, 66
multi-dimensional, 72
Aspect [41], 106, 145
assert statement (**assert**), 19, 61
assertions
failing, *see* run-time errors
assignment (:=), 16, 38, 112
and abstract variables, 107
narrowing in, 70
shorthand for maps (arrays), 68
AT&T crash, 1990, 0
authentic abstractions, 103
authentic variables, 103
authenticity property, 104
authenticity requirement, 104, 106, 128, 134, 137, 146
deviating from, 150
dispensing with, 104
enforcing, 104
generalizing, 150
AuthenticityProblem unit, 103
auxiliary variables, 88

Back, R.-J.R., 24
backtracking, 18
benevolent side effects, 102, 103, 117
examples, 102, 117
BenSideEffects , 117
BenSideSet , 118
reducing size of, 118
binary semaphores, 48
binding powers, 4
bits, 84
blatant hand waving,
block statement (⟦•⟧), 17, 112
body
of block statement, 17
using guard statement in, 19
of iterative statement, 22
of procedure, 56
bool , 68
boolean scalar, 6, 120
bound function, 22
box operator (▯), 20, 112
buff data field, 98–102
comment describing usage of, 99

C++ [23], 8, 12, 74, 78, 81, 82, 91
references in, 74
terminology, 78, 82–83
calculations
format of, *see* proof format
call , 58, 112
Caltech, v
cand , 62
ceiling (⌈ ⌉)
of functions, 30
of programs, 34
of states, 38
Chandy, K. Mani, v
chapter dependencies, 3
choice compositions, 20
choice operators, 20
Civitype, v
classical data refinement, 2, 3, 86
and Simula, 108
generalization of, 87, 106
shortcomings of, 88
client, 8, 96, 100, 102, 105, 108, 109, 147, 149, 150
CLU [54], 51, 109
code reuse, 56
collaboration, 4
collections, 74
command, 8
compositions
duality between, 33
exceptional, 17, 42
normal, 16, 39
conditional operators, *see* short-circuit operators
conjunction ($\wedge$), 5
conjunctivity, 6
monotonicity implied by, 6, 44, 119
of *wlp* , 119
of *wp* , 44

consume method, 144
Consume procedure, 144
consumer example, 144
  problem with, 147
Consumer unit, 144
consumers, 144
contents (of *FastSeq.T*), 147
contributions, *see* thesis contributions
control structures, 14
convention, *see* requirements, convention
correctness
  partial, 15
  total, 15
cyclic index entries, *see* index entries, cyclic
C [45], 8, 12, 56, 59, 68–70
  initialization of variables in, 70
  references in, 74
  types in, 70

data abstraction, 66, 84
  importance of, 88
  in modular programs, 96
data fields, 78
  abstract, 89
  in record types, 72
  of objects, 79
  on notation of, 81
  protection levels, 83
data hiding, 94, 96, 105, 147
data refinement, 66, 84, 86
  and partial commands, 88
  classical, *see* classical data refinement
  in modular programs, 66
  in subtypes, 80
data structures, 66
data types, *see also* types
  declaring new, 73
  in common programming languages, 70
  meaning of, 80
  relation between, 80
deallocation, 74
DEC SRC, *see* Digital
declaring type, 79
demonic choice, *see* box operator
Denver International Airport, 0
dependencies, 101
  cyclic, 142
  inferring, 108
  pointwise, 147
dependency graph, 106
dependency relation, 113
Depends, 113
depends, 3, 101, 112
  formalization of, 110
  placement of, 102
  restrictions on usage of, 102
    enforcing, 104
  shortcomings of, 146
  successes of, 144
dereference (^), 73, 78
dereference map, 74
Detlefs, David, v
Digital
  SRC Extended Static Checker, v
  Systems Research Center, v, 4
Dijkstra, Edsger W., iii, 1, 12, 24, 35, 43
disjunction ($\vee$), 5
disjunctivity, 6
distributivity, 6
double composition ($\langle\circ\rangle$), 28
downward closure, 101, 113
dynamic type, 78

element type, 68
`ELSE`, 59
else operator ($\boxtimes$), 20
`ELSIF`, 59
ensures, 57
entry variables, *see* initial-value variables
enumerations, 66, 71
Env, 119
environment, 112

erroneous (Ada lingo), 63
errors
  in software, *see* software errors
Euclid [47], 74
everywhere operator ([ ]), 5
  for different state spaces, 124
except-ensures , 57
exception
  definition, 2
exception handler, *see* exceptional composition
exceptional composition ($\lhd$), 17
exceptional programs
  definition, 3
exceptional projection, *see* ceiling
exceptional semantics
  previous work, 35
exceptions, 12
  in program construction, 52, 54
  more than one, 35, 38, 60
  theorem regarding usage of, 51
Excluded Middle, Law of, 18
explicit functions, 107
expressions, 61–63
  and compilers, 62
  evaluation of, 16, 61–63
extended static checking, v, 1, 2, 145, 154
  vs. full verification, 1
  idea of, 1

$f\star$ functions, 114
false , 6
  outcome component implicitly being, 60, 119
FastSeq unit, 147
FaultyClient unit, 100
feasible statements, *see* partial commands
Feijen, Wim H.J., 7
fields, *see* data fields
file writers, *see* writers, file
fin , 38
fixed points, 21
floor ($\lfloor\ \rfloor$)
  of functions, 30
  of programs, 34
  of states, 38
flush method, 99
flushed data field, 100, 146
forest requirement, 106
frame, *see* specifications, frame of
`FREE`, 75
friends interfaces, 97, 98, 147
function application, 4
function compositions, 26
  programs as, 33
functionalization, 114
Functionalize , 114
functions, 4
  nondeterministic, 85
  of two arguments, 17, 26
  uninterpreted, 126

GetSpec , 117
global in , 63
global variables, *see* variables, global
van der Goot, Marcel R., v
guard function, 19
guard of a command, 19
guard statement ($\rightarrow$), 18, 112
guarded commands, 8, 12, 14

Harley, Robert J., v
heuristic for program construction, 52
Heydon, Allan, v
Hoare triples, 1, 48
Hoare, C.A.R., iii, 1, 84
Hofstee, H. Peter, v

identifiers
  naming of, 111
if fi brackets, 20
`IF THEN` , 59
if then fi , 59
impl , 58, 111
import, 96
import , 111

import closure, 111
ImportClosure, 111
imported values, 149
index entries
  cyclic, *see* cyclic index entries
index set, 68
  infinite, 81
index type, 68
indices of maps, 68
inf, 38
ingrown toenails?, *see* physician
inheritance, 79
initial-value variables, 22, 23, 57, 75, 111, 121
input stream, 144
input/output streams, 96, 98, 144
instantiation, 136
int, 68
interfaces, 89, 96, 97, 110
intermediate code, 62
Invariance Theorem, 21, 48
iteration, 21
iterative statement (**do** $\mapsto$ **od**), 21
  absence of, 112

Jackson, Daniel, 145
joke (getting to know my readers), vi
junctivity, 6
Juno-2 [35], 20

Knaster-Tarski Theorem, 21

L projection, 17, 27
Larch [33], 57
last, 38
LaTeX, v
lattices, 12, 24
leaked values, 149
left composition ($\langle\circ$), 17, 27
Leibniz's Rule, 116, 126, 131, 134
Leino, India Jane, vi
libraries, 94
  of axioms, 109
lifting, 5, 28
LIM [10], 18
logical variables, *see* initial-value variables
Lookup, 117
loop, *see* iterative statement
  infinite, 22
Lukkien, Johan J., 37

Manasse, Mark S., 36
Manohar, Rajit, v
map (in functional programming), 144
map types, 68
map$\star$ maps, 74, 78
maps, 68
  axioms for, 69
  data fields, *see* data fields
  implementation of, 69, 81
  in specifications, 75
  totality of, 68
  updating, 68
Martin, Alain J., v
Massingill, Berna L., v
mathematical variables, *see* initial-value variables
method, 79, 80
  impl, 80
  spec, 79
methods, 78, 79
  C++ lingo for, 83
  declaration of, 79
  implementations, 79
  invocation of, 79
  protection levels, 83
  virtual (C++ lingo), 83
Microsoft, v, 2
  personal experiences at, 2
miracle
  invoking, 18
miraculous statements, *see* partial commands
modifies, 57
  abstract variables listed by, 101
  and maps, 75
  in modular specifications, 97

Modula-2 [81], 8, 12, 89, 96, 108
Modula-3 [71], 8, 12, 34, 51, 56, 59, 60, 68–74, 80–82, 89, 91, 96, 98, 99, 101, 105, 108, 110, 146
  object types in, 81–82
  partially opaque types in, 101
  references in, 74
  structural equivalence in, 82
modular programs
  and *SwitchToAbstract*, 87
  definition, 2
  precise interpretation of, 3
modular specifications
  problem of writing, 97
modular verification, 2, 3, 94, 97
  soundness of, 3
  adequacy of, 105
  completeness of, 105
  example of inadequacy of, 106
  problem in, 94
  soundness of, 94, 105, 128
modularity, 94
module invariant, 103
modules, 89, 96, 97, 110
monotonicity, 6, 15, 21, 25, 44
Moye, S.G., v

name qualifier (.), 98, 111
narrowing, 70, 71
  in abstract interpretation, 22
Nelson, Greg, v, 4, 36
new, 74, 150, 151
nil, 73, 78
nil-dereferencing, *see* run-time errors
nondeterminism, 20, 24
nondeterministic functions, 85
normal composition, *see* sequential composition
normal projection, *see* floor
notation
  explanation of, 4
  how it inspires us, 31
  on the choice of, 35

object invariant, 103, 145
object simplicity, 80
object types, 73, 78
object-oriented programming
  essence of, 99
objects, 66
  attributes of, 79
  creating new, 74
  implementation of, 81
  properties of, 79
  simplicity in modeling, 66
oc, *see* outcome coordinate
operators
  binding powers, *see* binding powers
originality, 4
outcome, 15, 38, 60
  erroneous, 60
  exceptional, 15
outcome coordinate (*oc*), 37
outline
  of Part I, 12
  of Part II, 66
  of Part III, 94
  of thesis, 3
output stream, 98

packages (Ada lingo), 89
parameters
  evaluation of, 63
  narrowing of, 70
  of methods, 79
  of procedures, 57
  special, 79
partial commands, 18
  examples of, 19
  operational interpretation of, 18
partial correctness, *see* correctness, partial
partial representations, 89
  reason for restriction on, 90
  restriction on, 90
partially opaque types, 80, 101
partitioned predicates, 35, 43, 112
Pascal [42], 8, 12

early version of [79], 74
Penelope [62], 57, 63, 109
pointers, *see* references
pointwise dependencies, 147
postcondition
in specifications, 22
specify-
ing for procedure, *see* **ensures** *or* **except-ensures**
POSTSCRIPT, v
precondition
in specifications, 22
specifying for procedure, *see* **requires**
weakest, *see* weakest preconditions
weakest liberal, *see* weakest liberal preconditions
predicate transformers, 5, 12
predicates, 5, 12
interpretation of abstract variables in, 101
over two states, *see* two-state predicates
prime factor example, 7
private, 149, 150
`private`, 83
private values, 148, 151
privatizable, 151
procedures, 56, 111
calls to, 58
implementation of, 58
parameters, 57
evaluation of, 63
narrowing of, 70
result values, 57
narrowing of, 70
signature of, 56
specification of, 56
tail-recursive, 112
termination of, 59
program constructs
implementation of, 81
implementing, 24, 36
program derivation, 12, 52
program state space, 16, 37
program statement, 8
program variables, 84, 111
as functions, 84
mixed with abstract variables, 88
programming methodology, 12, 81, 91, 148
programming notation, 14, 56, 110
for programs with units, 110
programming tools, 154
projections
L and R, 27
on states, 38
proof format, 7
proof hints, 7
vague, 123
`protected`, 83
protection levels, 83
`public`, 83
PutChar procedure, 98

quantifications, 5
dummy of, 5
existential ($\exists$), 5
notation for, 5
range of dummy in, 5
range omitted in, 5, 7
set constructor, *see* set constructor
term of, 5
union ($\cup$), 5
universal ($\forall$), 5

R projection, 17, 27
raise, 16, 39
free occurrences of, 49
RawTarget, 119
readers, 144
receiver, 79
record types, 66, 72
`REF`, 73
reference types, 73, 78
references, 66, 73
creating new, 74

crux with modeling, 74
dereferencing, 73
Refine, 110, 112, 120
monotonicity of, 129
refinement ($\sqsubseteq$), 15, 24, 58, 94, 120
and abstract variables, 86
data, *see* data refinement
proving, 122
rep, 85, 112
RepAxiom, 117
RepAxioms, 117
RepPreds, 117
representation function, *see* abstraction function
requirements, 102
convention, 104, 105, 142
necessity of, 105
deviating from, 150
enforcing, 104
explaining to programmers, 105
generalizing, 150
too strict, 146
requires, 57
reserved character, *see* $\star$
residues, 102, 110, 124
importance of, 124
Resolve, 113
shorthand for equalities, 113
resolve set, *see* downward closure
Rifkin, Adam F, v
right composition ($\circ\!\rangle$), 17, 27
run-time checks, 62
run-time errors, 60
checked, 1, 62
proving absence of, 1
checks for, 62
unchecked, 63

S.kip procedure, 151
Sanders, Beverly A., v
saving variables
observation regarding, 23
scope, 98
self, 79
semantics, 1, 3, 12
concreteness of, 36, 47
correctness of, 36
influence of, 1
operational, 3, 37
separate compilation, 94
seq[**char**], 98, 146
sequential composition (;), 16, 112
set constructor, 6
set difference ($\setminus$), 6
set types, 73
sets
as data types, 66
explanation of notation, 6
short-circuit operators, 61
side effects
benevolent, *see* benevolent side effects
in expressions, 16
signature of procedures, 56
simulation of types, 80
Simula [14], 108, 109
Sivilotti, Paolo A.G., v, 124
skip, 16, 39, 112
van de Snepscheut, Jan L.A., 1953–1994, v, 4
software
today, iii, 0
software errors, 0
possible causes of, 0
soundness
violation of, 100, 103, 105, 150, 151
soundness of modular verification, *see* modular verification
soundness proof
in retrospect, 141
soundness theorem, *see* modular verification, soundness of
spec, 56, 111
spec var, 84, 111
specification languages
related work, 108
specification statement, 22, 57, 79

being partial, 23, 24
implementation of, 24
operational interpretation of, 22
specification variables, *see* abstract variables
specifications, 1, 97
frame of, 22, 97, 111
full, 1
lack of understanding of writing, 3
modular, 96
basic problem with, 100
solution that problem, 101
of a consumer, 144
of modular programs, 3
of procedures, 56
of streams library, 96
postcondition in, 22, 111
precondition in, 22, 111
SpecVarProjection, 118
state, *see* program state space
state, 102, 145
state space, 66
statement, 8
statement compositions, 16
asymmetry between, 34
statements
that "go wrong", 60
states
normal and exceptional, 38
streams library, 96
strict commands, 18
subrange types, 66, 71
subscripting a variable by 0, *see* initial-value variables
subst, 69
substitution ([:=]), 4
X, *see* X
substitution theorem, 136
substitutions
duplicate, 115
subtype relation (<:), 78, 80, 99
subtypes, 78
proper, 78
purpose of, 99
subtyping, 66, 78, 80
considered independently of abstraction, 80
methodology, 81, 91
notion of, 66
summary, 154
supertype, 78
immediate, 78, 80
SwitchToAbstract, 86, 107
and modular programs, 87

tail recursion, 112
target, 98
buffered portion of, 99
flushed portion of, 99
representation of, 100
Target (function), 119
Target (procedure), 99
target set, 119
temperature unit conversion example, 85
termination, 14, 15, 21, 22, 59, 60
normal, 14
proving, 15, 22
testing, 0
`TEXT`, 98, 146
text writers, *see* writers, text
TextWr, 99
TextWr.T, 99
TextWrImpl, 99, 146
then, 59
theorem provers (automatic), 1
theorem proving
automatic, 1
theory packages, 109
Therac-25, 0
thesis contributions, 2, 4, 154
this, 79
Thornley, John, v
three-address code, 62
tools, 154
total commands, 18
total correctness, *see* correctness, total

trace, 38
trace semantics, 36
trace sets, 37
  programs as, 38
transitive reduction, 106
transposition ($\sim$), 32
  implementation of, 34
true , 6
  as postcondition, 145
  range, omitted, 7
`TRY FINALLY END`, 34
two-state predicates, 23, 57, 124
type
  declaring, *see* declaring type
type , 73, 78
type checking, 2, 154
type constraints, 145
types, 68
  allocated, *see* dynamic type
  composite, 68
  data, *see* data types
  dynamic, *see* dynamic type
  element, *see* element type
  enumeration, *see* enumerations
  index, *see* index type
  map, *see* map types
  nonempty, 68
  of variables, 68, 69
  record, *see* record types
  set, *see* set types
  structural equivalence among, 82
  subrange, *see* subrange types
  subtyping, *see* subtyping

unit , 111
unit statements, 16, 39, 60, 112
  raise , *see* *raise*
  skip , *see* *skip*
  wrong , *see* *wrong*
units, 97
  convention of naming, 99

valid paradigm, 103, 145
  implicit *vs.* explicit, 103, 145
var , 69, 111
variables, 16, 111
  abstract, *see* abstract variables
  authentic, 103
  auxiliary, 88
  concrete, *see* program variables
  dependencies between, *see* dependencies
  functions over, 84
  global, 68, 69
    abstract, 84
  local, 17, 69, 84
    abstract, 84
    scope rules for, 111
  program, *see* program variables
  structure of, 66
  visibility of, 69
variant function, *see* bound function
verification, 0
  and refinements, 94
  challenges of, 1
  feasibility of, 2, 94, 96
  full, 1, 146
verification conditions, 110
  generating, 110
verification process, 8
verification system
  building of, 95
virtual memory, 85
visibility requirement, 102, 104, 128, 133, 142, 146
  deviating from, 150
  enforcing, 104
  generalizing, 150
visible declaration (Anna lingo), 108
voltages, 85

weakest liberal preconditions, 12, 14, 15, 60, 119
weakest preconditions, 1, 3, 12, 14, 43, 45, 60
  in trace semantics, 43, 45
widening, 22
Wilson, Richard M., v

Wirth, Niklaus, 24
wlp, *see* weakest liberal preconditions
universal conjunctivity of, 60, 119
wp, *see* weakest preconditions
positive conjunctivity of, 44
Wr, 98
Wr.T, 98
`WrClass`, 98
WrFriends, 98
writer example, 98
writers, 98
file, 98, 99
idea behind subtypes of, 99
text, 98, 99
wrong, 60, 112

X, 131, 135–141