Verification of Floating Point Programs

Jan Andrzej Duracz

Doctor Of Philosophy

Aston University

December 2010

Aston University

Verification of Floating Point Programs

Jan Andrzej Duracz

Doctor Of Philosophy, 2010

Thesis Summary

In this thesis we present an approach to automated verification of floating point programs. Existing techniques for automated generation of correctness theorems are extended to produce proof obligations for accuracy guarantees and absence of floating point exceptions. A prototype automated real number theorem prover is presented, demonstrating a novel application of function interval arithmetic in the context of subdivision-based numerical theorem proving. The prototype is tested on correctness theorems for two simple yet non-trivial programs, proving exception freedom and tight accuracy guarantees automatically. The prover demonstrates a novel application of function interval arithmetic in the context of subdivision-based numerical theorem proving. The experiments show how function intervals can be used to combat the information loss problems that limit the applicability of traditional interval arithmetic in the context of hard real number theorem proving.

Keywords: static analysis, floating point, formal software verification, automated theorem proving

Acknowledgements

I thank my supervisor Michal Konečný, for his patience in guiding me through the theory of computing, and for his availability whenever advise or guidance was needed. It kept me motivated and made the research experience most enjoyable. I thank Dr. Amin Farjudian for his collaboration and inspiring discussions.

I thank Dr. Alan Barnes, for providing accommodation and stimulating conversations. I extend special thanks to Dr. Patchara Punyamoonwongsa, she has been a great friend and provided valuable advise and motivation.

It gives me great pleasure to thank my dear parents Anna and Andrzej and brother Adam. During the course of my studies they have provided me with all I could wish for and it is thanks to them that I have had the peace of mind to focus completely on my studies.

Finally, I acknowledge the Engineering and Physical Sciences Research Council and Praxis High Integrity Systems, who funded my research. In particular, I thank Dr. Roderick Chapman, the project liaison at Praxis, for providing generous access to training and other resources at the company.

Contents

List of Figures

1 Introduction

CONTENTS

It is trivial to implement any algorithm incorrectly

While the statement above is trivial in itself it raises an interesting question: what does it mean to implement an algorithm correctly? The answer that is usually given is that one defines the *correctness* of a program relative to a *specification* stating the desired properties the program should satisfy. To reach mathematical precision, this statement of properties must be expressed in a *formal* language, allowing for a rigorous analysis of the correspondence between program and contract. The field of *formal verification* collects techniques to support and automate such analyses. A program specification may be see as a *contract* between the provider and user of the program. The type of contract depends on the application area for the program. In high integrity applications, such as control systems and embedded software in the aerospace, rail, nuclear and military industries, the consequence of failure can be catastrophic. Therefore, it becomes imperative to reach high confidence that *safety* properties, such as exception freedom, of such systems are satisfied. Systems handling sensitive data, or that control access to such data, need to satisfy *security* properties, such as isolation between parts of the system dealing with data of different sensitivity. As pointed out in the opening statement one really should always aim to verify that a given program satisfies its contract, but due to the increased costs, the deployment of formal verification techniques in the industry has generally been limited to those areas where other methods, such as code inspection and testing, are deemed insufficient. Among industrially used languages for high integrity applications, SPARK Ada [9, 10, 21, 49] has proved to be particularly successful, with over twenty years of deployment. It does however lack adequate support for expressing functional properties of floating point programs. As of late a major air traffic system has been commissioned to be written in SPARK Ada, but due to the lack of support in the language and tool set, formal verification of the numeric algorithms has not been employed.

The work presented in this thesis addresses the particular case of verifying properties related to execution of real number algorithms using finite precision arithmetic and focuses on automation of such correctness proofs. Although applicable to any classical imperative language, one of the aims of the work has been to provide concrete advice for an extension of the SPARK Ada language and supporting tool set to include verification of functional properties of floating point programs. Consequently, the approach has been to leverage

the facilities already existing in SPARK Ada for automated generation of correctness theorems, by extending the annotation language and verification condition generator, and to look at ways to extend the capacity of existing theorem proving technologies to automatically prove the resulting correctness theorems.

The introduction continues with a brief account of the history of verification, abridged and adapted from [50], followed by an overview of approaches to automated verification and concludes with a summary of the contents of the thesis.

1.1 History of verification

The idea of formally reasoning about programs seems to go back to the very origins of automated computation. Charles Babbage (1791-1871), who is commonly considered the father of the computer, wrote on the "Verification of the Formulae Placed on the [Operation] Cards" [65]. In modern times, Goldstine and von Neumann wrote the 1947 paper [41] about using "assertion boxes" to reason about the effects of "operation boxes". While they also discussed issues with numerical approximations and rounding errors, the focus was on reasoning as an aid for program development. They did however hit upon a central idea in verification, that of combining the control part of a program with assertions about the values it computes.

Turing presented a correctness proof for a program computing the factorial function [71] at the 1949 Cambridge conference on high speed automatic calculating machines. He illustrated his reasoning using a flow diagram labelled with control points where assertions about the state are made. In a 1961 presentation and the following 1963 paper [54] McCarthy called for the investigation of a "Mathematical Theory of Computation". He makes the point that in order to reason about programs one needs to make the semantics of the programming language precise. This is clear, since one cannot draw conclusions about the execution of a program from the program's code, unless the compiler preserves the semantics of the program. Wijngaarden gave a presentation in 1964 and published the paper [72] in 1966 where he outlines a formal framework for reasoning about the evaluation of expressions in inexact arithmetic. The 1966 paper by Naur [59] introduced "General Snapshots", which are assertions formulated in plain English mathematics and written into the code of a program as comments. It is similar to the approach of Turing,

updated for a higher level language. Naur still promotes a carefully formulated correctness argument in English, rather than a proof in a formal language. In contrast to Naur, the 1967 paper by Floyd [39] used propositional logic as the language for assertions and placed them as labels on the edges of the control flow graph of the program.

Floyd gave *verification conditions* ensuring that the assertions in the flow chart corresponded to the program statements. In the paper Floyd was using "strongest verifiable consequents", which are obtained from an assertion preceding a statement by deducing the strongest assertion that holds after the statement. The rule for assignment statements thus obtained becomes complicated as it uses an existential quantifier. Floyd subsequently found the "backwards rule" for assignment statements. The main difference between Turing's and Floyd's work is the expressivity of the assertion languages. Jones [50] points out that in Turing's paper [71], the assertions may only relate values between variables in terms of explicit expressions, while Floyd's allows arbitrary predicates. Floyd cites Perlis and Gorn as inspiration for his approach to correctness. Jones confirms that Gorn was using flow diagrams to reason about programs in [42] and that he gives an "induction rule" foreshadowing *loop invariants*, but that no systematic treatment of assertions was given. Crucially, Floyd was the first to present formal rules for checking verification conditions.

Hoare adopted the backwards rule in his axiomatic approach, presented in his celebrated 1969 paper [47], where he introduces *Hoare triples*, citing the papers by Floyd, Naur and van Wijngaarden mentioned above as inspiration. A Hoare triple $\{P\}S\{Q\}$ contains the *precondition* assertion P and *postcondition* assertion Q placed within braces around the program segment S, the braces representing comments à la Naur's General Snapshots. The triple has the following interpretation: if S is executed in a state satisfying P, then, *provided the execution terminates*, the state after execution will satisfy Q. Hoare gives deduction rules defined inductively over the language constructs: assignments, conditionals and while loops. Jones points out that this marks a departure from *reasoning operationally* about the program execution, by presenting a proof as a sequence of deductions using inference rules. The property proved by this method is called *partial correctness*, where *partial* refers to the termination requirement. If proof of termination is provided as well, we say that *total correctness* has been established.

The development of formal frameworks for program verification in the direction of increasing formalisation makes possible the development of algorithms that automate the translation of specified programs into correctness theorems. Dijkstra's 1975 paper [34] moves verification yet another step towards automation by introducing the *predicate transformer wp*. Dijkstra's idea is to view program constructs as parameters of functions that transform (assertion) predicates. In Hoare's approach a triple $\{P\}S_1;S_2\{Q\}$ is constructed from triples $\{P\}S_1\{R\}$ and $\{R'\}S_2\{Q\}$, with $R \Rightarrow R'$. Dijkstra in contrast asks what the *weakest* precondition is, such that $S_1;S_2$ terminates and produces a state for which Q is true. The advantages compared to Hoare's approach is that *wp* handles termination and that *wp* has an appealing calculus that is well suited for implementation on a computer. The (total) correctness theorem corresponding to a Hoare triple $\{P\}S\{Q\}$ has the succinct form

$$P \Rightarrow wp(S, Q)$$

where $wp(S, Q)$ stands for the weakest predicate that makes S terminate and produce a result satisfying Q.

Since its introduction *wp* has been generalised in various ways to include a wider class of language constructs. In [35] Dijkstra relaxed the *termination* requirement on *wp* obtaining the *weakest liberal precondition wlp*, later generalised by Lamport in [53] to the *weakest invariant* predicate transformer *win* for reasoning about *concurrency*. *Probabilistic* predicate transformers were described by Morgan, McIver and Seidel in [58] and a *quantum weakest precondition* predicate transformer was introduced by D'Hondt and Panangaden in [33].

1.2 Related work

Proving software correctness may of course be done by hand, in the way mathematicians usually solve problems. The main drawback of this approach is that it does not scale well in the sense that due to human factors, sufficiently large analyses will inevitably contain errors, some of which may invalidate the correctness proofs. Thus, even partial automation of analysis and certification is a highly attractive goal for research aimed at practical software verification. The two main approaches to proving software correctness choose either full automation for a fixed set of properties, or partial automation for a larger class

of properties. Techniques such as Model Checking and Abstract Interpretation belong the former group, while Verification by Theorem Proving belongs to the latter.

In *abstract interpretation* [23, 24, 25, 26] computation is soundly approximated, yielding abstractions of the state space over which some properties may be safely verified. The drawback is that the representations we may choose for an abstract state are limited on one side by the efficiency with with it may be computed and on the other by the property we wish to prove over it. Full automation is achieved by using coarsely grained approximations, which are computationally tractable but imprecise and can therefore not prove highly accurate properties. When precision loss becomes prohibitive, one needs to use abstract states that encode information about relations between program variables. To this end specialised *domains* have been proposed [55, 67] which attempt to strike a balance between information loss and computational complexity. Most implementations are however based on interval arithmetic and therefore suffer the usual information loss resulting from *wrapping effects* described in Section 2.2. Tools based on abstract interpretation, such as Fluctuat [43, 64], have been developed specifically with verification of floating point properties in mind. Industrial applications of abstract interpreters have been described in *e.g.* [7, 32, 38]. The most notable application to floating point verification is the certification of aeronautical software using the Astrée Static Analyzer [15, 16, 27].

Model checking algorithms solve the following problem: given a model of the system, automatically check that a given property is satisfied by the model [48]. In practice such algorithms work by building a representation of the state space of the model and then traversing the space, checking that the property is satisfied in each particular state. This approach is limited by the size of the state space, and is generally applied only to Boolean and integer problems [3]. Model checking has had some success in software verification, notably static analysis of C implementations of device drivers using the BLAST model checker [14] and of Microsoft Windows device drivers using the SLAM model checker [8]. Attempts have been made to combat the state space problem by combining model checking and theorem proving [22], but to date no success at verification of floating point programs has been reported.

The *theorem proving* approach splits the verification task into two steps. First, a correctness theorem is generated for the contract-program pair and then a theorem prover is used to prove the generated theorem. Automation, which is achieved through approximation,

becomes nearly impossible when wishing to prove very precise properties of a program. In such cases at least the generation of correctness theorems from formally specified programs may be automated. The proof effort consists of modelling the computation in an interactive proving environment and then manually guiding the proof construction. Often some of this work can be performed during the generation of the correctness theorem, *e.g.* most verification condition generators eliminate control flow from the generated theorem. We show in Chapters 3 and 5 that it is possible to eliminate all references to operational semantics in the generation step, resulting in correctness theorems for floating point programs containing only the usual real number functions and relations. Verification of functional properties of floating point C programs has been demonstrated using the Caduceus verification condition generator and the Coq interactive theorem prover [17, 18].

The main contribution of [18] was the introduction of so-called *model values*, which are ghost variables serving as documentation of the *intended* real number values a program computes. Thus, it becomes possible to specify both *method error*, *i. e.* the error introduced by rounding operations, as well as *model error*, *i. e.* the total deviation of the actual values returned by the program from the values sought by the programmer. The drawback with this approach is that the model values are given by *equalities* of *exact real number* expressions, often involving transcendental functions, which famously are not decidable. Since our aim is to automate correctness theorem proofs we have to abandon model values and instead formulate error properties of floating point programs using approximations, such as the ones presented in Chapter 5.

The advantages of the theorem proving approach compared to proving by hand are that a correctly implemented system will not allow erroneous deductions and that the generated proofs may be mechanically checked by external systems. A drawback is that the typical system will require a high level of familiarity to be used efficiently and even then the proof efforts may take months even for rather simple theorems. The solution is to employ an automated theorem prover in the second step, as is demonstrated in [19] where the automated prover Gappa [19, 30, 31] is used to discharge verification conditions generated by Caduceus. Unlike most automated numerical theorem provers, Gappa also generates a formal proof that may be checked by an independent prover, thus giving even greater confidence in the validity of the proof. As is the case for the vast majority of automated numerical theorem provers, Gappa uses interval arithmetic to obtain bounds on numeric

expressions. The pathological information loss of interval arithmetic, described in Section 2.2, limits the range of properties that may be automatically verified this way. In Chapters 7 and 8 we address the problem of proof automation for numerical programs and demonstrate how using an arithmetic that is less prone to information loss can improve the performance of subdivision-based decision procedures.

1.3 Contents of Thesis

The work presented in this thesis is divided into two parts. Each begins with introductions of definitions and concepts and a recount of related work. The first part comprises Chapters 2 to 6 and the second Chapters 7 and 8.

Part One

The first part of the thesis addresses the problem of automated generation of proof obligations for functional properties of floating point code.

In Chapter 2 mathematical definitions needed in the following chapters are given and a review of relevant work is presented. Chapter 3 introduces the model programming language PROC and its extension PEA by contracts and invariants. PROC is obtained from the extension of WHILE with procedures as defined in Section 1.2 and 2.5 of [13] by disallowing global variables, allowing multiple procedure in and out parameters and introducing implicit declaration of functions from procedure declarations with a single out parameter. Operational and denotational semantics for PROC and PEA are then introduced along with corresponding notions of correctness. The operational semantics take a particularly simple form, when compared with the operational sematics given for the extended WHILE in Section 2.21 and 2.5.1 in [13], due to the lack of global variables in PROC. A theorem is stated and proved, showing that denotational correctness is stronger than operational correctness, in the sense that it is sufficient to prove the denotationally derived correctness theorem in order to prove operationally defined correctness. The main reason for the introduction of the model languages is to allow for the statement and proof of this theorem and to set the discussion of program specification on a firm footing. The chapter concludes with a discussion of referential transparency, where the notion is lifted from the usual setting of expression languages to statement languages, making precise the

intuition that PROC is in essence a pure language.

Chapter 4 is a short account of the IEEE floating point standard and the parts of the Ada language standard relating to the implementation of floating point functions. In Chapter 5 suitable specifications for the built-in operations and the library functions described in Chapter 4 are considered. Then, the problem of specifying arbitrary floating point programs is addressed, resulting in an extension of the PEA annotation language with primitives making the formulation of certain contracts more succinct and introducing the integral operator to the expression sublanguage.

Chapter 6 concludes part one with complete implementations in SPARK Ada of the example PEA programs given in the preceding chapter. The implementation serves as a recommendation for the extension of the facilities in SPARK Ada for verification of floating point programs.

Part Two

The second part of the thesis addresses the problem of proof automation for the correctness theorems generated by the methods described in part one.

Chapter 7 defines the notion of approximation and generalisations of interval arithmetic used in the subsequent discussion and implemented by the prototype theorem prover presented in the following chapter.

Chapter 8 describes a prototype theorem prover, based on the classical subdivision search algorithm used in most numerical constraint solvers. The novelty is the use of polynomial function interval arithmetic, which extends the interval arithmetic traditionally used in the field, allowing for variable precision approximation of continuous functions over the real numbers. Chapter 8 concludes by presenting experiments showing that the computational overheads of using polynomial function interval arithmetic compared to traditional interval arithmetic are outweighed by the increased approximation precision when used in a subdivision search algorithm.

Chapter 9 presents the conclusions drawn from the investigation, questions left unanswered and potential directions that future investigations may take.

1.4 Summary of contributions

The problem this work addresses is that of automation of correctness proofs for imperative floating point programs in general and SPARK Ada floating point programs in particular. The approach taken is by automated generation of correctness theorems followed by automated theorem proving. The main contributions are the following:

Defining referential transparency for statement languages, which we take to mean that each variable a program interacts with is passed explicitly as a parameter. In SPARK Ada this is achieved by demanding that all variables that are not formal parameters of the program, but that are written or read by the program, are declared in the program's specification. Using this property we formulate a simplified form of *operational semantics* for a model language based on the SPARK Ada core language in Chapter 3. A *denotational semantics* based on Dijkstra's *weakest precondition* predicate transformer semantics is also presented in Chapter 3 formalising the generation of correctness theorems for the model language and a theorem relating the denotational and operational semantics is given. Previously, a predicate transformer semantics has been given in [46] for a sublanguage of SPARK Ada with side-effecting functions. Due to our approach disallowing global state the exposition is simplified by allowing the omission of a top level environment. Referential transparency for procedural languages is formalised by treating statements as generalised expressions and lifting the standard definition, as given in [70], accordingly.

Automated generation of floating point proof obligations is achieved by extending the existing verification condition generation facilities to programs with floating point expressions by equipping floating point operations and functions with specifications for functional properties. Alternative ways to specify these properties are discussed in Chapter 5 and SPARK Ada implementations of these ideas are provided in Chapter 6.

Introduction of the integration operator to the specification language, making the expression sublanguage more expressive. Described in Section 5.4.1, the extension allows users to relate values computed by a program with functions defined in terms of integrals, giving an alternative to adding special functions to the specification language.

Introduction of interval expressions to the specification language, described in Section 5.4.3. The extension allows users to express program properties in terms of intervals, making the resulting contracts concise and composable. To our knowledge these exten-

sions are novel in the context of specification languages for floating point programs.

Novel approach to information loss in subdivision search algorithms. So called *wrapping effects* and the *dependency problem* of interval arithmetic limit the size and complexity of numerical theorems that these standard techniques can tackle. We propose for the first time to use *function interval arithmetic* to reduce such information loss effects in numerical theorem provers and describe the implementation of a prototype prover in Chapter 8. An informal analysis of the information loss reduction is presented for two small but nontrivial programs but future rigorous analyses for various classes of programs will be needed before definite conclusions may be drawn.

Automation of some exception freedom and hard functional property proofs, described in Section 8.3. The hypothesis that function interval arithmetic may improve the overall performance of subdivision search algorithms is tested by using the prover to mechanise the proofs of exception freedom and tight accuracy guarantees for two nontrivial programs, including one with a while loop.

2 Background

CONTENTS

In the following chapter we define concepts necessary for specifying approximate properties of FP programs. In Chapter 5 we will use real interval arithmetic to express bounds on results of floating point computations. The definitions will be given in some generality, as we hope to show that the various levels of approximation presented throughout the work follow a natural pattern. We begin by defining partially ordered sets in Section 2.1 and then intervals and interval arithmetic in Section 2.2. In Section 2.3 we outline issues associated with approximation of elements of a poset with elements of its poset of order intervals.

2.1 Posets

Definition 2.1.1 (Partial order)**.** *A partial order ω on a set R is a relation on R, i.e. a subset of $R \times R$, for which the following properties hold:*

- *if (x, y) and (y, z) are in ω, then (x, z) is in ω (transitive)*
- *(x, x) is in ω for all x in R (reflexive)*
- *if (x, y) and (y, x) are in ω, then $x = y$ (antisymmetric)*

We will usually write relations in-fix, so that the statement $x \,\omega\, y$ will mean $(x, y) \in \omega$. Each partial order ω has a *strict* version defined by: $(x, y) \in \omega \wedge x \neq y$, we will usually denote partial orders by $\leq$ or $\sqsubseteq$, and their strict versions by $<$ and $\sqsubset$, respectively. When $x \leq y$, we say that x is *lesser than* or *below* y, or dually, that y is *greater than* or *above* x. In the case when $x < y$ we say that x is *strictly lesser than* or *strictly below* y, or dually, that y is *strictly greater than* or *strictly above* x.

Definition 2.1.2 (Opposite relation)**.** *Each relation $\rho \subseteq X \times X$ on a set X has an* opposite *version $\overline{\rho} \subseteq X \times X$ defined by $(x, y) \in \overline{\rho} \iff (y, x) \in \rho$.*

Example 2.1.3 (Opposite partial order)**.** *Each partial order ω has an opposite version $\overline{\omega}$ given by the corresponding opposite relation, i.e. $x \,\overline{\omega}\, y \iff y \,\omega\, x$. The opposites of $\leq$ and $\sqsubseteq$ are denoted by $\geq$ and $\sqsupseteq$, respectively.*

Definition 2.1.4 (Poset)**.** *A pair $(R, \leq)$, where R is a set and $\leq$ is a partial order on R, is called a partially ordered set, or poset.*

Whenever $(R, \leq)$ is a poset, we will also refer to the set R as partially ordered. Each subset $S \subseteq R$ of a poset defines a poset ordered by the order on R, restricted to S. We call such posets *sub-posets* of R.

Example 2.1.5 (Powerset and opposite powerset poset)**.** *The poset* $(\wp(X), \subseteq)$ *is given by the powerset* $\wp(X)$ *of a set* X *ordered by set inclusion. Its opposite poset* $(\wp(X), \supseteq)$ *is ordered by reverse set inclusion.*

Example 2.1.6 (Poset-valued function poset)**.** *The poset* $(X \to R, \dot{\leq})$ *is given by the set* $X \to R$ *of functions from a set* X *to a poset* $(R, \leq)$*, ordered by the* pointwise $\leq$*-order* $\dot{\leq}$*, defined by*

$$f \dot{\leq} g \equiv \forall x \in X \,.\, f(x) \leq g(x)$$

for all $f, g \in X \to R$*.*

Definition 2.1.7 (Poset morphism)**.** *Let* $(R, \leq)$ *and* $(S, \sqsubseteq)$ *be posets. A function* $f : R \to S$ *is called a* poset morphism *if it is order preserving, i. e. if*

$$x \leq y \Rightarrow f(x) \sqsubseteq f(y)$$

for all $x, y \in R$*.*

It follows that the image $f(R)$ of a poset morphism $f : R \to S$ is a sub-poset of S. Functions of many arguments that are a poset morphism in each argument are called *isotonic*:

Definition 2.1.8 (Isotonic function)**.** *Let* $(R, \leq)$ *and* $(S \sqsubseteq)$ *be posets and* $f : R^n \to S$ *a function. We call* f isotonic *if* $x_i \leq x_i'$ *implies* $f(x_1, \ldots, x_i, \ldots, x_n) \sqsubseteq f(x_1, \ldots, x_i', \ldots, x_n)$ *for each* $i \in \{1, \ldots, n\}$*.*

Definition 2.1.9 (Maximal and Minimal)**.** *Let* $(R, \leq)$ *be a poset and* $S \subseteq R$*. We call* $x \in S$

- *a maximal element of* S *whenever there is no* $y \in R$ *strictly greater than* x
- *a minimal element of* S *whenever there is no* $y \in R$ *strictly lesser than* x

Example 2.1.10 (Maximal subsets)**.** *In the sub-poset* $(\wp(X) \setminus \{\varnothing\}, \supseteq)$ *of the opposite powerset poset from Example 2.1.5 the maximal elements are the* singleton sets $\{x\}$*, for each* $x \in X$*.*

Example 2.1.11 (Minimal functions)**.** *Let* $(X \to R, \dot{\leq})$ *be the pointwise ordered poset valued functions from Example 2.1.6 and* $R_{\min}$ *the set of minimal elements of R. Then the set* $X \to R_{\min}$ *of* $R_{\min}$*-valued functions is the set of minimal elements of* $X \to R$*.*

Definition 2.1.12 (Upper and lower bound, greatest and least element)**.** *Let* $(R, \leq)$ *be a poset,* $S \subseteq R$ *and* $x \in R$*. We call x*

- *an upper bound of S whenever x is greater than any* $y \in S$*, and say that S is bounded above by x. If x in addition lies in S we call x a greatest element of S*
- *a lower bound of S whenever x is lesser than any* $y \in S$*, and say that S is bounded below by x. If x in addition lies in S we call x a least element of S*

Note that greatest and least elements need not exist, but when they do they are unique by antisymmetry.

Example 2.1.13 (Two-point extended reals)**.** *The poset* $(\mathbb{R}, \leq)$ *of real numbers with the usual order has no maximal or minimal elements. Extending* $\mathbb{R}$ *and* $\leq$ *by the points* $-\infty$ *and* $+\infty$ *in the obvious way we get the poset* $(\mathbb{R}^{\pm\infty}, \leq)$ *of two-point extended reals, with* $-\infty$ *as least element and* $+\infty$ *as greatest element.*

Example 2.1.14 (Greatest and least subset)**.** *In the sub-poset* $(\wp(X) \setminus \{\varnothing\}, \supseteq)$ *of the opposite powerset poset from Example 2.1.5,* the set *X is the least element and if X has two or more elements, then there is no greatest element.*

Definition 2.1.15 (Supremum and infimum)**.** *Let* $(R, \leq)$ *be a poset and* $S \subseteq R$ *be a bounded subset of R*

- *if S is bounded above and the set of upper bounds has a least element, then we call this element the least upper bound, or the supremum, of S and denote it by* $\sup S$
- *if S is bounded below and the set of lower bounds has a greatest element, then we call this element the greatest lower bound, or infimum, of S and denote it by* $\inf S$

Example 2.1.16 (Supremum and infimum in $\wp(X)$)**.** *In the powerset poset* $(\wp(X), \subseteq)$ *from Example 2.1.5, the supremum is given by* $\cup$ *and infimum by* $\cap$*. Dually, we may order* $\wp(X)$ *by the superset relation* $\supseteq$*, in which case the roles of* $\cup$ *and* $\cap$ *are reversed.*

Definition 2.1.17 (Lattice)**.** *A poset L is called a* join-semilattice *if each finite subset has a supremum in L and called a* meet-semilattice *if each finite subset has an infimum in L. L is called a* lattice *if L is both a join and a meet semilattice.*

In the language of lattices, sup and inf are usually called *join* and *meet* and denoted by $\sqcup$ and $\sqcap$, respectively. We shall often write meet and join in-fix for two element sets, *i. e.* $x \sqcup y$ for $\sqcup\{x, y\}$ and $x \sqcap y$ for $\sqcap\{x, y\}$. When viewing a poset L as a lattice we often make the lattice operation explicit, writing $(L, \sqcup)$, $(L, \sqcap)$ and $(L, \sqcup, \sqcap)$ for a meet-semilattice, join-semilattice and lattice, respectively.

Example 2.1.18 (Real number lattice)**.** *The set of real numbers $\mathbb{R}$ with* max *and* min *as join and meet form the lattice* $(\mathbb{R}, \max, \min)$ *associated to the usual partial order $\leq$ on $\mathbb{R}$.*

Example 2.1.19 (Powerset lattices)**.** *The powerset poset* $(\wp(X), \subseteq)$ *and opposite powerset poset* $(\wp(X), \supseteq)$ *correspond to the powerset lattice* $(\wp(X), \cup, \cap)$ *and opposite powerset lattice* $(\wp(X), \cap, \cup)$.

Example 2.1.20 (Lattice valued function lattice)**.** $(X \to L, \dot{\sqcup}, \dot{\sqcap})$*, the set $X \to L$ of functions from a set X to a lattice* $(L, \dot{\sqcup}, \dot{\sqcap})$*, with* pointwise *join* $\dot{\sqcup}$ *and meet* $\dot{\sqcap}$*, defined by*

$$f \dot{\sqcup} g = \lambda x \,.\, f(x) \sqcup g(x) \quad \text{and} \quad f \dot{\sqcap} g = \lambda x \,.\, f(x) \sqcap g(x)$$

for all $f, g \in X \to L$.

Definition 2.1.21 (Complete lattice)**.** *A join-semilattice in which any subset has a supremum and a meet-semilattice in which any subset has an infimum are called* complete*. A lattice that is complete as join- and meet-semilattice is called complete.*

In particular, complete join/meet-semilattices have a greatest/least or *top*/*bottom* element, denoted $\top$ and $\bot$ respectively. It is customary to make the top and bottom element explicit in complete lattices, writing $(L, \sqcup, \sqcap, \top, \bot)$. When a join- or meet-semilattice L is not complete one can always extend L to a complete semilattice $L^\top$ or $L_\bot$ by adding a top or bottom element to L and extending the lattice operation in the obvious way. Any lattice $(L, \sqcup, \sqcap)$ can therefore be extended to a complete lattice $(L^\top_\bot, \sqcup, \sqcap, \top, \bot)$.

Example 2.1.22 (Two-point extended reals lattice)**.** *The two-point extended reals $\mathbb{R}^{\pm\infty}$ from Example 2.1.13 extend the real number lattice from Example 2.1.18 to the complete lattice* $(\mathbb{R}, \sup, \inf, +\infty, -\infty)$.

Example 2.1.23 (The complete powerset lattices)**.** *The powerset lattice* $(\wp(X), \cup, \cap, X, \varnothing)$ *is complete, as is the opposite powerset lattice* $(\wp(X), \cap, \cup, \varnothing, X)$.

2.2 Interval arithmetic

In the previous section various order structures were defined. In the present section we shall focus on the meet-semilattice associated with a particular sub-poset of the poset $(\wp(\mathbb{R}), \supseteq)$ and the interplay of the order structure with the arithmetic structure in $\mathbb{R}$.

Interval Arithmetic (IA) was first published in its current form by M. Warmus in the 1956 paper "Calculus of Approximations" [73]. R. E. Moore re-discovered and popularised [57] IA by demonstrating the first nontrivial applications. Below we present the basic definitions in some generality.

Definition 2.2.1 (Order interval)**.** *Let $(R, \leq)$ be a poset and $a, b \in R$ such that $a \leq b$, then the R-interval given by a and b is the set*

$$\{x \in R \mid a \leq x \wedge x \leq b\}$$

We write the interval given by $a \leq b$ as $[a, b]$. The requirement that $a \leq b$ guarantees that an interval is nonempty.

Definition 2.2.2 (R-intervals)**.** *Let $(R, \leq)$ be a poset. The set $\mathbb{I}(R)$ of R-intervals is defined by*

$$\mathbb{I}(R) = \{[a, b] \mid a, b \in R \wedge a \leq b\}$$

We shall use boldface letters for interval variables: $\boldsymbol{r} \in \mathbb{I}(R)$ and underline the lower and overline the upper bounds: $\boldsymbol{r} = [\underline{r}, \overline{r}]$. We define a partial order $\sqsubseteq$ on $\mathbb{I}(R)$ in terms of the order $\leq$ on R:

Example 2.2.3 (Real intervals)**.** *The set of compact and connected subsets of $\mathbb{R}$ coincides with the set $\mathbb{I}(\mathbb{R})$ of $\mathbb{R}$-intervals.*

Relations on posets may be lifted to the associated interval poset in a canonical fashion:

Definition 2.2.4 (Induced relation)**.** *Let $(R, \leq)$ be a poset and ρ a relation on R. The relation $\rho_{\mathbb{I}}$* induced *by ρ on $\mathbb{I}(R)$ is defined by*

$$\boldsymbol{r}_1 \, \rho_{\mathbb{I}} \, \boldsymbol{r}_2 \equiv \forall r_1 \in \boldsymbol{r}_1, r_2 \in \boldsymbol{r}_2 \,.\, r_1 \, \rho \, r_2 \tag{2.1}$$

for $\boldsymbol{r}_1, \boldsymbol{r}_2 \in \mathbb{I}(R)$.

Example 2.2.5 (Induced relation)**.** *Let* $(R, \leq)$ *be a poset. By Definition 2.2.4,* $\leq$ *induces a relation* $\leq_{\mathbb{I}}$ *on* $\mathbb{I}(R)$ *satisfying (2.1). In this case we can express the relation in terms of the endpoints of its interval arguments:*

$$[a, b] \leq_{\mathbb{I}} [c, d] \equiv b \leq c$$

for $[a, b], [c, d] \in \mathbb{I}(R)$*. Note that* $\leq_{\mathbb{I}}$ *is no longer reflexive and therefore not a partial order on* $\mathbb{I}$*.*

Definition 2.2.6 (Refinement order)**.** *Let* $(R, \leq)$ *be a poset, and* $[a, b], [c, d] \in \mathbb{I}(R)$*. The* refinement partial order $\sqsubseteq$ *on* $\mathbb{I}(R)$ *is defined by*

$$[a, b] \sqsubseteq [c, d] \equiv a \leq c \wedge d \leq b$$

and we say that $[a, b]$ *is* refined *by* $[c, d]$*.*

Note that $\sqsubseteq$ is just the superset relation $\supseteq$ on R. Indeed, $(\mathbb{I}(R), \sqsubseteq)$ is a sub-poset of $(\wp(R), \supseteq)$. The refinement order $\sqsubseteq$ is often referred to as the *information order*. In this context one reads $\boldsymbol{r}_1 \sqsubseteq \boldsymbol{r}_2$ as "$\boldsymbol{r}_1$ approximates $\boldsymbol{r}_2$", for $\boldsymbol{r}_1, \boldsymbol{r}_2 \in \mathbb{I}(R)$. Traditionally, approximation of elements of R within $\mathbb{I}(R)$ is studied in the complete meet-semilattice $(\mathbb{I}(R)_\perp, \sqcap, \perp)$, associated to the sub-poset $(\mathbb{I}(R), \sqsubseteq)$ of $(\wp(R), \supseteq)$. The maximal elements of $(\mathbb{I}(R), \sqsupseteq)$ are the singleton intervals $[x, x]$ with $x \in R$. When the interval $\boldsymbol{r}_2$ approximates an element $r \in R$ *i. e.* $\boldsymbol{r}_2 \sqsubseteq [r, r]$ and $\boldsymbol{r}_1 \sqsubseteq \boldsymbol{r}_2$, then $\boldsymbol{r}_1$ also approximates r but the information about r in $\boldsymbol{r}_1$ is more "diluted" than in $\boldsymbol{r}_2$. Thus, the maximal elements in $(\mathbb{I}(R), \sqsubseteq)$ provide maximal information about the elements of R and the bottom element, which approximates any element of R, provides none.

Definition 2.2.7 (Interval semilattice)**.** *Let* $(L, \sqcup, \sqcap)$ *be a lattice and* $(L, \leq)$ *the associated poset. Then* $(\mathbb{I}(L), \bar{\sqcap})$ *is the meet-semilattice associated to the refinement order on* $\mathbb{I}(L)$*, where* $\bar{\sqcap}$ *is defined by*

$$[a, b] \,\bar{\sqcap}\, [c, d] = [a \sqcap c, b \sqcup d]$$

for $[a, b], [c, d] \in \mathbb{I}(L)$*. By adding a bottom element* $\perp$ *we obtain the complete interval semilattice* $(\mathbb{I}(L)_\perp, \bar{\sqcap}, \perp)$*.*

Definition 2.2.8 (Outer interval extension, containment property)**.** *Let R be a poset. Given an operation* $\circ : R \times R \to R$ *on R we call an operation* $\bullet : \mathbb{I}(R) \times \mathbb{I}(R) \to \mathbb{I}(R)$ *on* $\mathbb{I}(R)$ *an* outer interval extension *of* $\circ$ *whenever it satsifies the* containment property*:*

$$\boldsymbol{x} \bullet \boldsymbol{y} \supseteq \{x \circ y \mid x \in \boldsymbol{x} \wedge y \in \boldsymbol{y}\}$$

for intervals $\boldsymbol{x}, \boldsymbol{y} \in \mathbb{I}(R)$.

Operations on a lattice $(L, \max, \min)$ that are monotonic in each argument can be extended by defining the extension directly on the endpoints of its arguments:

$$[a, b] \bullet [c, d] = [\min\{a \circ c, a \circ d, b \circ c, b \circ d\}, \max\{a \circ c, a \circ d, b \circ c, b \circ d\}] \tag{2.2}$$

Equation (2.2) gives a general formula for interval extensions of isotonic operators. Below, we give extensions for the field operations in $\mathbb{R}$.

Definition 2.2.9 (Arithmetic in $\mathbb{I}(\mathbb{R})$)**.** *Let* $[a, b], [c, d] \in \mathbb{I}(\mathbb{R})$*, then the interval extensions of the operations* $+, -, \cdot, / : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$ *are given by*

- $[a, b] + [c, d] = [a + c, b + d]$
- $[a, b] - [c, d] = [a - d, b - c]$
- $[a, b] \cdot [c, d] = [\min\{a \cdot c, a \cdot d, b \cdot c, b \cdot d\}, \max\{a \cdot c, a \cdot d, b \cdot c, b \cdot d\}]$
- $[a, b]/[c, d] = [\min\{a/c, a/d, b/c, b/d\}, \max\{a/c, a/d, b/c, b/d\}]$

where in the case of division we demand that $c > 0$ *or* $d < 0$*, i. e. that the denominator interval does not contain zero.*

Note that the formulas for addition and subtraction have been simplified. This is due to $a \leq b$ and $c \leq d$ implying $\min\{a+c, a+d, b+c, b+d\} = a+c$ and $\max\{a+c, a+d, b+c, b+d\} = b+d$ for addition, and analogously for subtraction, $\min\{a-c, a-d, b-c, b-d\} = a-d$ and $\max\{a - c, a - d, b - c, b - d\} = b - c$. The main attraction of this arithmetic is that it is defined over sets, but computed over elements, *i. e.* it makes it possible to perform a possibly infinite number of evaluations by a finite number of computations. Clearly (2.2) may be used for operations that are monotonic in each argument, or more generally, whenever the maximal/minimal values are attained at endpoints of the argument intervals. This poperty holds true for unary operations/functions as well. Let $f : R \to R$ be a function and $\boldsymbol{f} : \mathbb{I}(R) \to \mathbb{I}(R)$ be an extension of f, then:

$$\boldsymbol{f}([a, b]) \supseteq \{f(x) \mid x \in [a, b]\} \tag{2.3}$$

for intervals $[a, b] \in \mathbb{I}(R)$. When we have equality in (2.3), we call $\boldsymbol{f}$ the *maximal extension* of f. When f attains its maximal/minimal values at a and b, as is the case for monotonic functions, then we get an analogue of (2.2):

$$\boldsymbol{f}([a, b]) = [\min\{f(a), f(b)\}, \max\{f(a), f(b)\}]$$

and we see that such functions have canonical maximal extensions. The containment property of interval arithmetic makes it useful for safely approximating numeric expressions. It suffices to define interval extensions of the constituent functions and operations, after which bounds for the compound expression are computed automatically. Here, the term *safe approximation* refers to an *outer* approximation, in the sense that all values of the approximated expression will be contained within the approximation, possibly also including superfluous values. This means that we may *safely* use the obtained approximation in place of the approximated expression when attempting to decide relations involving the approximated expression, leading to a *sound*, if not *complete* analysis, *i. e.* we can prove some but not all relations using outer approximations.

2.3 Approximation in complete semilattices

When we extend the set $\mathbb{I}(R)$ of R-intervals with a bottom element $\perp$ to $\mathbb{I}(R)_\perp$ there is a canonical way to extend interval extensions of *partial* functions and operations $R^n \rightharpoonup R$ to *total* ones, by mapping elements for which the function is undefined to $\perp$. Thus, let $n \in \mathbb{N}$ and $f : R^n \rightharpoonup R$ be a partial function, defined on a subset $S \subset R^n$, and let $\boldsymbol{f} : \mathbb{I}(R)^n \rightharpoonup \mathbb{I}(R)$ be an extension of f, then we may extend $\boldsymbol{f}$ to $\bar{\boldsymbol{f}} : \mathbb{I}(R)^n \to \mathbb{I}(R)_\perp$ as follows:

$$\bar{\boldsymbol{f}}(\boldsymbol{x}) = \begin{cases} \boldsymbol{f}(\boldsymbol{x}) & \text{if } \boldsymbol{x} \subseteq S \\ \perp & \text{otherwise} \end{cases} \tag{2.4}$$

Going one step futher, we may extend $\bar{\boldsymbol{f}} : \mathbb{I}(R)^n \to \mathbb{I}(R)_\perp$ to $\bar{\bar{\boldsymbol{f}}} : \mathbb{I}(R)^n_\perp \to \mathbb{I}(R)_\perp$ by mapping $\perp$ to $\perp$ and we see that any (possibly partial) R-valued function on R^n may be extended to a total $\mathbb{I}(R)_\perp$-valued function on $\mathbb{I}(R)^n_\perp$. This property, along with the containment property, make interval arithmetic very useful when reasoning about real functions. Containment means that we may use approximations, that may be easier to handle than the original expressions, to safely reason about relations over real numbers, with the bottom element $\perp$ acting as an exception that is "thrown" whenever a function may be called with arguments for which it is undefined and that then is propagated throughout the approximation.

We have seen that bounded semilattices are suitable for approximate reasoning and we have shown that given a partially ordered set R, there is a canonical bounded semilattice of intervals $\mathbb{I}(R)_\perp$ within which arithmetic in R may be approximated. While $\mathbb{I}(R)_\perp$ is suitable for approximating *elements* of R, it is less so for approximating *functions* on R.

The reason is that R-intervals track the extreme values of an expression, *i. e.* they use a uniform approximation over the entire domain of the function. While functions remain close to constant this approximation is acceptable, but when approximating a function over a set where it is varies significantly, we encounter the main drawback with interval arithmetic, the *information loss problem*. Different causes exist for this phenomenon; *wrapping effects* refer to the loss of information about the *shape* of the function graph.

Wrapping effects

As an example of wrapping consider approximating the square root over the interval $[0, 4]$. The maximal interval extension maps the interval to $[0, 2]$, *i. e.* we approximate the graph of $\sqrt{}$ over $[0, 4]$, with the rectangle $[0, 4] \times [0, 2]$. Now consider the relation

$$\sqrt{x} \geq \frac{1}{2}(x - 1)$$

over $[0, 4]$. We already have the approximation $[0, 4] \times [0, 2]$ of $\sqrt{}$, and the maximal extension of $\frac{x}{2} - \frac{1}{2}$ over $[0, 4]$ gives $[0, 4] \times [-\frac{1}{2}, \frac{3}{2}]$. Although the distance between the two graphs is at least $\frac{1}{2}$, the two rectangles intersect and we cannot prove the relation. As we alluded to above, the shape of the graphs have been lost to the approximation. There is a way around this problem, by reducing the size of the domain. When we approximate the relation above over $[1, 3]$ instead of $[0, 4]$, we obtain the rectangle $[1, 3] \times [1, \sqrt{3}]$ for the left hand side and the rectangle $[1, 3] \times [0, 1]$ for the right hand side. As the right hand rectangle clearly is above the left hand rectangle, we can safely conclude that the relation holds over $[1, 3]$. In part two we shall see that reducing the domain width is no panacea for wrapping effects as it has an intristic exponential cost limiting its applicability.

The dependency problem

A second instance of the information loss problem is the *dependency problem*. The issue here is that due to the representation of approximations as intervals, all information about dependencies between sub-expressions of two given expressions is lost. As an example consider approximating the expression $x - x$ over some interval $[a, b]$. The approximation first approximates each x by $[a, b]$ and then uses the maximal extension for the difference

as given by (2.3):

$$\begin{aligned} x - x &= [a, b] - [a, b] \\ &= [a - b, b - a] \\ &= (b - a) \cdot [-1, 1] \end{aligned}$$

Thus we see that the width of the interval approximating the actual value 0, is directly proportional to the width of the domain over which we are approximating.

3 Generation of correctness theorems

CONTENTS

In the following chapter we show how to derive proof obligations, or *verification conditions* (VCs), for a small imperative language. The approach is to extend the language with annotations, allowing for the expression of program properties directly in the program source code. In this sense, the approach implements a version of the interpretation of programs proposed by Floyd [39], as described in the introduction. The choice of model language was made to suit two purposes. Firstly, the language should model relevant aspects of SPARK Ada, so that the intuition and solutions developed for the model language carry over to practice. Secondly, the language should be small, so as to make the exposition of ideas and techniques as clear as possible.

In Section 3.1 we present the model imperative language PROC and in Section 3.2 PROC is equipped with the annotation language ANOT, extending PROC it to the annotated model language PEA. In Section 3.3 an operational semantics for PROC is introduced and extended to PEA, allowing for the formal definition of *operational correctness*. Section 3.4 presents a denotational, or logical, semantics for PEA based on predicate transformers in the spirit of Dijkstra's weakest precondition predicate transformer. It is used to formally define a notion of *denotational*, or *logical correctness* for PEA programs. Section 3.5 concludes the chapter with a theorem relating the two notions of correctness by comparing the strength of the resulting correctness theorems. To keep it small, the chosen language only models a subset of SPARK Ada programs.

3.1 Model language PROC

PROC is a slight variation on the WHILE language, as presented in [60], adapted for the purpose of demonstrating derivation of correctness theorems for imperative programs.

3.1.1 Syntax of PROC

We give the syntax of the PROC language in classical BNF but will withhold details for as long as possible to keep the presentation focused. We define the following syntactic

categories:

a	$\in$	**AExp**	arithmetic expressions	n	$\in$	**Num**	numerals
b	$\in$	**BExp**	Boolean expressions	p	$\in$	**Proc**	procedures
c	$\in$	**Conn**	Boolean connectives	r	$\in$	**Rel**	relations
d	$\in$	**Dom**	domain declarations	S	$\in$	**Stmt**	statements
D	$\in$	**Decl**	declarations	x	$\in$	**Var**	variables
f	$\in$	**Fun**	functions				

where we assume **Fun** to contain the usual arithmetic operations, **Rel** the equality and order relations and **Conn** the propositional connectives. The syntax of the PROC language is given by:

$$
\begin{array}{lcl}
a & ::= & x \mid n \mid f(a_1, \ldots, a_k) \\
b & ::= & \top \mid \bot \mid \neg b \mid b_1 \; c \; b_2 \mid a_1 \; r \; a_2 \\
S & ::= & \texttt{skip} \mid x := a \mid S_1 ; S_2 \mid p(a_1, \ldots, a_m; x_1, \ldots, x_n) \mid \\
 & & \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2\ \texttt{fi} \mid \texttt{while}\ b\ \texttt{do}\ S\ \texttt{od} \\
d & ::= & \texttt{null} \mid x \in [n_1, n_2] \\
D & ::= & d \mid D_1 ; D_2 \mid \texttt{proc}\ p(\texttt{in}\ d_1, \ldots, d_m;\ \texttt{out}\ d_{m+1}, \ldots, d_n)\ \texttt{is}\ D\ \texttt{begin}\ S\ \texttt{end}
\end{array}
$$

The statement S in a procedure declaration is called the *body* of the program. We demand that `in`-mode parameters of procedures are *immutable*, in the sense that they do not appear on the left hand side of assignments or as `out`-mode parameters in calls to subprocedures. We also demand that each `out`-mode parameter appears on the left hand side of an assignment or as `out`-mode parameter of a subprocedure call in S. Further, we demand that each variable x appearing in the procedure body, that is not a formal parameter of the procedure, has a domain $[n_1, n_2]$ declared in the procedure definition. We do not allow global variables, and require that values are passed between program and subprogram via the parameters of the subprogram. A *program* in PROC is given by a procedure declaration.

Figure 3.1 shows three example programs `one`, `two` and `three`. Given $f, g, h \in$ **Fun** and $n \in$ **Num**, `one` computes the composition of f with g, `two` attempts to compute a fixpoint of f and `three` computes the function h defined by:

$$
h(x) \;=\; \begin{cases} f(x) & \text{when} \quad x < n \\ g(x) & \text{when} \quad x \geq n \end{cases}
$$

```
proc one(in x ∈ 𝔽; out y ∈ 𝔽) is ... begin
    y := f(g(x))
end
```

```
proc two(in x ∈ 𝔽; out y ∈ 𝔽) is ... begin
    y := x; while f(y) ≠ y do y := f(y) od
end
```

```
proc three(in x ∈ 𝔽; out y ∈ 𝔽) is ... begin
    if x < n then y := f(x) else y := g(x) fi
end
```

Figure 3.1: Declarations of example programs `one`, `two` and `three`.

which is a generalisation of the classical step function defined as 0 for negative x and 1 otherwise. Note that the floating point format $\mathbb{F}$ in the domain declaration $x \in \mathbb{F}$ in the example programs corresponds to the subset of *finite* values in the format.

3.1.2 PROC functions as procedures

Functions in arithmetic expressions in PROC programs may be replaced with procedure calls by means of the map $f2p$ defined below.

$f2p$ defines, for each function $f : \mathbb{F}_1 \times \ldots \times \mathbb{F}_m \to \mathbb{F}$, a procedure p_f declared by:

$$\texttt{proc}\ p_f(\texttt{in}\ x_1 \in \mathbb{F}_1, \ldots, x_m \in \mathbb{F}_m;\ \texttt{out}\ y \in \mathbb{F})\ \texttt{is null begin}\ y := f(x_1, \ldots, x_m)\ \texttt{end}$$

and adds the definition of p_f to the program declaration, provided that p_f is not the program itself and that it does not yet appear in the program declaration. Assignments of function values become interchangeable with procedure calls:

$$y := f(a_1, \ldots, a_k) \quad \leftrightarrow \quad p_f(a_1, \ldots, a_k; y)$$

Note that procedures having *precisely one* `out`-mode parameter implicitly define functions through this correspondence and that the function defined by p_f is f itself. We will often use the same name for the function and procedure versions of a subprogram, since the appropriate version always can be identified from the context, as functions only appear in expressions and procedures in statements and declarations.

We define a map $a2p$ that *unrolls* composite expressions within a statement into a se-

```
proc one(in x ∈ 𝔽 ; out y ∈ 𝔽) is
    proc p_f(in x ∈ 𝔽; out y ∈ 𝔽) is null begin y := f(x) end;
    proc p_g(in x ∈ 𝔽; out y ∈ 𝔽) is null begin y := g(x) end;
    y1 ∈ 𝔽
begin
     p_g(x;y1); p_f(y1;y)
end
```

```
proc two(in x ∈ 𝔽; out y ∈ 𝔽) is ... begin
    y := x; p_f(y;y1); while y ≠ y1 do p_f(y;y); p_f(y,y1) od
end
```

```
proc three(in x ∈ 𝔽; out y ∈ 𝔽) is ... begin
    if x < n then p_f(x,y) else p_g(x,y) fi
end
```

Figure 3.2: Declarations resulting from applying $a2p$ and $f2p$ to the example programs from Figure 3.1 now use procedure calls. The omitted procedure declarations for p_f and p_g in `two` and `three` are identical to those in `one`.

quence of statements using the correspondence defined by $f2p$:

$$
\begin{aligned}
x := f(\ldots) &\mapsto p_f(\ldots; x) \\
p(\ldots f(\ldots)\ldots) &\mapsto p_f(\ldots; x_1);\ p(\ldots x_1 \ldots) \\
\texttt{if}\ b(f(\ldots))\ \texttt{then}\ \ldots &\mapsto p_f(\ldots; x_1);\ \texttt{if}\ b(x_1)\ \texttt{then}\ \ldots \\
\texttt{while}\ b(f(\ldots))\ \texttt{do}\ S\ \texttt{od} &\mapsto p_f(\ldots; x_1);\ \texttt{while}\ b(x_1)\ \texttt{do}\ S;\ p_f(\ldots; x_1)\ \texttt{od}
\end{aligned}
$$

where x_1 is a fresh variable with domain equal to the codomain of f. Each new variable introduced this way is added by $a2p$ to the declaration of the program.

While $f2p$ declares new procedures in a program, corresponding to functions appearing in the program body, $a2p$ transforms the body of a program by replacing composite expressions with sequences of procedure calls and adds domain declarations for local variables to the program definition.

By repeated application of $f2p$ and $a2p$ to a PROC program we obtain a program with a body that is a sequence of statements entirely without functions. In Figure 3.2 we show the result of applying $f2p$ and $a2p$ to the example programs `one`, `two` and `three` in Figure 3.1.

3.2 Annotated model language PEA

The notion of *correctness* for a program in PROC is defined relative to a *contract*, *i. e.* a statement of conditions under which the program is to be called and guarantees on the results for legal calls. The conditions are formalised as a *precondition* predicate on `in` mode parameters and guarantees are expressed as a *postcondition* predicate on the parameters of the program. The pre- and postcondition predicates are expressed in an *annotation language*, *i. e.* a formalisation of first order predicate logic embedded within the programming language. In the following sections we give the definition of the annotation language ANOT for PROC and show how to embed ANOT within PROC, yielding our model annotated language PEA.

3.2.1 Syntax of ANOT

ANOT extends the arithmetic and Boolean expression sublanguages of PROC, defined in Section 3.1.1, by providing additional functions, relations and quantifiers. To this end we define the following syntactic categories:

$$\begin{array}{lcll} \gamma & \in & \textbf{Abs} & \text{abstract functions} \\ \tau & \in & \textbf{Term} & \text{terms} \\ \varphi & \in & \textbf{Form} & \text{formulas} \end{array}$$

and syntax of the ANOT language

$$\begin{array}{lcl} \tau & ::= & a \mid \gamma(\tau_1, \ldots, \tau_k) \\ \varphi & ::= & b \mid \neg\varphi \mid \tau_1\ r\ \tau_2 \mid \varphi_1\ c\ \varphi_2 \mid \exists x \in [\tau_1, \tau_2]\ .\ \varphi \mid \forall x \in [\tau_1, \tau_2]\ .\ \varphi \end{array}$$

In ANOT we have syntactic sugar and the usual operator precedence for arithmetic operations allowing us to use infix notation for binary operations and omission of many parentheses.

3.2.2 Syntax of PEA

Correctness of a program is defined relative to a description of the intended behaviour of the program. Such a description is often called a *contract* as it may be seen as a contract between the provider of the program and its users. In PEA precondition predicates are preceded by the keyword `pre` and postcondition predicates are preceded by the keyword

post. A *loopcut assertion* is a predicate on program variables that is expected to hold when execution reaches it. Assertion predicates in PEA are preceded by the keyword assert. The syntax of PEA is obtained from PROC by adding contracts to procedure declarations:

$$\texttt{proc}\ p(\texttt{in}\ x_1 \in \mathbb{F}_1, \ldots, x_m \in \mathbb{F}_m;\ \texttt{out}\ x_{m+1} \in \mathbb{F}_{m+1}, \ldots, x_n \in \mathbb{F}_n)\ \texttt{pre}\ \varphi_1\ \texttt{post}\ \varphi_2\ \texttt{is}\ \ldots$$

where φ_1 depends on $x_1 \ldots x_m$ and φ_2 depends on $x_1 \ldots x_n$, and adding loopcut assertions to PROC while statements:

$$\texttt{while}\ b\ \texttt{assert}\ \varphi\ \texttt{do}\ S\ \texttt{od}$$

Thus, PEA programs may be translated to PROC by deleting all pre, post, and assert keywords and ANOT formulas from the program text.

3.2.3 PEA functions as procedures

In PEA we demand that functions have contracts. As described in Section 3.1.2, composite expressions may be unrolled into sequences of procedure calls. The map $f2p$ taking functions to procedures in PROC is extended to PEA by lifting function contracts to procedure contracts. As functions have a single anonymous return value they require a special kind of postcondition, called a *return condition*, naming the returned value so that its properties may be stated. Thus, each function $f \in \mathbf{Fun}$ in PEA with parameters $x_1 \ldots x_k$ has a contract:

$$\texttt{pre}\ \varphi_f\ \texttt{return}\ y\ .\ \psi_f \tag{3.1}$$

where $y \in \mathbf{Var}$, φ_f depends on $x_1 \ldots x_k$ and ψ_f depends on $x_1 \ldots x_k$ and y. The return condition $\texttt{return}\ y\ .\ \psi_f$ is pronounced "given the values $x_1 \ldots x_k$ the function f returns a value y such that $\psi_f(x_1, \ldots, x_k, y)$ holds". A PEA function $f : \mathbb{F}_1 \times \ldots \times \mathbb{F}_m \to \mathbb{F}$ with the contract (3.1) now corresponds to the PEA procedure declaration:

$$\texttt{proc}\ p_f(\texttt{in}\ x_1 \in \mathbb{F}_1, \ldots, x_m \in \mathbb{F}_m\ ;\ \texttt{out}\ y \in \mathbb{F})\ \texttt{pre}\ \varphi_f\ \texttt{post}\ \psi_f\ \texttt{is}\ \texttt{null}$$
$$\texttt{begin}\ y := f(x_1, \ldots, x_k)\ \texttt{end}$$

Our work addresses the problem of choosing apropriate pre- and postconditions for programs, such as one in Figure 3.3, when given contracts for its subprograms. In the case of one we wish to find $\varphi_{\texttt{one}}$ and $\psi_{\texttt{one}}$, given φ_f, ψ_f, φ_g and ψ_g. We shall make the problem precise and lay the ground for constructing a solution in the following section.

```
proc one(in x ∈ 𝔽 ; out y ∈ 𝔽) pre φ_one post ψ_one is
    proc f(in x ∈ 𝔽; out y ∈ 𝔽) pre φ_f post ψ_f is ... end;
    proc g(in x ∈ 𝔽; out y ∈ 𝔽) pre φ_g post ψ_g is ... end
begin
    y := f(g(x))
end
```

Figure 3.3: Declaration of annotated example program one using procedure declaration to implicitly define the functions f and g.

3.3 Structural operational semantics for PROC and PEA

To give correctness theorems for PEA programs a precise meaning this section presents a *structural operational semantics* $[\![\cdot]\!]^{SOS}$ for PEA, introduced as a function from statements to functions on state. In Section 3.3.1 we give the definition of the semantics for PROC and PEA and in section 3.3.2 we use $[\![\cdot]\!]^{SOS}$ to formally define *correctness* of terminating PEA programs.

3.3.1 The map $[\![\cdot]\!]^{SOS}$

We begin by defining elements of the set $\mathbf{State}_{\mathbb{R}}$ of $\mathbb{R}$-*states*. An $\mathbb{R}$-state σ is a partial function $\mathbf{Var} \rightharpoonup \mathbb{R}$ taking a variable x, with domain declaration $x \in [\underline{x}, \overline{x}]$, to a real number $\sigma(x) \in [\underline{x}, \overline{x}]$. Numerals are interpreted as real numbers through the map $\mathcal{N} : \mathbf{Num} \to \mathbb{R}$. An $\mathbb{R}$-state encodes the program's *state vector* at a specific point of the execution of the program. Initially, only variables corresponding to the `in`-mode parameters of the program are initialised to actual values. The operational semantics then defines how execution of consecutive statements in the program updates the values of variables in the state vector. We assume that the program body has been *fully simplified* using the transformations $a2p$ and $f2p$, from Section 3.1.2 on page 34, so that arithmetic expressions are either numerals or variables. Evaluation of expressions is assumed not to have effect on state, we therefore drop the semantic bracket around expressions to make the distinction

explicit:

$$\begin{aligned} x\sigma &= \sigma(x) \\ n\sigma &= \mathcal{N}(n) \\ f(a_1,\ldots,a_k)\sigma &= [\![f]\!]^{SOS}\,(a_1\sigma,\ldots,a_k\sigma) \\ (a_1\ r\ a_2)\sigma &= (a_1\sigma)\ [\![r]\!]^{SOS}\ (a_2\sigma) \\ (b_1\ c\ b_2)\sigma &= (b_1\sigma)\ [\![c]\!]^{SOS}\ (b_2\sigma) \\ (\neg b)\sigma &= \neg(b\sigma) \end{aligned} \tag{3.2}$$

where $[\![f]\!]^{SOS}$, $[\![r]\!]^{SOS}$ and $[\![c]\!]^{SOS}$ are the semantic counterparts of the syntactic elements f, r and c in PROC. When there is no risk of confusion we shall use the same notation for the syntactic and semantic version of functions, relations and connectives.

The value of $a\sigma$ is either a real number or an *exceptional value*, resulting from the application of a partial function outside its domain, denoted by $\bullet$. Each $[\![f]\!]^{SOS}$ thus becomes a *total* $\mathbb{R} \cup \{\bullet\}$-valued function. Functions applied to $\bullet$ result in $\bullet$ and relations applied to $\bullet$ return false. Semantically, $\bullet$ is intended to model the special floating point value NaN, defined in Section 4.2.2 on page 60.

Executing a `skip` statement has no effect on the state:

$$[\![\texttt{skip}]\!]^{SOS}\,\sigma \;=\; \sigma \tag{3.3}$$

or equivalently, $[\![\texttt{skip}]\!]^{SOS}$ is the identity function. In the case of an assignment $x := a$ we have:

$$[\![x := a]\!]^{SOS}\,\sigma \;=\; \sigma[x \mapsto a\sigma] \tag{3.4}$$

where $\sigma[x \mapsto a\sigma]$ denotes the state obtained from σ by updating the value of the variable x to $a\sigma$. Note that it is implicitly assumed here that evaluation of the expression a does not have any effect on the state. Executing a sequence of statements has the effect of updating the state in the order prescribed by the sequence:

$$[\![S_1; S_2]\!]^{SOS}\,\sigma \;=\; [\![S_2]\!]^{SOS}\left([\![S_1]\!]^{SOS}\,\sigma\right) \tag{3.5}$$

or equivalently, sequencing $[\![S_1; S_2]\!]^{SOS}$ is mapped to *reverse* composition $[\![S_2]\!]^{SOS}\ [\![S_1]\!]^{SOS}$. The effect of executing a conditional statement depends on the value of the conditional:

$$[\![\texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2\ \texttt{fi}]\!]^{SOS}\,\sigma \;=\; \begin{cases} [\![S_1]\!]^{SOS}\,\sigma & \text{if} \quad b\sigma \text{ is true} \\ [\![S_2]\!]^{SOS}\,\sigma & \text{if} \quad \neg b\sigma \text{ is true} \end{cases} \tag{3.6}$$

Execution of a loop repeatedly updates the state according to the effect of the loop body until the exit condition evaluates to true:

$$[\![\texttt{while}\ b\ \texttt{do}\ S\ \texttt{od}]\!]^{SOS}\ \sigma\ =\ \begin{cases} [\![S; \texttt{while}\ b\ \texttt{do}\ S\ \texttt{od}]\!]^{SOS}\ \sigma & \text{if} \quad b\sigma \text{ is true} \\ \sigma & \text{if} \quad \neg b\sigma \text{ is true} \end{cases} \tag{3.7}$$

We note that the definition above introduces the possibility for non-termination, which is correct as `while` statements may loop indefinitely, as is the case when $b = \top$. Note also that, as in the case of assignments above, the evaluation of the Boolean expression in conditionals and loops does not affect the state. As we have disallowed global variables in PROC the semantics of a call to a procedure p is declared by:

$$\texttt{proc}\ p(\texttt{in}\ x_1 \in \mathbb{F}_1, \ldots, x_m \in \mathbb{F}_m;\ \texttt{out}\ x_{m+1} \in \mathbb{F}_{m+1}, \ldots, x_n \in \mathbb{F}_n)\ \texttt{is} \ldots \texttt{begin}\ S\ \texttt{end}$$

may be expressed in terms of states as follows:

$$[\![p(a_1, \ldots, a_m; y_{m+1}, \ldots, y_n)]\!]^{SOS}\ \sigma\ =\ \sigma[y_{m+1} \mapsto x_{m+1}\sigma', \ldots, y_n \mapsto x_n\sigma'] \tag{3.8}$$

where σ' is obtained from σ by executing the body of p in a state where no variable other than $x_1, \ldots, x_m$ has been initialised:

$$\sigma'\ =\ [\![S]\!]^{SOS}\left([x_1 \mapsto a_1\sigma, \ldots, x_m \mapsto a_m\sigma]\right)$$

The definition ensures that no values can pass between the main program's state and the state of the subprogram, other than *explicitly* through the subprogram's parameters. This property simplifies the definition of operational semantics given above by eliminating the need to factor states into environments and stores maintaining global state. For the case of semantics that allow subprogram calls to read and modify variables that do not appear explicitly in parameters see *e.g.* [60, 63]. The operational semantics for PROC are readily extended to PEA by treating annotations as comments:

$$[\![\texttt{proc}\ p(\ldots)\ \texttt{pre}\ \varphi\ \texttt{post}\ \psi\ \texttt{is}\ \ldots]\!]^{SOS}\ =\ [\![\texttt{proc}\ p(\ldots)\ \texttt{is}\ \ldots]\!]^{SOS}$$

$$[\![\texttt{while}\ b\ \texttt{assert}\ \varphi\ \texttt{do} \ldots]\!]^{SOS}\ =\ [\![\texttt{while}\ b\ \texttt{do} \ldots]\!]^{SOS}$$

3.3.2 Operational correctness for PEA programs

The evaluation $b\sigma$ of boolean expressions b at states σ is extended to formulas by evaluating each boolean expression in the formula:

$$\varphi(b)\sigma \quad \Leftrightarrow \quad \varphi(b\sigma) \tag{3.9}$$

Using $[\![\cdot]\!]^{SOS}$ and the notation $\varphi\sigma$ for the predicate φ at the state σ we may now formally define the notion of correctness theorem corresponding to the Hoare [47] triple $\{\varphi\}S\{\psi\}$ for a terminating PEA statement S, as follows:

$$\forall\sigma \,.\, \varphi\sigma \rightarrow \psi([\![S]\!]^{SOS}\ \sigma) \tag{3.10}$$

i. e. if the precondition holds at some initial state σ, then the postcondition holds at the final state $[\![S]\!]^{SOS}\ \sigma$. The definition (3.10) is extended to programs with subprogram declarations:

$$\texttt{proc}\ p(\texttt{in}\ x_1 \in \mathbb{F}_1 \ldots;\ \texttt{out}\ x_{m+1} \in \mathbb{F}_{m+1} \ldots)\ \texttt{pre}\ \varphi\ \texttt{post}\ \psi\ \texttt{is}\ D\ \texttt{begin}\ S\ \texttt{end}$$

by demanding that (3.10) be proved for the main program p and each subprogram p_i, with pre- and postcondition φ_i and ψ_i and program body S_i, declared in D, then (3.10) becomes:

$$\forall\sigma \,.\, \varphi\sigma \rightarrow \psi([\![S]\!]^{SOS}\ \sigma)\ \wedge\ \forall i \,.\, \varphi_i\sigma \rightarrow \psi_i([\![S_i]\!]^{SOS}\ \sigma)\ \wedge\ \ldots \tag{3.11}$$

In the following section we present a method for deriving proof obligations that imply this statement through *predicate transformer semantics*.

3.4 Predicate transformer semantics for PEA

In the preceding section an operational notion of correctness was described. It has the advantage of being intuitive but relies on both the initial and the end state of the program. To enable static, or compile time, verification one needs a notion of correctness that refers to the initial state only, *i. e.* one that does not rely on the operational semantics of the language.

In this section we formalise an alternative notion of correctness for the annotated model language PEA from Section 3.2 on page 36. As discussed in the introduction, the idea of Floyd [39] was to reason over the *control flow graph* (CFG) of the program. *Invariants* are placed in the body of `while` statements, yielding *loop cuts*, which define cutpoints in the CFG. The analysis is thereby split into analyses over the *control flow paths* (CFPs) between the beginning, end and cutpoints in the CFG. The following section will present one way of deriving proof obligations, called *verification conditions* (VCs) and how they combine to form the *correctness theorem* (CT) for the entire program.

We present a slight variation on the algorithm suggested above. Instead of computing VCs for CFPs we will present a *predicate transformer semantics* in the spirit of Dijkstra's *weakest precondition semantics* with the differences that we relax the requirement that the transformer should produce the *weakest* precondition and that we shall *assume* termination, rather than, as Dijkstra does, demand that the result guarantees termination. The idea is to compute the correctness theorem directly from the syntax of the program. We define the map $\mathcal{CT}$ from program declarations to formulas quantified over state, taking annotated programs with subprogram declarations to their associated CT, in terms of the predicate transformer $[\cdot]$ from statements to maps from formulas to formulas, taking statements to the corresponding map on formulas.

3.4.1 The map $\mathcal{CT}$

The map $\mathcal{CT}$ gives us the CT of a PEA program, it is defined inductively over the syntactic structure of PEA declarations:

$$\mathcal{CT}(d) = \texttt{true}$$

i. e. the null program and domain declarations for local variables are trivially correct. The CT of a sequence of declarations demands we prove the CT for each declaration in the sequence:

$$\mathcal{CT}(D_1; D_2) = \mathcal{CT}(D_1) \wedge \mathcal{CT}(D_2)$$

and the CT of a procedure declaration demands we prove each declared subprocedure and the proof obligation that the body S satisfies the contract $\texttt{pre}\ \varphi\ \texttt{post}\ \psi$, given that each subprocedure is correct:

$$\mathcal{CT}(\texttt{proc} \ldots \texttt{pre}\ \varphi\ \texttt{post}\ \psi\ \texttt{is}\ D\ \texttt{begin}\ S\ \texttt{end}) = \mathcal{CT}(D) \wedge \forall\sigma\ .\ \Big(\varphi \rightarrow [S]\psi\Big)\sigma$$

where $\forall\sigma\ .\ \phi\sigma \equiv \forall x_1 \in \boldsymbol{x}_1 \ldots \forall x_k \in \boldsymbol{x}_k\ .\ \phi$, assuming the set of all free variables in ϕ is $\{x_1, \ldots, x_k\}$. The domain $\boldsymbol{x}_i$ for each variable x_i is obtained from its declaration $d_i = x_i \in \boldsymbol{x}_i$ as procedure parameter or local variable. Thus, quantifying a predicate over all states is taken to mean quantification over each free variable in the predicate, ranging over its declared domain. We will assume that the program body has been *fully simplified* using $a2p$ and $f2p$, so that expressions are either numerals or variables. We make this assumption to simplify the exposition of CT generation.

3.4.2 The predicate transformer $[\cdot]$

Our take on Dijkstra's *weakest precondition* [34] predicate transformer produces for each PROC statement S a map $[S]$ on formulas by *pulling back* a predicate on state *after* a statement to a predicate on state *before* the statement. The property we wish $[\cdot]$ to have is that if the predicate $[S]\phi$, resulting from an application of $[S]$ to the predicate ϕ, holds at a state σ, then ϕ holds at the state resulting from executing S beginning at σ. `skip` statements do not affect state, and should therefore act as the identity on predicates:

$$[\texttt{skip}]\phi = \phi \tag{3.12}$$

while assignments update the state of a variable to that of the assigned expression at the initial state:

$$[x := a]\phi = \phi[{}^{a}/_{x}] \tag{3.13}$$

where $\phi[{}^{a}/_{x}]$ denotes ϕ with each occurrence of x replaced by a.

Pulling back a predicate through a sequence of statements correspond to pulling them through one statement at a time:

$$[S_1; S_2]\phi \;=\; [S_1]\Big([S_2]\phi\Big) \tag{3.14}$$

or, equivalently, the action of a sequence $[S_1; S_2]$ is the composition of the action of the constituent statements $[S_1][S_2]$.

The execution of a conditional statement can take either of two paths, depending on the value of the Boolean guard b, so we generate a proof obligation for each alternative:

$$[\texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2\ \texttt{fi}]\phi \;=\; \Big(b \to [S_1]\phi\Big) \wedge \Big(\neg b \to [S_2]\phi\Big) \tag{3.15}$$

There are four paths to consider for a while statement. When b holds we enter the loop and stop at the loop cut θ. If b holds after a traversal of the loop we traverse it again, starting and ending at θ. Once θ is false, we exit the loop and if b was false to begin with, we skip past the loop. In total we get four proof obligations:

$$\begin{gathered}[\texttt{while}\ b\ \texttt{assert}\ \theta\ \texttt{do}\ S\ \texttt{od}]\phi \;= \\ \Big(b \to \theta\Big) \wedge \Big\langle \theta \to [S](b \to \theta)\Big\rangle \wedge \Big\langle \theta \to [S](\neg b \to \phi)\Big\rangle \wedge \Big(\neg b \to \phi\Big)\end{gathered} \tag{3.16}$$

where the angled brackets $\langle\cdot\rangle$ denote implicit renaming of all variables, preventing substitution in proof obligations for paths beginning inside the loop.

To illustrate the use of $\langle\cdot\rangle$ we apply $[\cdot]$ to a statement corresponding to a loop preceded by an initialisation statement:

$$[S_1 \,;\, \texttt{while}\ b\ \texttt{assert}\ \theta\ \texttt{do}\ S_2\ \texttt{od}]\,\phi \;=$$
$$[S_1]\Big(\big(b \to \theta\big) \wedge \big\langle \theta \to [S_2](b \to \theta)\big\rangle \wedge \big\langle \theta \to [S_2](\neg b \to \phi)\big\rangle \wedge \big(\neg b \to \phi\big)\Big) =$$
$$\Big([S_1](b \to \theta)\Big) \wedge \big\langle \theta \to [S_2](b \to \theta)\big\rangle \wedge \big\langle \theta \to [S_2](\neg b \to \phi)\big\rangle \wedge \Big([S_1](\neg b \to \phi)\Big)$$

i. e. $[\cdot]$ acts as the identity on $\langle\cdot\rangle$-bracketed expressions. Evaluation of a formula at a state treats $\langle\cdot\rangle$ as grouping $(\cdot)$. As a result, initialisation is only taken into account in proof obligations for paths starting before the loop, as one would expect.

A call $p(a_1, \ldots, a_m; y_{m+1}, \ldots, y_n)$ to a procedure with the specification:

$$p(\texttt{in}\ x_1 \in \mathbb{F}_1, \ldots, x_m \in \mathbb{F}_m;\ \texttt{out}\ x_{m+1} \in \mathbb{F}_{m+1}, \ldots, x_n \in \mathbb{F}_n)\ \texttt{pre}\ \varphi\ \texttt{post}\ \psi\ \texttt{is}\ldots$$

is modelled by a proof obligation demanding that the call is *legal*, *i. e.* that the precondition is satisfied at the time of call, and a proof obligation corresponding to the assumption that the contract is *valid*, *i. e.* that the postcondition holds at the point of return from legal calls. In order to make sure that the postcondition only states current properties, `out`-mode parameters $y_{m+1} \ldots y_n$ are substituted by fresh variables $z_{m+1} \ldots z_n$, with domains equal to those of the corresponding substituted variable: $y_j \in [n_1, n_2] \Rightarrow z_j \in [n_1, n_2]$. We get the following two proof obligations:

$$[p(a_1, \ldots, a_m; y_{m+1}, \ldots, y_n)]\phi \;=\; \varphi' \wedge \Big(\psi' \to \phi[{}^{z_{m+1}}/_{y_{m+1}}] \cdots [{}^{z_n}/_{y_n}]\Big) \tag{3.17}$$

where $\varphi' \equiv \varphi[{}^{a_1}/_{x_1}] \cdots [{}^{a_m}/_{x_m}]$ and $\psi' \equiv \psi[{}^{a_1}/_{x_1}] \cdots [{}^{a_m}/_{x_m}][{}^{z_{m+1}}/_{x_{m+1}}] \cdots [{}^{z_n}/_{x_n}]$.

3.5 $\mathcal{CT}$ is stronger than $[\![\cdot]\!]^{SOS}$

In Section 3.3 we gave an operational semantics $[\![\cdot]\!]^{SOS}$ for PEA and defined the CT for a PEA program using $[\![\cdot]\!]^{SOS}$ in (3.11). In Section 3.4 we gave a predicate transformer semantics $\mathcal{CT}$, generating the CT for a PEA programs using the predicate transformer $[\cdot]$. In this section we show that the CT derived using $[\cdot]$ implies the CT formulated using $[\![\cdot]\!]^{SOS}$, providing simpler goals for verification.

Theorem 3.5.1. *Correctness theorems derived by $\mathcal{CT}$ imply operational correctness.*

Proof. It suffices to show:

$$\big([S]\phi\big)\sigma \to \phi\big([\![S]\!]^{SOS}\,\sigma\big) \tag{3.18}$$

for all $\mathbb{R}$-states σ, formulas ϕ and terminating statements S in PEA, since $\forall\sigma\,.\,\varphi\sigma \to \big([S]\psi\big)$ and (3.18) together imply (3.10). We proceed to prove (3.18) by induction on $S \in \mathbf{Stmt}$.

- for *skip* statements the result follows directly:

$$\begin{aligned}\big([\texttt{skip}]\phi\big)\sigma &\Leftrightarrow \phi\sigma && \text{by (3.12)}\\ &\Leftrightarrow \phi\big([\![\texttt{skip}]\!]^{SOS}\,\sigma\big) && \text{by (3.3)}\end{aligned}$$

- as does the case for *assignment* statements:

$$\begin{aligned}\big([x := a]\phi\big)\sigma &\Leftrightarrow \big(\phi[{}^{a}/_{x}]\big)\sigma && \text{by (3.13)}\\ &\Leftrightarrow \phi\big(\sigma[x \mapsto a\sigma]\big) && \text{by (3.2) and (3.9)}\\ &\Leftrightarrow \phi\big([\![x := a]\!]^{SOS}\,\sigma\big) && \text{by (3.4)}\end{aligned}$$

- the case for *sequencing* statements requires induction steps because we are proving the case without knowing what each component statement does:

$$\begin{aligned}\big([S_1;S_2]\phi\big)\sigma &\Leftrightarrow \big([S_1]\big([S_2]\phi\big)\big)\sigma && \text{by (3.14)}\\ &\Rightarrow [S_2]\phi\big([\![S_1]\!]^{SOS}\,\sigma\big) && \text{by induction hypothesis}\\ &\Rightarrow \phi\big([\![S_2]\!]^{SOS}\,\big([\![S_1]\!]^{SOS}\,\sigma\big)\big) && \text{by induction hypothesis}\\ &\Leftrightarrow \phi\big([\![S_1; S_2]\!]^{SOS}\,\sigma\big) && \text{by (3.5)}\end{aligned}$$

- the *conditional* case must likewise use the induction hypothesis:

$$\big([\texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2\ \texttt{fi}]\phi\big)\sigma \Leftrightarrow \big(\big(b \to [S_1]\phi\big) \wedge \big(\neg b \to [S_2]\phi\big)\big)\sigma$$

by (3.15), which is equivalent to

$$\big(b\sigma \to \big([S_1]\phi\big)\sigma\big) \wedge \big(\neg b\sigma \to \big([S_2]\phi\big)\sigma\big)$$

by (3.2) and (3.9), which implies

$$\big(b\sigma \to \phi\big([\![S_1]\!]^{SOS}\,\sigma\big)\big) \wedge \big(\neg b\sigma \to \phi\big([\![S_2]\!]^{SOS}\,\sigma\big)\big)$$

by the induction hypothesis. This is equivalent to

$$\begin{cases} \phi\big([\![S_1]\!]^{SOS}\ \sigma\big) & \text{if} \quad b\sigma \text{ is true} \\ \phi\big([\![S_2]\!]^{SOS}\ \sigma\big) & \text{if} \quad \neg b\sigma \text{ is true} \end{cases}$$

which, by (3.6), is equivalent to

$$\phi\big([\![\texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2\ \texttt{fi}]\!]^{SOS}\ \sigma\big)$$

- the case for terminating *while* statements requires some setting up. We define states $\sigma_i = [\![S]\!]^{SOS}\ \sigma_{i-1}$ for all $i > 0$ and $\sigma_0 = \sigma$. Since S is assumed to terminate, it follows that there exists a least non-negative integer n such that:

$$\forall i \geq 0\ .\ (i < n \rightarrow b\sigma_i) \wedge \neg b\sigma_n$$

i. e. σ_n is the state at which computation exits the loop. We have

$$\big([\texttt{while}\ b\ \texttt{assert}\ \theta\ \texttt{do}\ S\ \texttt{od}]\phi\big)\sigma \ \Leftrightarrow$$
$$\big(b \rightarrow \theta\big)\sigma \ \wedge\ \big\langle \theta\sigma \rightarrow \big([S](b \rightarrow \theta)\big)\sigma\big\rangle \ \wedge$$
$$\big\langle \theta\sigma \rightarrow \big([S](\neg b \rightarrow \phi)\big)\sigma\big\rangle \ \wedge\ \big(\neg b \rightarrow \phi\big)\sigma$$

for all σ, by (3.16). In particular, this implies

$$\big(b \rightarrow \theta\big)\sigma \ \wedge\ \big\langle \theta\sigma \rightarrow \big([S](b \rightarrow \theta)\big)\sigma\big\rangle \ \wedge$$
$$\big\langle \theta\sigma_1 \rightarrow \big([S](b \rightarrow \theta)\big)\sigma_1\big\rangle \ \wedge\ \ldots\ \wedge\ \big\langle \theta\sigma_{n-2} \rightarrow \big([S](b \rightarrow \theta)\big)\sigma_{n-2}\big\rangle \ \wedge$$
$$\big\langle \theta\sigma_{n-1} \rightarrow \big([S](\neg b \rightarrow \phi)\big)\sigma_{n-1}\big\rangle \ \wedge\ \big(\neg b \rightarrow \phi\big)\sigma$$

which implies

$$\big(b \rightarrow \theta\big)\sigma \ \wedge\ \big\langle \theta\sigma \rightarrow (b \rightarrow \theta)\big([\![S]\!]^{SOS}\ \sigma\big)\big\rangle \ \wedge$$
$$\big\langle \theta\sigma_1 \rightarrow (b \rightarrow \theta)\big([\![S]\!]^{SOS}\ \sigma_1\big)\big\rangle \ \wedge\ \ldots\ \wedge\ \big\langle \theta\sigma_{n-2} \rightarrow (b \rightarrow \theta)\big([\![S]\!]^{SOS}\ \sigma_{n-2}\big)\big\rangle \ \wedge$$
$$\big\langle \theta\sigma_{n-1} \rightarrow (\neg b \rightarrow \phi)\big([\![S]\!]^{SOS}\ \sigma_{n-1}\big)\big\rangle \ \wedge\ \big(\neg b \rightarrow \phi\big)\sigma$$

by induction hypothesis, which is equivalent to

$$\big(b \rightarrow \theta\big)\sigma \ \wedge\ \big\langle \theta\sigma \rightarrow (b \rightarrow \theta)\sigma_1\big\rangle \ \wedge$$
$$\big\langle \theta\sigma_1 \rightarrow (b \rightarrow \theta)\sigma_2\big\rangle \ \wedge\ \ldots\ \wedge\ \big\langle \theta\sigma_{n-2} \rightarrow (b \rightarrow \theta)\sigma_{n-1}\big\rangle \ \wedge$$
$$\big\langle \theta\sigma_{n-1} \rightarrow (\neg b \rightarrow \phi)\sigma_n\big\rangle \ \wedge\ \big(\neg b \rightarrow \phi\big)\sigma$$

by the definition of σ_i above. By (3.2) and (3.9) this is equivalent to

$$\begin{gathered}\Big(b\sigma \to \theta\sigma\Big) \;\wedge\; \Big\langle \theta\sigma \to (b\sigma_1 \to \theta\sigma_1)\Big\rangle \;\wedge \\ \Big\langle \theta\sigma_1 \to (b\sigma_2 \to \theta\sigma_2)\Big\rangle \;\wedge\; \ldots \;\wedge\; \Big\langle \theta\sigma_{n-2} \to (b\sigma_{n-1} \to \theta\sigma_{n-1})\Big\rangle \;\wedge \\ \Big\langle \theta\sigma_{n-1} \to (\neg b\sigma_n \to \phi\sigma_n)\Big\rangle \;\wedge\; \Big(\neg b\sigma \to \phi\sigma\Big)\end{gathered} \tag{3.19}$$

which implies $\neg b\sigma \to \phi\sigma$. If $n = 0$, then $\neg b\sigma$ is true and then (3.19) implies $\phi\sigma$, which by (3.7) is equivalent to

$$\phi\Big([\![\texttt{while}\ b\ \texttt{assert}\ \theta\ \texttt{do}\ S\ \texttt{od}]\!]^{SOS}\ \sigma\Big)$$

as required. If instead $n > 0$, then $b\sigma_i$ is true for each $i < n$ and $\neg b\sigma_n$ is true. Therefore, (3.19) is equivalent to

$$\begin{gathered}\theta\sigma \;\wedge\; \Big\langle \theta\sigma \to \theta\sigma_1\Big\rangle \;\wedge\; \Big\langle \theta\sigma_1 \to \theta\sigma_2\Big\rangle \;\wedge\; \ldots \\ \ldots \;\wedge\; \Big\langle \theta\sigma_{n-2} \to \theta\sigma_{n-1}\Big\rangle \;\wedge\; \Big\langle \theta\sigma_{n-1} \to \phi\sigma_n\Big\rangle\end{gathered}$$

which implies $\phi\sigma_n$. By definition, σ_n is the state at termination of the loop, *i. e.* under the assumptions made on n above, $\phi\sigma_n$ is equivalent to

$$\phi\Big([\![\texttt{while}\ b\ \texttt{assert}\ \theta\ \texttt{do}\ S\ \texttt{od}]\!]^{SOS}\ \sigma\Big)$$

as required.

- A *call* to the *procedure* p with declaration:

$$\texttt{proc}\ p(\texttt{in}\ x_1 \in \mathbb{F}_1 \ldots;\ \texttt{out}\ x_{m+1} \in \mathbb{F}_{m+1} \ldots)\ \texttt{pre}\ \varphi\ \texttt{post}\ \psi\ \texttt{is} \ldots \texttt{begin}\ S\ \texttt{end}$$

assumes that the body of the procedure satisfies the contract, *i. e.* that (3.10) holds for φ, S and ψ. Then, by (3.17), we have

$$\begin{gathered}\Big([p(a_1, \ldots, a_m; y_{m+1}, \ldots, y_n)]\phi\Big)\sigma \;\Leftrightarrow \\ \varphi'\sigma \;\wedge\; \Big(\psi'\sigma \to \Big(\phi[{}^{z_{m+1}}/_{y_{m+1}}] \cdots [{}^{z_n}/_{y_n}]\Big)\sigma\Big)\end{gathered} \tag{3.20}$$

where $\varphi' \equiv \varphi[{}^{a_1}/_{x_1}] \cdots [{}^{a_m}/_{x_m}]$ and $\psi' \equiv \psi[{}^{a_1}/_{x_1}] \cdots [{}^{a_m}/_{x_m}][{}^{z_{m+1}}/_{x_{m+1}}] \cdots [{}^{z_n}/_{x_n}]$. Since the set of free variables in φ is contained in $\{x_1, \ldots, x_m\}$ and the set of free variables in ψ is contained in $\{x_1, \ldots, x_n\}$ and $z_{m+1}, \ldots, z_n$ are fresh, we have:

$$\begin{aligned}\varphi'\sigma &\;\Leftrightarrow\; \varphi[x_1 \mapsto a_1\sigma, \ldots, x_m \mapsto a_m\sigma] \\ \psi'\sigma &\;\Leftrightarrow\; \Big(\psi[{}^{z_{m+1}}/_{x_{m+1}}] \cdots [{}^{z_n}/_{x_n}]\Big)[x_1 \mapsto a_1\sigma, \ldots, x_m \mapsto a_m\sigma]\end{aligned}$$

Therefore, the right hand side of (3.20) is equivalent to:

$$\varphi[x_1 \mapsto a_1\sigma, \ldots, x_m \mapsto a_m\sigma] \ \wedge$$

$$\Big(\big(\psi[{}^{z_{m+1}}/_{x_{m+1}}] \cdots [{}^{z_n}/_{x_n}]\big)[x_1 \mapsto a_1\sigma, \ldots, x_m \mapsto a_m\sigma] \rightarrow \big(\phi[{}^{z_{m+1}}/_{y_{m+1}}] \cdots [{}^{z_n}/_{y_n}]\big)\sigma\Big)$$

where the fresh variables $z_{m+1}, \ldots, z_n$ represent values returned by the procedure through its `out`-mode parameters $x_{m+1}, \ldots, x_n$. Since the z_is are implicitly universally quantified, we may specialise them to the operationally defined values $z'_i = x_i\Big(\llbracket S \rrbracket^{SOS}\ [x_1 \mapsto a_1\sigma, \ldots, x_m \mapsto a_m\sigma]\Big)$, for $i \in \{m+1, \ldots, n\}$, in:

$$\varphi[x_1 \mapsto a_1\sigma, \ldots, x_m \mapsto a_m\sigma] \ \wedge$$

$$\Big(\big(\psi[{}^{z'_{m+1}}/_{x_{m+1}}] \cdots [{}^{z'_n}/_{x_n}]\big)[x_1 \mapsto a_1\sigma, \ldots, x_m \mapsto a_m\sigma] \rightarrow \big(\phi[{}^{z'_{m+1}}/_{y_{m+1}}] \cdots [{}^{z'_n}/_{y_n}]\big)\sigma\Big)$$

Because S may not assign to the `in`-mode parameters $x_1, \ldots, x_m$ of the procedure, the above is equivalent to:

$$\varphi\sigma' \ \wedge \ \Big(\psi(\llbracket S \rrbracket^{SOS}\ \sigma') \rightarrow \phi\sigma''\Big)$$

where

$$\begin{aligned} \sigma' &= [x_1 \mapsto a_1\sigma, \ldots, x_m \mapsto a_m\sigma] \\ \sigma'' &= \sigma[y_{m+1} \mapsto x_{m+1}(\llbracket S \rrbracket^{SOS}\ \sigma'), \ldots, y_n \mapsto x_n(\llbracket S \rrbracket^{SOS}\ \sigma')] \end{aligned}$$

Since we assumed $\forall\sigma\,.\,\varphi\sigma \rightarrow \psi(\llbracket S \rrbracket^{SOS}\ \sigma)$, then in particular $\varphi\sigma' \rightarrow \psi(\llbracket S \rrbracket^{SOS}\ \sigma')$:

$$\varphi\sigma' \ \wedge \ \Big(\psi(\llbracket S \rrbracket^{SOS}\ \sigma') \rightarrow \phi\sigma''\Big) \ \wedge \ \Big(\varphi\sigma' \rightarrow \psi(\llbracket S \rrbracket^{SOS}\ \sigma')\Big) \ \Rightarrow$$

$$\psi(\llbracket S \rrbracket^{SOS}\ \sigma') \ \wedge \ \Big(\psi(\llbracket S \rrbracket^{SOS}\ \sigma') \rightarrow \phi\sigma''\Big) \ \Rightarrow$$

$$\phi\sigma'' \Leftrightarrow$$

$$\phi\Big(\llbracket p(a_1, \ldots, a_m; y_{m+1}, \ldots, y_n) \rrbracket^{SOS}\ \sigma\Big)$$

by (3.8), which concludes the last case of the induction.

We have proven:

$$\big([S\,]\phi\big)\sigma \rightarrow \phi\Big(\llbracket S \rrbracket^{SOS}\ \sigma\Big)$$

for all $\mathbb{R}$-states σ, formulas ϕ and terminating statements S in PEA, which concludes the proof of the theorem. □

3.6 Referential transparency

In the introduction we mentioned as one of our contributions the treatment of SPARK Ada as a *referentially transparent language*. The notion of referential transparency is usually encountered in the context of functional languages so we need to specify what we mean by a referentially transparent *imperative* language. In the sections below we give the definition as formalised in [70] for purely applicative languages, *i. e.* languages consisting entirely of expressions and lacking side-effects, and then lift the definition to side-effecting languages containing statements.

3.6.1 Pure and Purple expressions

In [70] Søndergaard and Sestoft give a detailed analysis of referential transparency and other related *substitution principles* and show how a simple purely applicative model language may be modified to possess all, some or none of these properties. The definitions given below are slightly simplified compared to those given by the authors, but they yield an equivalent notion of referential transparency.

Given syntactic categories $\boldsymbol{Term}$ of *terminal symbols* t, $\boldsymbol{Oper}$ of *operator symbols* Ω, and $\boldsymbol{Expr}$ of *expression symbols* e, the syntax of the applicative language is given by:

$$e \quad ::= \quad t \mid \Omega\ e_1\ \ldots\ e_n$$

The syntax is interpreted by means of a *denotational semantics* $D[\![\cdot]\!] : \boldsymbol{Expr} \to \mathcal{D}$, where $\mathcal{D}$ is the set of *denotations*. The *identity* $\overset{D}{=}$ of expressions is defined by *denotational equivalence*:

$$e_1 \overset{D}{=} e_2 \quad \Leftrightarrow \quad D[\![e_1]\!] = D[\![e_2]\!]$$

A *position* p is defined as a *sequence of natural numbers*, expressed in the grammar:

$$p \quad ::= \quad \varepsilon \mid i \cdot p$$

where ε denotes the empty sequence, "$\cdot$" denotes the sequence constructor and $i \in \mathbb{N}$. The partial operation $(e, e', p) \mapsto e[{}^{e'}/_{p}]$ of *inserting expression* e' *at position* p *in expression* e is defined by:

$$e[{}^{e'}/_{\varepsilon}] \quad ::= \quad e'$$

$$(\Omega\ e_1\ \ldots\ e_i\ \ldots\ e_n)[{}^{e'}/_{i \cdot p}] \quad ::= \quad (\Omega\ e_1\ \ldots\ e_i[{}^{e'}/_{p}]\ \ldots\ e_n)$$

and *undefined* otherwise.

A position may thus be viewed as the *address* of a node in the *parsing tree* for an expression, and the insertion operation may be visualised as replacing the subtree at position p in the tree for e with the tree for e'.

Definition 3.6.1 (Purely referential position)**.** *An expression e is* purely referential *in position p if and only if*

$$e_1 \stackrel{D}{=} e_2 \quad \Rightarrow \quad e[e_1/p] \stackrel{D}{=} e[e_2/p]$$

for all $e_1, e_2 \in \boldsymbol{Expr}$.

Definition 3.6.2 (Referentially transparent/opaque operator)**.** *An operator* Ω *is* referentially transparent in place *i if and only if*

$$e_i \textit{ is purely referential in } p \quad \Rightarrow \quad \Omega\, e_1 \ \ldots\ e_i \ \ldots\ e_n \textit{ is purely referential in } i \cdot p$$

for all $e_i \in \boldsymbol{Expr}$. *Otherwise* Ω *is* referentially opaque in place *i. An operator is* referentially transparent *if it is referentially transparent in each place and* referentially opaque *otherwise.*

The applicative language $\boldsymbol{Expr}$ is then said to be referentially transparent if and only if each of its operators is referentially transparent, and otherwise said to be referentially opaque. Thus, the substitution property possessed by a referentially transparent applicative language is that replacement of denotationally equivalent expressions within an expression preserves denotational equivalence.

The problem of lifting the concept of referential transparency to a statement language has partially been addressed in unpublished work [28] by A. Daniels. The author defines referential transparency for expression sublanguage of the procedural language Purple, which in essence is an extension of WHILE with procedures and types. Procedures in Purple may be declared as *pure* and then have restrictions put on parameters similar to those imposed on PROC procedures, *i. e.* in-mode parameters are immutable and all assigned variables must be passed as out-mode parameters. Functions in Purple are restricted from accessing global variables, performing input/output (I/O), or calling impure procedures. Thanks to these restrictions, pure Purple procedures may be called from within functions without breaking referential transparency.

Daniels provides a definition of referential transparency for the expression sublanguage

of Purple, and thus generalises the definition given in [70] to typed expression languages with imperatively defined functions. Equality of expressions is parametrised by a set X of identifiers and the expressions E_1 and E_2 are deemed X-equal whenever E_1 and E_2 yield the same values for all states that map each member of X to a non-bottom value. An expression E is then called referentially transparent if and only if, expressions E_1 and E_2, that are well-typed and X-equal, may replace a given sub-expression within E to produce two expressions that are well-typed and X-equal, for every E_1 and E_2.

We see that Daniels has not completely resolved the problem, although the notion can now be applied to typed expressions defined using functions containing procedure calls, the notion of referential transparency is still confined to an expression language. Our aim is to define what a referentially transparent *statement* is.

3.6.2 PROC statements

One would usually refrain from defining referential transparency of statements in terms of an operational semantics, *i. e.* as a substitution property based on identity of statements under operational equivalence, since this would involve an explicit global context. It is often argued that the advantage of computing with pure expressions is that their evaluation abstracts from operations on memory, *i. e.* does not make explicit the allocation, update and deallocation of temporary variables used by the constituent functions, thus simplifying reasoning about such computations. This abstraction makes it useful to introduce substitution principles that formalise situations when it is possible to localise the reasoning, thus making it safe to use optimisations based on rewriting. We will define equivalence of statements in terms of a state-transformer semantics and then lift referential transparency accordingly.

The main problem we are facing is that of restricting the equivalence of statements so that a sufficiently strong notion of referential transparency is obtained. Drawing inspiration from Daniels' work we parametrise equivalence of statements by a set of variables:

Definition 3.6.3 (Restricted equivalence)**.** *Let* $S_1, S_2 \in \mathbf{Stmt}$ *and* $X \subseteq \mathbf{Var}$. *Given a state transformer semantics* $[\![\cdot]\!] : \mathbf{Stmt} \to (\mathbf{State}_{\mathbb{R}} \to \mathbf{State}_{\mathbb{R}})$ *we say that* S_1 *and* S_2 *are* X-equivalent *with respect to the semantics* $[\![\cdot]\!]$*, and write* $S_1 \overset{[\![\cdot]\!],X}{\cong} S_2$*, if and only if*

$$[\![S_1]\!]\,\sigma|_X = [\![S_2]\!]\,\sigma|_X$$

where $\sigma|_X$ *denotes the restriction of* σ *to* X*, for each* $\sigma \in \mathbf{State}_{\mathbb{R}}$*. If* $[\![\cdot]\!]$ *or* X *are obvious from the context we omit them and write* $S_1 \overset{[\cdot]}{\cong} S_2$*,* $S_1 \overset{X}{\cong} S_2$ *or simply* $S_1 \cong S_2$*. In the latter case we call* S_1 *and* S_2 equivalent*.*

Thus, restricted equivalence is simply the equivalence relation induced by the function $\lambda S.\lambda\sigma.\,[\![S]\!]\,\sigma|_X$.

Definition 3.6.4 (Strength ordering)**.** *The subset relation* $\subseteq$ *on* **Var** *induces a partial ordering on restricted equivalences. Whenever* $X \subseteq Y$*, we call* X*-equivalence* weaker *than* Y*-equivalence, and dually,* Y*-equivalence* stronger *than* X*-equivalence, for* $X, Y \subseteq$ **Var**.

So two statements equivalent under a weaker equivalence may not be equivalent under a stronger one, since the corresponding state transformations are compared over additional variables. The utility of the definitions above comes from the parametrisation of the corresponding notion of referential transparency obtained from restricted equivalences.

The constructs of the statement language **Stmt** may, in analogy with the operators of the expression language ***Expr***, be viewed as generalised operators Σ with argument types restricted by syntax. In this view, loop statements `while` b `do` S `od` correspond to expressions $\Sigma_{\texttt{while}}\ s_1\ s_2$ with $s_1 \in$ **BExp** and $s_2 \in$ **Stmt**, procedure call statements $p(a_1, \ldots, a_m; x_{m+1}, \ldots, x_n)$ correspond to expressions $\Sigma_p\ s_1\ \ldots\ s_n$ where $s_1, .., s_m \in$ **AExp** and $s_{m+1}, \ldots, s_n \in$ **Var**, *etc*. This view leads to a generalised view of expressions and with it a generalisation of *purely referential positions*:

Definition 3.6.5 (Visible variables)**.** *The function* $\mathcal{W} : \mathbf{AExp} \cup \mathbf{BExp} \cup \mathbf{Stmt} \to \wp(\mathbf{Var})$ *maps a generalised expression to its set of* visible variables*. It is defined by*

$$\mathcal{W}(\Sigma\ s_1\ \ldots\ s_n) = \bigcup_{i=1}^{n} \mathcal{W}(s_i) \quad \textit{and} \quad \mathcal{W}(t) = \begin{cases} \{t\} & \textit{if } t \in \mathbf{Var} \\ \varnothing & \textit{otherwise} \end{cases}$$

Definition 3.6.6 (Generalised purely referential position)**.** *A generalised expression* $s \in$ **AExp** $\cup$ **BExp** $\cup$ **Stmt** *is* purely referential *in position* p *if and only if*

$$s_1 \overset{W}{\cong} s_2 \quad \Rightarrow \quad s[{}^{s_1}/_p] \overset{\mathbf{Var}}{\cong} s[{}^{s_2}/_p] \tag{3.21}$$

for all $s_1, s_2 \in$ **AExp** $\cup$ **BExp** $\cup$ **Stmt** *such that* $s[{}^{s_1}/_p], s[{}^{s_2}/_p] \in$ **AExp** $\cup$ **BExp** $\cup$ **Stmt**, *with* $W = \mathcal{W}(s_1) \cup \mathcal{W}(s_2)$*.*

The definition states that a generalised expression s is purely referential in a position p if the expressions obtained by replacing sub-expressions at p, which modify their visible variables in the same way, yield the same state transformations. Having obtained a notion of pure referentiality in the context of a statement language, we can now define referentially transparent statements:

Definition 3.6.7 (Referentially transparent/opaque generalised operator)**.** *A generalised operator* Σ *is* referentially transparent in place i *if and only if*

$$s_i \text{ is purely referential in } p \;\Rightarrow\; \Sigma\ s_1\ \ldots\ s_i\ \ldots\ s_n \text{ is purely referential in } i \cdot p$$

for all $s_i \in \mathbf{AExp} \cup \mathbf{BExp} \cup \mathbf{Stmt}$ *and* $i \in \mathbb{N}$ *such that* $\Sigma\ s_1\ \ldots\ s_i\ \ldots\ s_n \in \mathbf{AExp} \cup \mathbf{BExp} \cup \mathbf{Stmt}$. *Otherwise* Σ *is* referentially opaque in place i. *A generalised operator is* referentially transparent *if it is referentially transparent in each place and* referentially opaque *otherwise.*

The *statement language* **Stmt** is then said to be *referentially transparent* if and only if each of its generalised operators is referentially transparent, and otherwise said to be referentially opaque.

It should be clear from the definition that PROC is referentially transparent. The language owes this property to the restriction that statements may only modify *visible* variables. PROC actually satisfies a much stronger property. Since functions do not modify state, and procedures may only modify variables appearing as `out`-mode parameters, the W-equivalence in (3.21) could be weakened *e.g.* by only demanding U-equivalence of substituted expressions, where U is the set of *updateable* variables, *i. e.* variables appearing on the left of assignments and as `out`-mode parameters in procedure calls. Note however that the current notion of generalised statement does not allow for the identification of operator arguments yielding updateable variables. It is an interesting question, how to design a framework for defining language syntax that is general enough to allow for a clean definition of referentiality, while offering the possibility of defining equivalences weaker than the one given by visibility.

From the discussion above we can easily see how to define referentially opaque languages. One obvious way is to lift the restriction forbidding global variables. Suppose there are two functions `f` and `g`, identically declared, except for `g` containing an additional assignment to a variable declared in the main procedure and not appearing as a formal parameter.

Clearly, `f` and `g` are W-equivalent, but when substituted yield non-identical state transformations.

4 Floating point computation

CONTENTS

In the previous chapter we saw how correctness theorems for annotated while procedures may be generated by the predicate transformer $[\cdot]$, and Theorem 3.5.1 on page 44 shows that the generated correctness theorem is a sufficient condition for operational correctness. The reductions $a2f$ and $a2p$, described in Section 3.1.2, eliminate expressions from a program, replacing them with procedure calls corresponding to the constituent functions of the reduced expressions. The operational semantics for expressions given in (3.2) on page 39 is retrieved in the denotational semantics by equipping functions with contracts defining their semantic interpretation. Clearly, it is essential that compilers, or ideally languages, give accuracy guarantees for implementations of built-in operations and numeric functions provided by standard libraries. Our approach to verification of SPARK Ada floating point properties relies on the Ada language definition providing such guarantees. In the current chapter we list the relevant parts of the IEEE-754 and IEEE-854 standards for floating point arithmetic [68, 69] and the ISO-8652:1995 Ada 95 language definition [1] showing how one may view the Ada accuracy guarantees as generalisations of the accuracy guarantees of basic operations given by the IEEE standards. In Chapter 5 we consider various ways of expressing the guarantees described in the present chapter in the model language PEA from Chapter 3.

The finitary nature of digital computation imposes two main restrictions on the implementation of numeric algorithms. *Finite execution time* means that in many cases we cannot compute the sought values completely, since such computation may involve an infinite number of steps, as in the evaluation of Taylor series or Newton's algorithm. *Finite storage space* limits the representation of computed values. We may *e.g.* wish to compute the value of a function at $1/3$ while having only base 2 and 10 formats available. In this case we have to settle for an approximate value resulting from rounding of the infinite expansion $0.01010101\ldots$ in base 2 or $0.33333333\ldots$ in base 10. Narrow ranges of rational numbers may be represented by integers scaled by a fixed representable number, so called *fixed point numbers*. In order to represent a wider range of real numbers, scaling by a variable representable number yields *floating point numbers*, which may be seen as a compromise between optimal *precision* and *size* of the representation. One may view floating point types as discretisations of bounded subsets of the real line. In this view it should not come as a surprise that the behaviour of an algorithm operating on real numbers may change considerably when executed using floating point arithmetic. One of

many accounts of the dangers of reasoning about floating point programs as if they were executed using exact real numbers is given in [56].

4.1 Floating point numbers

The main idea behind floating point numbers is based on the following identity [44]

$$x = sign(x) \cdot \frac{|x|}{\beta^{\lfloor log_\beta |x| \rfloor}} \cdot \beta^{\lfloor log_\beta |x| \rfloor} \tag{4.1}$$

which holds for any nonzero real number x and positive integer $\beta \geq 2$. It factors x into its sign, a real number between 1 and β and an integer power of β. Thus, (4.1) defines functions $s(x) = sign(x)$, $f(x) = \frac{|x|}{\beta^{\lfloor log_\beta |x| \rfloor}}$ and $e(x) = \lfloor log_\beta |x| \rfloor$, called the *sign*, *fraction* and *exponent*, such that

$$x = s(x) \cdot f(x) \cdot \beta^{e(x)} \tag{4.2}$$

In order to represent the fraction in finite memory we have to approximate $f(x)$ somehow. From the definition of f we have $1 \leq f(x) < \beta$, thus there is a sequence $\{d_i\}_{i \in \mathbb{N}}$ of *digits*, with $d_i \in \{0, \ldots, \beta - 1\}$ and $d_0 \neq 0$, such that

$$f(x) = d_0.d_1 d_2 \cdots \tag{4.3}$$

is the expansion of $f(x)$ in base β. In order to represent the number within finite memory, say using p digits, we can make the following approximation of $f(x)$

$$f_0(x) \quad = \quad \lfloor f(x) \cdot \beta^{p-1} \rfloor \cdot \beta^{1-p}$$

corresponding to truncation after p digits, or equivalently, to *rounding* $f(x)$ *towards zero* to p digits. Implementations of interval arithmetic need approximations that consequently round in the same direction. An approximation of $f(x)$ from below can be obtained by setting

$$f_{-\infty}(x) \quad = \quad s(x) \cdot \lfloor s(x) \cdot f(x) \cdot \beta^{p-1} \rfloor \cdot \beta^{1-p}$$

which corresponds to *rounding* $f(x)$ *down* to p digits. Reversing the direction, *i. e.* approximating $f(x)$ from above, can be realised as follows

$$f_{+\infty}(x) \quad = \quad s(x) \cdot \lceil s(x) \cdot f(x) \cdot \beta^{p-1} \rceil \cdot \beta^{1-p}$$

which corresponds to *rounding* $f(x)$ *up* to p digits. Denoting rounding by $\mathcal{R}$, we arrive at the following finite/storable approximations for nonzero real numbers

$$\begin{aligned} \mathcal{R}_0(x) &= s(x) \cdot f_0(x) \cdot \beta^{e(x)} \\ \mathcal{R}_{-\infty}(x) &= s(x) \cdot f_{-\infty}(x) \cdot \beta^{e(x)} \\ \mathcal{R}_{+\infty}(x) &= s(x) \cdot f_{+\infty}(x) \cdot \beta^{e(x)} \end{aligned} \tag{4.4}$$

Each floating point format is thus determined by a choice of base β, precision p and bounds $e_{\min} \leq e \leq e_{\max}$ on the exponent. The format is denoted $\mathbb{F}_{\beta,p,e_{\min},e_{\max}}$ which we abbreviate as $\mathbb{F}$ whenever possible. We have obtained a family of formats that use fixed amounts of memory to represent real numbers between $\mathbb{F}_{\min} = -(\beta-1)^p\beta^{e_{\max}}$ and $\mathbb{F}_{\max} = (\beta-1)^p\beta^{e_{\max}}$. Restricting $\mathcal{R}_0, \mathcal{R}_{-\infty}$ and $\mathcal{R}_{+\infty}$ to the range $[\mathbb{F}_{\min}, \mathbb{F}_{\max}]$ defined by a format $\mathbb{F}$ gives roundings:

$$\mathcal{R}_0^{\mathbb{F}}, \mathcal{R}_{-\infty}^{\mathbb{F}}, \mathcal{R}_{+\infty}^{\mathbb{F}} : [\mathbb{F}_{\min}, \mathbb{F}_{\max}] \to \mathbb{F} \tag{4.5}$$

taking a real number to its approximation in $\mathbb{F}$. We drop the superscript $\mathbb{F}$ whenever it is evident from the context. We shall sometimes refer to a *general rounding operator*, writing $\mathcal{R}$ with no subscript, meaning one of the operators defined in this chapter. In addition, each format defines an *absolute epsilon* value $\varepsilon_{abs} = \beta^{e_{\min}}$ and an additional choice of rounding operator defines a *relative epsilon* value ε_{rel}, defined as the smallest positive value in $\mathbb{F}$ such that $1.0 \oplus \varepsilon_{rel} \neq 1.0$, where $\oplus$ denotes rounded addition. Sect. 4.2 and 4.3 present standards for implementations of floating point arithmetic and exception handling.

4.2 The IEEE floating point standard

In order to give a uniform framework for implementations of floating point arithmetic, the IEEE-754 [68] standard for binary floating point arithmetic and the IEEE-854 [69] standard for radix independent floating point arithmetic were issued by the Institute of Electrical and Electronics Engineers. A short summary of the relevant properties of IEEE FP arithmetic are given below.

4.2.1 Standard floating point formats

A nonzero floating point number x is represented as in Sect. 4.1

$$x = s(x) \cdot f(x) \cdot \beta^{e(x)}$$

where s, f and e are defined as by 4.2 and

$$f(x) = d_0.d_1d_2 \ldots d_{p-1} \tag{4.6}$$

IEEE-754 defines *single precision* floating point numbers by

$$\beta = 2, p = 23, e_{min} = -126 \text{ and } e_{max} = 127$$

and *double precision* floating point numbers by

$$\beta = 2, p = 52, e_{min} = -1022 \text{ and } e_{max} = 1023$$

In IEEE-754 the only allowed base is 2 while IEEE-854 also allows base 10. A floating point number is *normalised* when $d_0 \neq 0$ in (4.6). Normalised numbers have unique representations. In order to handle numbers approaching zero, *denormalised* numbers are defined by $d_0 = \cdots = d_k = 0$ in (4.6) for some $0 \leq k < p - 1$ and $e = e_{min}$. This departs from our earlier presentation of floating point formats as it effectively extends the format to include values with exponent $e_{\min} - q$ and precision $p - q$, with $0 < q < p$.

4.2.2 Special floating point values

When performing a FP operation and the exponent of the resulting number is larger than e_{max} we say that the number *overflows*. Similarly, when the result of a FP operation has an exponent less than e_{min}, then we say that the number *underflows*. In order to handle overflows and underflows, the standard defines, in addition to normalised and denormalised numbers, the following special values:

- the *signed infinities* $+\infty$ and $-\infty$, resulting from overflows and division by zero
- their reciprocals, the *signed zeros* $+0$ and -0, resulting from underflows and division by an infinity

FP operations are not total even when disregarding over- and underflows. Examples of expressions with undefined values are $\pm\infty \cdot \pm 0$ and $\pm\infty + \mp\infty$. To handle such cases the standard defines an additional special value to represent undefined values:

- the *not a number* value NaN, representing undefined values resulting from the application of partial functions to values outside their domain

4.2.3 Rounding modes

In Section 4.1 we defined, for each FP format $\mathbb{F}$, rounding operations $\mathcal{R} : [\mathbb{F}_{\min}, \mathbb{F}_{\max}] \to \mathbb{F}$ taking a real number x to a FP number neighbour $\tilde{x}$. When the real number is itself a member of $\mathbb{F}$, then $x = \tilde{x}$, but whenever $x \notin \mathbb{F}$, a choice has to be made: which or the two nearest FP numbers should be chosen? In addition to the *rounding modes*: *upward rounding*, *downward rounding* and *rounding to zero*, corresponding to the rounding operators $\mathcal{R}_{+\infty}$, $\mathcal{R}_{-\infty}$ and $\mathcal{R}_0$, the IEEE standard specifies *rounding to nearest* $\mathcal{R}_\sim$, which chooses the floating point neighbour of a real number that is closer. When both neighbours are at an equal distance the one with even last digit is chosen. Rounding to nearest will, over a sufficiently large sample, result in an even distribution of upward and downward roundings, and may therefore lead to better performance of algortithms that can otherwise experience "drift" caused by consistently rounding in a fixed direction.

4.2.4 Accuracy of elementary operations

The IEEE standard demands that implementations of the arithmetic operations $+, -, \cdot, /$ and $\sqrt{}$ are *exactly rounded*. This means that the FP result is obtained by rounding the result of the corresponding real operation:

$$x \circledcirc y = \mathcal{R}(x \circ y) \tag{4.7}$$

where $\mathcal{R}$ denotes rounding in one of the modes defined above and $\circledcirc$ denotes the floating point operation corresponding to the real operation $\circ \in \{+, -, \cdot, /, \sqrt{}\}$. The precision requirements on other functions commonly found in numeric packages, such as the exponential and trigonometric functions as well as their inverses, are left to the implementation.

4.3 The ARM numeric annex

One of the goals stated in the introduction to the thesis was to propose an extension of the industrially deployed language SPARK Ada that would facilitate the verification of

functional properties of floating point code. We pointed out above that Ada is a suitable language for floating point verification, as the language definition specifies accuracy properties of the arithmetic operators and numeric functions provided by the standard libraries. Below follows a distillation of the relevant properties from the Ada 95 language reference manual.

Section G.2.4 "Accuracy Requirements for the Elementary Functions" of the Ada Reference Manual's Numerics Annex [1] gives guarantees on the accuracy of the functions found in the standard numeric packages. The main difference from the guarantees given by the IEEE standard is that for these functions, the result will not be a single point value. Instead, an interval is given, within which the result is guaranteed to lie. The interval itself is determined by the operation and the argument values. The guarantees assume that compilation is performed in a so-called *strict mode*, if this is not the case, then no guarantees on exceptional behaviour or accuracy are given by the language standard.

The standard specifies the two constants π and e and the behaviour of the built-in operations; *addition, subtraction, multiplication, division and power*, where power is the operation raising a floating point value to an integer power. In addition to the built-in operations the standard also specifies the package Ada.Numerics.Elementary_Functions, containing a number of functions corresponding to the most common functions from elementary analysis: *square root, natural logarithm, logarithm with given base, exponential function, forward and inverse trigonometric functions taking arguments in radians, or with a specified cycle*, and *hyperbolic trigonometric functions*.

Note that the *inverse tangent and cotangent* functions come in one and two parameter versions. The two argument arctangent implements the *argument function*, which generalises the inverse tangent function. The inverse cotangent function correponds to the inverse tangent function with its arguments flipped. The *argument function* takes two parameters y and x and returns the angle θ in the interval $(-\pi, \pi]$ that the vector $x + iy$ in the complex plane makes with the positive real axis. The sign of the angle is given by $\mathrm{sgn}(\theta) = \mathrm{sgn}(y)$. The restriction of the codomain corresponds to choosing the principal branch of the argument function. The one parameter versions are implemented as the corresponding two argument version with its second argument set to one.

4.3.1 Ada model of FP arithmetic

The accuracy requirements are formulated generically for floating point formats $\mathbb{F}$ in terms of *model numbers*, which are floating point numbers on the same form as the format, but without an upper bound on the exponent. The arguments and results of floating point operations and functions are assumed to lie in *model intervals*, which are intervals whose end points are model numbers. The model interval corresponding to a numeric value is the smallest model interval containing the value. The result of conversion of a numeric value to a floating point type is the model interval associated with the operand value. The result of evaluating a FP function `f` at a value v is required to belong to a model interval, called the *result interval*, containing all values of the form $(1 + d) \cdot f(x)$, where x lies in the model interval for v, $f : \mathbb{R} \to \mathbb{R}$ is the exact real function that `f` is approximating and d is a real number such that $|d|$ is less than or equal to the function's *maximum relative error*.

The standard requires that implementations of built in binary arithmetic operations $+, -, *, /$ have zero maximum relative errors. Exponentiation to integer power has accuracy corresponding to evaluation of the corresponding sequence of multiplications, allowing for arbitrary grouping, *e.g.* x^4 can be computed as $x * (x * x * x)$ or $(x * x) * (x * x)$, followed by a final division in the case of negative exponents. If division is implemented as multiplication with reciprocal in the underlying hardware, then the accuracy of division and exponentiation with negative integer exponents are implementation defined.

4.3.2 Accuracy guarantees for numeric functions

Below follow the maximum relative errors for functions in the Ada numerics package.

- `sqrt`, `sin` and `cos` have a maximum relative error of $2 \cdot \varepsilon_{rel}$
- `log`,`exp`,`tan` and `cot` have a maximum relative error of $4 \cdot \varepsilon_{rel}$
- the forward and inverse hyperbolic functions have a maximum relative error of $8 \cdot \varepsilon_{rel}$
- the maximum relative error for exponentiation with floating point exponent depends on the parameter values of the base x and exponent y, it is $\left(4 + \frac{|y \cdot log(x)|}{32}\right) \cdot \varepsilon_{rel}$

where ε_{rel} is defined in 4.1 on page 58. If we assume that all values passed to functions are finite floating point numbers in the format of their corresponding function parameters,

then the model interval of a value v contains just v itself. The guarantees then say that the result of a floating point function `f` at v is the rounding of some number in the interval $(1 + \boldsymbol{\varepsilon}_{\mathtt{f}})f(v)$, where $\boldsymbol{\varepsilon}_{\mathtt{f}} = [-\varepsilon_{\mathtt{f}}, \varepsilon_{\mathtt{f}}]$, $\varepsilon_{\mathtt{f}}$ is the maximum relative error of `f` and $f : \mathbb{R} \to \mathbb{R}$ is the exact real function that `f` approximates. Being exactly rounded is then equivalent to having zero maximum relative error. One may therefore view the statement of accuracy guarantees in the Ada language in terms of maximum relative errors as generalising the exact rounding property of IEEE floating point formats. In the next chapter we shall look further at using intervals to approximate floating point functions.

4.3.3 Ada Floating Point Exceptions

The Ada standard defines two exceptions that may be raised by incorrect use of numeric functions. Constraint_Error is raised to signal overflow and Numerics.Argument_Error is raised to signal that an argument value is passed for which the function is undefined. To guarantee predictable behaviour of numeric exceptions, compilation must be performed in strict mode and in addition, the flag Machine_Overflows must be set to true. Otherwise, the behaviour of numerical exceptions is implementation defined, *i. e.* no guarantees for the behaviour are given.

It is guaranteed that the result of a built-in floating point operation returns a value in the result interval, provided the result interval is contained within the range defined by the floating point type. Otherwise, either a value in the result interval is returned, or the error Constraint_Error is raised. The behaviour of numeric functions provided in the Numerics.Elementary_Functions package have similar behaviour to the built-in operations. If the result interval is contained in the range defined by the floating point type, then a value in the result interval is returned. Otherwise a value in the result interval is returned or the error Constraint_Error is raised, signaling overflow.

When an argument of an elementary function is outside the domain of the corresponding real function, then the error *Numerics.Argument_Error* is raised. In particular, the error is raised by:

- any *forward* or *inverse trigonometric* function with specified cycle when the value of the cycle parameter is zero or negative
- the *logarithm* function with specified base, when the value of the base parameter is

zero, one or negative

- the *square root* and *natural logarithm* functions when the argument is negative
- the *exponentiation operator* when the left operand value is negative or when both operand values are zero
- the *inverse sine and cosine* and *inverse hyperbolic tangent* functions when the absolute value of the argument exceeds one
- the *inverse tangent* and *cotangent* functions when the parameters *y* and *x* both have the value zero
- the *inverse hyperbolic cosine* function when the value of the parameter x is less than one
- the *inverse hyperbolic cotangent* function when the absolute value of the parameter *x* is less than one

Evaluation of an elementary function at a pole of the corresponding real function causes the function to raise *Constraint_Error*. The error is raised by:

- the *natural logarithm, cotangent* and *hyperbolic cotangent* functions when the argument value is zero
- the *exponentiation operator* when the left operand value is zero and the right operand value is negative
- the *tangent* function with specified cycle, when the value of the argument is an odd multiple of the quarter cycle
- the *cotangent* function with specified cycle, when the value of the argument is zero or a multiple of the half cycle
- the *inverse hyperbolic tangent* and *cotangent* functions, when the absolute value of the argument is one

When both Constraint_Error and Numerics.Argument_Error are to be raised, then the latter takes precedence.

5 Specification of floating point properties

CONTENTS

A specification for a program serves as documentation describing the intended use of the program. It is therefore important that a specification is unambiguous and succinct. Also, it is important that the specifications for subprograms combine in some intuitive way, thus making it easier to infer specifications for programs from the specifications of their subprograms. In Chapter 3 we mentioned that the semantics of floating point expressions are included in the generation of verification conditions by equipping operations and functions with contracts expressing properties of interest. In this chapter the annotation language of our model language PEA is extended to facilitate the expression of functional and exception freedom properties for floating point functions and procedures based on the accuracy guarantees in the Ada Reference Manual Numerics Annex [1]. We focus on describing *functional* properties, *i. e.* tight approximations of the actual behaviour. Expressing the strongest possible properties is intractable due to the prohibitive complexity of rounding behaviour. We may however recover some of the relative simplicity of exact real number computation by abstracting rounding through *approximation*. We find interval arithmetic to provide a suitable language for expressing such approximate properties, additionally recovering the compositional nature of exact arithmetic which supports modular reasoning about floating point computations.

5.1 Rounding operators

In Section 4.2.4 we saw that the accuracy of built-in operations can be conveniently expressed in terms of *rounding operators* $\mathcal{R}$ satisfying:

$$\forall x \,.\, \exists e \,.\, |e| \leq \max(\varepsilon_{rel}|x|, \varepsilon_{abs}) \ \wedge \ \mathcal{R}(x) = x + e \tag{5.1}$$

where ε_{abs} and ε_{rel} are the *absolute error* and *relative error* of the floating point format. ε_{abs} is defined as the smallest normalised number in the format and ε_{rel} is defined as the smallest positive number x in the format for which $1.0 \oplus x \neq 1.0$ holds. We approximate the exact rounding operators in (5.1) by the *interval valued outward rounding operator* $\mathcal{R}_{out}$ defined by:

$$\mathcal{R}_{out}(x) = (1 + \boldsymbol{\varepsilon}_{rel})x + \boldsymbol{\varepsilon}_{abs} \tag{5.2}$$

where $\boldsymbol{\varepsilon}_{abs} = [-\varepsilon_{abs}, \varepsilon_{abs}]$ and $\boldsymbol{\varepsilon}_{rel} = [-\varepsilon_{rel}, \varepsilon_{rel}]$. Clearly, we have $\mathcal{R}(x) \in \mathcal{R}_{out}(x)$ for each rounding operator $\mathcal{R} \in \{\mathcal{R}_{+\infty}, \mathcal{R}_{-\infty}, \mathcal{R}_0, \mathcal{R}_\sim\}$ defined in 4.2.3. In Section 4.3.2, the accuracy of an elementary function was expressed in terms of its *maximum relative error*. Letting

$\mathtt{f}$ be an elementary function, f the corresponding exact function on real numbers and $\varepsilon_{\mathtt{f}}$ the maximum relative error of $\mathtt{f}$, we have an analogue of (5.1):

$$\forall x \,.\, \exists d \,.\, |d| \leq \varepsilon_{\mathtt{f}} \;\wedge\; \mathtt{f}(x) = \mathcal{R}((1+d)f(x)) \tag{5.3}$$

We approximate the right hand side of the equation in (5.3) by abstracting the variable d using the *generalised outward rounding operator* $\mathcal{S}_{out}$. It is an interval valued function defined by:

$$\mathcal{S}_{out}(e, x) = \mathcal{R}_{out}\Big((1 + e[-1, 1])x\Big) \tag{5.4}$$

The trivial equality $\mathcal{R}_{out}(x) = \mathcal{S}_{out}(0, x)$ formally connects the exact rounding property of IEEE arithmetic and Ada numeric functions noted in the previous chapter. We have arrived at the following approximation for result values of floating point functions:

$$\mathtt{f}(x) \in \mathcal{S}_{out}(\varepsilon_{\mathtt{f}}, f(x)) \tag{5.5}$$

5.2 Specification of built in operations

In the following section we present formal specifications for built in operations and elementary functions derived from the corresponding specifications given in the Ada Reference Manual [1], as presented in section 4.3 above.

5.2.1 Addition, Subtraction, Multiplication and Division

We use the *membership* relation $e \in \mathbb{F}$ to succinctly express that an expression e lies within the range defined by a numeric type $\mathbb{F}$ in the contracts for floating point operations. The simplest contract for a built in operation leaves the operation uninterpreted. The precondition should restrict the arguments of the operation to the subset of the domain for which the result interval is completely contained in the range of the type. In the case of floating point addition $\oplus$ we get the following precondition:

$$\mathtt{pre}\; x \oplus y \in \mathbb{F}$$

while the postcondition simply states that the returned value is the result of the operation:

$$\mathtt{return}\; r \,.\, r = x \oplus y$$

The drawback of this approach is that we need to know how to interpret each floating point operation separately. We can simplify the situation by eliminating floating point operations from our specifications by expressing that the result lies within a rounding of the exact value that the floating point operation approximates. This can be expressed using a *general rounding* $\mathcal{R}$, representing rounding in one of the modes described in Section 4.2.3 on page 60, each satisfying (5.1), to interpret the result of the operation in terms of the corresponding exact operation on real numbers and the *equality* relation:

$$\begin{gathered} \texttt{pre}\ \mathcal{R}(x+y) \in \mathbb{F} \\ \texttt{return}\ r\ .\ r = \mathcal{R}(x+y) \end{gathered} \tag{5.6}$$

Since we have not included rounding direction in (5.6), we are making an *approximation* of the *exact* behaviour of the program. An alternative approximation uses directed rounding operators, defined in section 4.2, to express bounds on the result using *inequalities*:

$$\begin{gathered} \texttt{pre}\ \mathcal{R}_{+\infty}(x+y) \in \mathbb{F}\ \wedge\ \mathcal{R}_{-\infty}(x+y) \in \mathbb{F} \\ \texttt{return}\ r\ .\ r \leq \mathcal{R}_{+\infty}(x+y)\ \wedge\ r \geq \mathcal{R}_{-\infty}(x+y) \end{gathered} \tag{5.7}$$

The rounding operators in the specifications above may also be approximated using the *signum* operator on real numbers:

$$\texttt{return}\ r\ .\ r \leq \left(1 + \operatorname{sign}(x+y)\varepsilon_{rel}\right)(x+y) + \varepsilon_{abs}\ \wedge$$

$$r \geq \left(1 - \operatorname{sign}(x+y)\varepsilon_{rel}\right)(x+y) + \varepsilon_{abs}$$

where $\operatorname{sign}(x) = {}^{|x|}/_{x}$ for $x \neq 0$ and $\operatorname{sign}(0) = 0$. The constants ε_{abs} and ε_{rel} may of course be replaced by their corresponding values, yielding a contract expressing the value of $x \oplus y$ in terms of real numbers and standard real operations. The interval rounding operator defined above may be used to express the contracts (5.6) and (5.7) using the *membership* and *subset* relations:

$$\texttt{pre}\ \mathcal{R}_{out}(x+y) \subseteq \mathbb{F}$$

$$\texttt{return}\ r\ .\ r\ \in\ \mathcal{R}_{out}(x+y)$$

While $\mathcal{R}$ simplifies contracts by replacing the four possible rounding operators by a single one, it remains difficult to reason about. $\mathcal{R}_{out}$ simplifies things further by having a simple definition in terms of intervals.

The contracts for subtraction, multiplication and division are analogous.

Note that the division precondition analogue $\mathcal{R}({}^{x}/_{y}) \in \mathbb{F}$ of the addition precondition in

(5.6) suffices, as the rules for evaluating relations with partial expressions, given in Section 4.2.2 on page 60, imply that $\mathcal{R}(x/y) \in \mathbb{F} \Rightarrow y \neq 0$.

5.2.2 Power

The Ada standard specifies the power operator in terms of the corresponding sequence of multiplications, taken in arbitrary order, with a final division in the case of a negative exponent. For non-negative values n of the exponent the power operator satisfies the following equation:

$$\texttt{power}(x, n) = \texttt{power}(x, n_1) \circledast \texttt{power}(x, n_2)$$

where $n = n_1 + n_2$. The condition that the multiplications may be taken in arbitrary order results in a complicated contract if we try to capture the strongest possible property. We can however approximate and thus express the specification much more concisely using intervals:

$$\texttt{power}(x, n) \in (1 + \boldsymbol{\varepsilon}_{rel})^n (x^n + n\boldsymbol{\varepsilon}_{abs})$$

For negative exponents n we have:

$$\texttt{power}(x, n) = 1 \oslash \texttt{power}(x, -n)$$

We have obtained the following contract for power:

$$\begin{gathered}
\texttt{pre}\ \Big(n < 0 \rightarrow (1 + \boldsymbol{\varepsilon}_{rel})\big(\tfrac{1}{(1+\boldsymbol{\varepsilon}_{rel})^n (x^n + n\boldsymbol{\varepsilon}_{abs})}\big) + \boldsymbol{\varepsilon}_{abs} \subseteq \mathbb{F}\Big)\ \wedge \\
\Big(n \geq 0 \rightarrow (1 + \boldsymbol{\varepsilon}_{rel})^n (x^n + n\boldsymbol{\varepsilon}_{abs}) \subseteq \mathbb{F}\Big) \\
\texttt{return}\ r\ .\ \Big(n < 0 \rightarrow r \in (1 + \boldsymbol{\varepsilon}_{rel})\big(\tfrac{1}{(1+\boldsymbol{\varepsilon}_{rel})^n (x^n + n\boldsymbol{\varepsilon}_{abs})}\big) + \boldsymbol{\varepsilon}_{abs}\Big)\ \wedge \\
\Big(n \geq 0 \rightarrow r \in (1 + \boldsymbol{\varepsilon}_{rel})^n (x^n + n\boldsymbol{\varepsilon}_{abs})\Big)
\end{gathered}$$

Note that we have used the subset relation $\subseteq$ between an interval expression and the interval given by the floating point type $\mathbb{F}$. In section 5.4.3 we shall argue that interval expressions and the subset relation form an attractive basis on which to build specification languages for floating point programs.

5.3 Specification of elementary functions

The postcondition for an elementary function `f` may be expressed in terms of the generalised outward rounding operator $\mathcal{S}_{out}$ as given by (5.5) in section 5.1:

$$\mathtt{f}(x) \in \mathcal{S}_{out}(\varepsilon_{\mathtt{f}}, f(x))$$

where f is the corresponding real function and $\varepsilon_{\mathtt{f}}$ is the maximum relative error of `f`, as defined in section 4.3. The precondition needs to enforce conditions under which neither of the possible Ada exceptions will be thrown. In the sections below we show how to express this property for the square root and exponentiation operators.

5.3.1 Square root

As specified in section 4.3.1, `sqrt` is undefined for negative arguments and returns non-negative values:

$$\mathtt{pre}\ x \geq 0$$

$$\mathtt{return}\ r\ .\ r \geq 0 \wedge r \in \mathcal{S}_{out}(2\varepsilon_{rel},\ \sqrt{x})$$

5.3.2 Exponentiation

The operation of raising a floating point value to another floating point value is provided as an elementary function. In section 4.3 conditions were given under which no exceptions will be raised. Thus, the precondition must contain:

$$x \geq 0\ \wedge\ (x \neq 0 \vee y \neq 0)$$

The precondition guaranteeing no overflows and the postcondition may, as we have seen previously, be expressed using the generalised outward rounding operator:

$$\mathtt{exponential}(x, y) \in \mathcal{S}_{out}\left(\left(4 + \frac{|y \cdot log(x)|}{32}\right)\varepsilon_{rel}, x^y\right)$$

We have obtained a contract for the exponential operator:

$$\mathtt{pre}\ x \geq 0\ \wedge\ (x \neq 0 \vee y \neq 0)\ \wedge\ \mathcal{S}_{out}\left(\left(4 + \tfrac{|y \cdot log(x)|}{32}\right)\varepsilon_{rel}, x^y\right) \subseteq \mathbb{F}$$

$$\mathtt{return}\ r\ .\ r \in \mathcal{S}_{out}\left(\left(4 + \tfrac{|y \cdot log(x)|}{32}\right)\varepsilon_{rel}, x^y\right)$$

Contracts for the remaining elementary functions are readily obtained in a similar fashion, by transcribing the exception freedom conditions into the precondition and the accuracy guarantees into the postcondition, using the generalised outward rounding operator $\mathcal{S}_{out}$.

5.4 Specification of functions and procedures

In the previous sections we described the arithmetic operations and elementary functions in terms of the corresponding operations and functions on real numbers. The main difference in specifying a user defined function or procedure is that the annotation language will generally not contain the (abstract) real analogue of the implemented function. In the section below we look at one way of providing some flexibility to the annotation language, allowing users to define new functions using a *higher-order operator*, without modifying the annotation language itself.

5.4.1 Extending the annotation language with the integral operator

We extend the term sublanguage of ANOT with the integral operator:

$$\tau \ ::= \ \ldots \mid \int_{x \in [\tau_1, \tau_2]} \tau_3$$

where we use the notation $x \in [\tau_1, \tau_2]$ to indicate the integration variable and domain. We now have available all functions f of the form:

$$f(x) = f(x_0) + \int_{x_0}^{x} f'(t)dt \tag{5.8}$$

where f' is expressed by a composition of functions already in ANOT. The integral operator introduced above is a kind of specialised *lambda* constructor, turning the integrand term into a function of the integration variable. It is tempting to introduce the general lambda constructor at this point. The argument against embedding full lambda calculus within the annotation language is that lambdas, although conceptually simple, are far from universally known. One of the most important properties of an annotation language is that it needs to be *universal* in the sense that the meaning of an annotation should be evident to the intended user. Needing to inspect the code that an annotation describes defeats the role of the annotation as a specification for the program. The integral operator is part of basic education and we may therefore expect a user to immediately recognise its intended meaning. In the following section we present an example of a program that implements a function defined by (5.8) to illustrate the use of the integral operator in a concrete specification.

```
1  proc erf(in x ∈ 𝔽; out r ∈ 𝔽)
2      pre 0 ≤ x ∧ x ≤ 4
3      post 2/√π ∫_{t∈[0,x]} e^{-t^2} − 0.00005 ≤ r ∧ r ≤ 2/√π ∫_{t∈[0,x]} e^{-t^2} + 0.00005
4  is ...
5      t1, t2, t3 ∈ 𝔽
6  begin
7      t1 := 1.0 ⊕ 0.47047 ⊛ x;
8      t2 := t1 ⊛ t1;
9      t3 := t1 ⊛ t2;
10     r := 1.0 ⊖ exp(−x ⊛ x) ⊛ (0.3480242 ⊘ t1 ⊖ 0.0958798 ⊘ t2 ⊕ 0.7478556 ⊘ t3);
11 end
```

Figure 5.1: Implementation of the error function in PEA.

5.4.2 Error function example

The *error function* is often used to evaluate the cumulative density function of a normally distributed random variable. It is given by the integral:

$$\operatorname{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt$$

Implementation in PEA

Rather than compute the integral directly, our program, shown in Figure 5.1, uses a rational approximation. The main advantages, compared to implementing the function using a loop computing an approximation of the Riemann sum of the integral, is the speed of execution and that termination of our program is trivially guaranteed. According to [2], this approximation has a maximal absolute error of less than $2.5 \cdot 10^{-5}$. When taking rounding into account, the approximation's error could increase. Therefore we specified the discrepancy of our procedure to be below $5 \cdot 10^{-5}$.

Using the integral operator in a specification does decrease the readability compared to having the real error function in the specification language, it does however increase the portability, in the sense that a user may not know the error function, but may be expected to know integrals. As pointed out in the previous section, using a generic operator, such as the integral, makes the specification language smaller and more expressive.

Correctness theorem

We derive the correctness theorem for the `erf` example above, assuming the simplest form of postconditions for arithmetic operations and the exponential function:

$$\texttt{return}\ r\ .\ r = \ldots$$

using the predicate transformer $[\cdot]$, defined in Chapter 3. To make the derivation more readable we give the postcondition a short name:

$$\phi(r) = \frac{2}{\sqrt{\pi}} \int_{t\in[0,x]} e^{-t^2} - 0.00005 \leq r \ \wedge\ r \leq \frac{2}{\sqrt{\pi}} \int_{t\in[0,x]} e^{-t^2} + 0.00005$$

making dependency on the free variable r explicit, as it will be transformed during the derivation. We wish to compute:

$$\left[\begin{array}{c} t_1 := 1.0 \oplus 0.47047 \circledast x;\ t_2 := t_1 \circledast t_1;\ t_3 := t_1 \circledast t_2; \\ r := 1.0 \ominus \texttt{exp}(-x \circledast x) \circledast (0.3480242 \oslash t_1 \ominus 0.0958798 \oslash t_2 \oplus 0.7478556 \oslash t_3) \end{array}\right] \phi(r)$$

We begin by transforming composite expressions into a sequence of procedure calls by applying $f2p$ to the program and then $a2p$ to the assignment statements:

$$\left[\begin{array}{c} \oplus(1.0, 0.47047 \circledast x; t_1);\ \circledast(t_1, t_1; t_2);\ \circledast(t_1, t_2; t_3); \\ \ominus(1.0, \texttt{exp}(-x \circledast x) \circledast (0.3480242 \oslash t_1 \ominus 0.0958798 \oslash t_2 \oplus 0.7478556 \oslash t_3); r) \end{array}\right] \phi(r)$$

and again, unfolding the procedure parameters:

$$\left[\begin{array}{c} \circledast(0.47047, x; r_1);\ \oplus(1.0, r_1; t_1);\ \circledast(t_1, t_1; t_2);\ \circledast(t_1, t_2; t_3); \\ \circledast(\texttt{exp}(-x \circledast x), (0.3480242 \oslash t_1 \ominus 0.0958798 \oslash t_2) \oplus 0.7478556 \oslash t_3; r_2);\ \ominus(1.0, r_2; r) \end{array}\right] \phi(r)$$

and so on until the program body has been fully simplified:

$$\left[\begin{array}{c} \circledast(0.47047, x; r_1);\ \oplus(1.0, r_1; t_1);\ \circledast(t_1, t_1; t_2);\ \circledast(t_1, t_2; t_3); \\ \circledast(x, x; r_5);\ \texttt{exp}(-r_5; r_3);\ \oslash(0.3480242, t_1; r_7);\ \oslash(0.0958798, t_2; r_8); \\ \ominus(r_7, r_8; r_6);\ \oslash(0.7478556, t_3; r_9);\ \oplus(r_6, r_9; r_4); \\ \circledast(r_3, r_4; r_2);\ \ominus(1.0, r_2; r) \end{array}\right] \phi(r)$$

applying the procedure call rule for $[\cdot]$ to the rightmost statement:

$$\left[\begin{array}{c} \circledast(0.47047, x; r_1);\ \oplus(1.0, r_1; t_1);\ \circledast(t_1, t_1; t_2);\ \circledast(t_1, t_2; t_3); \\ \circledast(x, x; r_5);\ \texttt{exp}(-r_5; r_3);\ \oslash(0.3480242, t_1; r_7);\ \oslash(0.0958798, t_2; r_8); \\ \ominus(r_7, r_8; r_6);\ \oslash(0.7478556, t_3; r_9);\ \oplus(r_6, r_9; r_4);\ \circledast(r_3, r_4; r_2) \end{array}\right]$$
$$\Big(\ 1.0 \ominus r_2 \in \mathbb{F} \wedge (r' = 1.0 \ominus r_2 \rightarrow \phi(r')) \ \Big)$$

and again:

$$\left[\begin{array}{c} \circledast(0.47047, x; r_1);\ \oplus(1.0, r_1; t_1);\ \circledast(t_1, t_1; t_2);\ \circledast(t_1, t_2; t_3); \\ \circledast(x, x; r_5);\ \mathtt{exp}(-r_5; r_3);\ \oslash(0.3480242, t_1; r_7);\ \oslash(0.0958798, t_2; r_8); \\ \ominus(r_7, r_8; r_6);\ \oslash(0.7478556, t_3; r_9);\ \oplus(r_6, r_9; r_4) \end{array}\right]$$
$$\left(\begin{array}{c} r_3 \circledast r_4 \in \mathbb{F} \wedge \left(r_2' = r_3 \circledast r_4 \rightarrow \right. \\ \left. 1.0 \ominus r_2' \in \mathbb{F} \wedge (r' = 1.0 \ominus r_2' \rightarrow \phi(r'))\right) \end{array}\right)$$

and again:

$$\left[\begin{array}{c} \circledast(0.47047, x; r_1);\ \oplus(1.0, r_1; t_1);\ \circledast(t_1, t_1; t_2);\ \circledast(t_1, t_2; t_3); \\ \circledast(x, x; r_5);\ \mathtt{exp}(-r_5; r_3);\ \oslash(0.3480242, t_1; r_7);\ \oslash(0.0958798, t_2; r_8); \\ \ominus(r_7, r_8; r_6);\ \oslash(0.7478556, t_3; r_9) \end{array}\right]$$
$$\left(\begin{array}{c} r_6 \oplus r_9 \in \mathbb{F} \wedge \left(r_4' = r_6 \oplus r_9 \rightarrow \right. \\ r_3 \circledast r_4' \in \mathbb{F} \wedge \left(r_2' = r_3 \circledast r_4' \rightarrow \right. \\ \left. \left. 1.0 \ominus r_2' \in \mathbb{F} \wedge (r' = 1.0 \ominus r_2' \rightarrow \phi(r'))\right)\right) \end{array}\right)$$

and so on, until we have reduced all statements to one formula:

$$\begin{array}{c} 0.47047 \circledast x \in \mathbb{F} \wedge \left(r_1' = 0.47047 \circledast x \rightarrow \right. \\ 1.0 \oplus r_1' \in \mathbb{F} \wedge \left(t_1' = 1.0 \oplus r_1' \rightarrow \right. \\ t_1' \circledast t_1' \in \mathbb{F} \wedge \left(t_2' = t_1' \circledast t_1' \rightarrow \right. \\ t_1' \circledast t_2' \in \mathbb{F} \wedge \left(t_3' = t_1' \circledast t_2' \rightarrow \right. \\ x \circledast x \in \mathbb{F} \wedge \left(r_5' = x \circledast x \rightarrow \right. \\ \mathtt{exp}(-r_5') \in \mathbb{F} \wedge \left(r_3' = \mathtt{exp}(-r_5') \rightarrow \right. \\ 0.3480242 \oslash t_1' \in \mathbb{F} \wedge \left(r_7' = 0.3480242 \oslash t_1' \rightarrow \right. \\ 0.0958798 \oslash t_2' \in \mathbb{F} \wedge \left(r_8' = 0.0958798 \oslash t_2' \rightarrow \right. \\ r_7' \ominus r_8' \in \mathbb{F} \wedge \left(r_6' = r_7' \ominus r_8' \rightarrow \right. \\ 0.7478556 \oslash t_3' \in \mathbb{F} \wedge \left(r_9' = 0.7478556 \oslash t_3' \rightarrow \right. \\ r_6' \oplus r_9' \in \mathbb{F} \wedge \left(r_4' = r_6' \oplus r_9' \rightarrow \right. \\ r_3' \circledast r_4' \in \mathbb{F} \wedge \left(r_2' = r_3' \circledast r_4' \rightarrow \right. \\ \left. 1.0 \ominus r_2' \in \mathbb{F} \wedge (r' = 1.0 \ominus r_2' \rightarrow \phi(r'))\right) \cdots \Big) \end{array}$$

corresponding to thirteen precondition checks, one for each rounded floating point operation, and one VC for the unique path through the program.

5.4.3 Extending the annotation language with intervals

The example in the previous section shows how adding a higher order operator to the term sublanguage of the annotation language makes the language *extensible*, which makes it *smaller* and more *expressive*. We now turn our attention to another important property of specifications; namely *composability*. The postcondition of the `erf` procedure expresses bounds on the returned value in terms of inequalities. The advantage to using inequalities is that their meaning is completely unambiguous, they do however have a drawback in that they do not compose easily. In the following section we elaborate on this, making an argument for another extension of the annotation language, by *interval* terms and relations.

As pointed out earlier in the chapter, bounds on the return values of a function `f` may be expressed in terms of the corresponding function f on real numbers and two inequalities:

$$\begin{cases} \mathtt{f}(x) \geq \mathcal{R}_{-\infty}\Big((1 - \text{sign}(f(x))\varepsilon_{\mathtt{f}})\, f(x)\Big) \\ \mathtt{f}(x) \leq \mathcal{R}_{+\infty}\Big((1 + \text{sign}(f(x))\varepsilon_{\mathtt{f}})\, f(x)\Big) \end{cases} \tag{5.9}$$

The main problem with (5.9) is that it de-couples the bounds on $\mathtt{f}(x)$, making it difficult to reason with such constraints. As an example, consider the program in Fig. 5.2 , computing the composition $\mathtt{sin}(\mathtt{cos}(x))$. Formulating the accuracy guarantees for `sin` and `cos` as in (5.9), we obtain the following information about resulting value r:

$$\begin{aligned} y &\leq \mathcal{R}_{+\infty}\Big(\Big(1 + \text{sign}\,(\cos(x))2\varepsilon_{rel}\Big)\cos(x)\Big) \\ y &\geq \mathcal{R}_{-\infty}\Big(\Big(1 - \text{sign}\,(\cos(x))2\varepsilon_{rel}\Big)\cos(x)\Big) \\ r &\leq \mathcal{R}_{+\infty}\Big(\Big(1 + \text{sign}\,(\sin(y))2\varepsilon_{rel}\Big)\sin(y)\Big) \\ r &\geq \mathcal{R}_{-\infty}\Big(\Big(1 - \text{sign}\,(\sin(y))2\varepsilon_{rel}\Big)\sin(y)\Big) \end{aligned} \tag{5.10}$$

To mechanically eliminate the intermediate variable y — that is to derive bounds for r that are functions of x — is not trivial in cases where the functions are not monotonic. In such cases the difficulty lies in determining which of the bounds should be used. In contrast, (5.11) shows the equivalent information expressed using intervals:

$$\begin{aligned} y &\in \mathcal{R}_{out}\Big((1 + 2\boldsymbol{\varepsilon}_{rel})\cos(x)\Big) \\ r &\in \mathcal{R}_{out}\Big((1 + 2\boldsymbol{\varepsilon}_{rel})\sin(y)\Big) \end{aligned} \tag{5.11}$$

Note the use in (5.11) of the interval $\boldsymbol{\varepsilon}_{rel} = [-\varepsilon_{rel}, \varepsilon_{rel}]$ corresponding to $\varepsilon_{rel} \in \mathbb{F}$. Since we know that y lies in a given interval, and that the interval for r is given in terms of y, we

```
1 y := cos(x);
2 r := sin(y);
```

Figure 5.2: Example of function composition.

may replace y in the constraint to obtain a constraint on r involving only x. It follows that the two constraints in (5.11) compose to:

$$r \in \mathcal{R}_{out}\Big((1 + 2\boldsymbol{\varepsilon}_{rel}) \sin\Big(\mathcal{R}_{out}\big((1 + 2\boldsymbol{\varepsilon}_{rel}]) \cos(x)\big)\Big)\Big)$$

This *compositionality* of interval inclusion constraints is one of two main advantages of using these constraints, the second being *conciseness* of the resulting annotations. What may seem to be an aesthetic argument, is in fact one found in engineering, which is that a clearly formulated specification is less likely to be misinterpreted. Now that we have motivated the utility of an interval annotation language, we must formally describe how ANOT is extended with interval terms and relations. We do this by *interpreting* the terms of ANOT not as terms over real numbers, but as terms over intervals. This is done through the embedding of the real numbers in the set of real intervals, taking the real number x to the *thin* interval $\boldsymbol{x} = [x, x]$. Continuous functions $f : \mathbb{R} \to \mathbb{R}$ are lifted to functions on intervals by taking the *image* of the argument interval, which under a continuous function is also an interval, as result interval: $\boldsymbol{f}(\boldsymbol{x}) = \{f(x) \mid x \in \boldsymbol{x}\}$ *i. e.* they are lifted to their maximal extensions, as defined in Section 2.2 on page 27. We introduce intervals into the term sublanguage of ANOT by adding an interval constructor:

$$\tau \ ::= \ \ldots \mid [\tau_1, \tau_2]$$

and syntactic sugar $\boldsymbol{\varepsilon}_{abs}, \boldsymbol{\varepsilon}_{rel}$ for the intervals $[-\varepsilon_{abs}, \varepsilon_{abs}]$ and $[-\varepsilon_{rel}, \varepsilon_{rel}]$, respectively. We also extend the formula sublanguage of ANOT by the *thin inclusion relation* $\in$ and the *subset relation* $\subseteq$:

$$\varphi \ ::= \ \ldots \mid \tau_1 \in \tau_2 \mid \tau_1 \subseteq \tau_2$$

with the constraint that the left hand term τ_1 in $\tau_1 \in \tau_2$ must be a *thin* interval. Thin inclusion should be understood as *approximation* in the following sense: the approximation of a numeric expression a by an interval $\tau = [\underline{\tau}, \overline{\tau}]$ is written $a \in \tau$ and is equivalent to $\underline{\tau} \leq a \wedge a \leq \overline{\tau}$.

5.4.4 Square root example

In this section we analyse an algorithm and express it together with its properties in PEA in order to provide another example of correctness theorem derivation using the predicate transformer $[\cdot]$ and show the use of interval terms and relations in specifications. The chosen algorithm is an instance of Newton's algorithm generating a sequence of values converging to the square root of an initial non-negative real number. The inductive definition of the sequence is as follows:

$$x_{n+1} = \begin{cases} x_0 & \text{if } x_0 = 0 \\ \frac{1}{2}\left(x_n + \frac{x_0}{x_n}\right) & \text{if } x_0 > 0 \end{cases} \tag{5.12}$$

for non-negative integers n and real number x_0. It follows from (5.12) that the sequence $\{x_n\}_{n\geq 1}$ is decreasing and bounded from below by $\sqrt{x_0}$, and therefore that $\{x_n\}_{n\geq 0}$ converges to $\sqrt{x_0}$. Note however that the algorithm only converges in a finite number of steps for the initial values 0 and 1, *i. e.* when the generated sequence is constant. When executing a real number algorithm using finite precision arithmetic issues of stability may arise. An algorithm that is sensitive to perturbations of the initial conditions, or values in intermediate stages of the computation, may very well loop indefinitely due to rounding an x_n to an x_m with $m < n$. For stable algorithms finite precision arithmetic can simplify an implementation of a real number algorithm such as (5.12), since the algorithm effectively converges in a finite number of steps due to each x_n rounding to the fixpoint, for sufficiently large n.

Note that we target *partial* rather than *total correctness*, and will therefore not specify termination criteria in the annotations. Instead, we *assume* termination, and seek to specify *accuracy* properties of the returned value. Reasoning about termination for WHILE programs with floating point expressions appearing in exit conditions is a hard problem. The difficulty comes from the necessity of determinig the exact value of an expression. The problem becomes evident when considering the implementation of the square root function in Figure 5.3, where the exit condition is given by an equality relation. Here, no approximation may be made, as the equality would immediatly become undecidable. Instead, symbolic techniques must be employed, effectively tracking the *exact* value the program computes. The identification of classes of programs for which termination proofs may be tractable is an interesting and important research direction. Progress in this aera could make it safe to employ natural expressions of algoritms, such as the one in Fig-

```
proc square_root(in x ∈ 𝔽; out r ∈ 𝔽)
    pre   x ∈ [0, 1]
    post  r ∈ (1 + 4ε_rel) √x + ε_abs
is  ...
    s ∈ 𝔽
begin
    r := x;
    s := 0.0;
    while r ≠ s assert r > 0 do
        s := r;
        r := 0.5 ⊛ (s ⊕ (x ⊘ s))
    od
end
```

Figure 5.3: Example implementation of the square root function in PEA specified using standard real functions and relations.

ure 5.3, in safety critical applications.

Implementation in PEA

Fig. 5.3 shows the code for an implementation of the above algorithm in PEA. It is a variation on the general fixpoint-finding program `two` listed in Fig. 3.1, modified not to recompute previously computed x_n when evaluating the exit condition.

Correctness theorem

We derive the correctness theorem for `square_root` by applying the formalism developed in Chapter 3. We wish to compute:

$$\left[\begin{array}{l} r := x;\ s := 0.0;\ \texttt{while}\ r \neq s\ \texttt{assert}\ r > 0\ \wedge\ r \leq 1 \\ \qquad\qquad \texttt{do}\ s := r;\ r := 0.5 \circledast (s \oplus (x \oslash s))\ \texttt{od} \end{array}\right] r \in (1 + 4\varepsilon_{rel})\sqrt{x} + \varepsilon_{abs}$$

which, by (3.16), is equivalent with:

$$[r := x;\ s := 0.0]\ (r \neq s \rightarrow r > 0)\ \wedge\ (r = s \rightarrow r \in (1 + 4\varepsilon_{rel})\sqrt{x} + \varepsilon_{abs})\ \wedge\ \Big\langle r > 0 \rightarrow$$
$$[s := r;\ r := 0.5 \circledast (s \oplus (x \oslash s))]\ (r \neq s \rightarrow r > 0)\ \wedge\ (r = s \rightarrow r \in (1 + 4\varepsilon_{rel})\sqrt{x} + \varepsilon_{abs})\Big\rangle$$

We rewrite the assignment using the operators $\circledast$, $\oplus$ and $\oslash$ as a sequence of procedure calls by applying $f2p$ and $a2p$:

$$[r := x;\ s := 0.0]\ (r \neq s \to r > 0)\ \wedge\ (r = s \to r \in (1 + 4\boldsymbol{\varepsilon}_{rel})\sqrt{x} + \boldsymbol{\varepsilon}_{abs})\ \wedge \left\langle r > 0 \to \left[\begin{array}{c} s := r;\ \oslash(x, s; r_1); \\ \oplus(s, r_1; r_2);\ \circledast(0.5, r_2, r) \end{array}\right] (r \neq s \to r > 0)\ \wedge\ (r = s \to r \in (1 + 4\boldsymbol{\varepsilon}_{rel})\sqrt{x} + \boldsymbol{\varepsilon}_{abs}) \right\rangle$$

and apply (3.17) to the call to $\circledast$, introducing the fresh variable r' to represent the returned value:

$$[r := x;\ s := 0.0]\ (r \neq s \to r > 0)\ \wedge\ (r = s \to r \in (1 + 4\boldsymbol{\varepsilon}_{rel})\sqrt{x} + \boldsymbol{\varepsilon}_{abs})\ \wedge$$
$$\left\langle \begin{array}{c} r > 0 \to [s := r;\ \oslash(x, s; r_1);\ \oplus(s, r_1; r_2)]\ r_2 \circledast \frac{1}{2} \in \mathbb{F}\ \wedge \\ \left(r' = r_2 \circledast \frac{1}{2} \to (r' \neq s \to r' > 0)\ \wedge\ (r' = s \to r' \in (1 + 4\boldsymbol{\varepsilon}_{rel})\sqrt{x} + \boldsymbol{\varepsilon}_{abs})\right) \end{array} \right\rangle$$

and again, to the call to $\oplus$:

$$[r := x;\ s := 0.0]\ (r \neq s \to r > 0)\ \wedge\ (r = s \to r \in (1 + 4\boldsymbol{\varepsilon}_{rel})\sqrt{x} + \boldsymbol{\varepsilon}_{abs})\ \wedge$$
$$\left\langle \begin{array}{c} r > 0 \to [s := r;\ \oslash(x, s; r_1)]\ s \oplus r_1 \in \mathbb{F}\ \wedge\ \left(r'_2 = s \oplus r_1 \to r'_2 \circledast \frac{1}{2} \in \mathbb{F}\ \wedge\right. \\ \left.\left(r' = r'_2 \circledast \frac{1}{2} \to (r' \neq s \to r' > 0)\ \wedge\ (r' = s \to r' \in (1 + 4\boldsymbol{\varepsilon}_{rel})\sqrt{x} + \boldsymbol{\varepsilon}_{abs})\right)\right) \end{array} \right\rangle$$

and again, to the call to $\oslash$:

$$[r := x;\ s := 0.0]\ (r \neq s \to r > 0)\ \wedge\ (r = s \to r \in (1 + 4\boldsymbol{\varepsilon}_{rel})\sqrt{x} + \boldsymbol{\varepsilon}_{abs})\ \wedge$$
$$\left\langle \begin{array}{c} r > 0 \to [s := r]\ x \oslash s \in \mathbb{F}\ \wedge\ \left(r'_1 = x \oslash s \to s \oplus r'_1 \in \mathbb{F}\ \wedge\ \left(r'_2 = s \oplus r'_1 \to r'_2 \circledast \frac{1}{2} \in \mathbb{F}\ \wedge\right.\right. \\ \left.\left.\left(r' = r'_2 \circledast \frac{1}{2} \to (r' \neq s \to r' > 0)\ \wedge\ (r' = s \to r' \in (1 + 4\boldsymbol{\varepsilon}_{rel})\sqrt{x} + \boldsymbol{\varepsilon}_{abs})\right)\right)\right) \end{array} \right\rangle$$

finally, we apply (3.13) to the assignment statements:

$$(x \neq 0 \to x > 0)\ \wedge\ (x = 0 \to x \in (1 + 4\boldsymbol{\varepsilon}_{rel})\sqrt{x} + \boldsymbol{\varepsilon}_{abs})\ \wedge$$
$$\left[\begin{array}{l} r > 0 \to \left(x \oslash r \in \mathbb{F} \wedge \left(r'_1 = x \oslash r \to r \oplus r'_1 \in \mathbb{F}\ \wedge\ \left(r'_2 = r \oplus r'_1 \to r'_2 \circledast \frac{1}{2} \in \mathbb{F}\ \wedge\right.\right.\right. \\ \left.\left.\left.\left(r' = r'_2 \circledast \frac{1}{2} \to (r' \neq r \to r' > 0)\ \wedge\ (r' = r \to r' \in (1 + 4\boldsymbol{\varepsilon}_{rel})\sqrt{x} + \boldsymbol{\varepsilon}_{abs})\right)\right)\right)\right) \end{array} \right]$$

We have obtained the postcondition part of the correctness theorem. By including the precondition we have the full CT for `square_root`:

$$x \in [0, 1] \to (x \neq 0 \to x > 0)\ \wedge\ (x = 0 \to x \in (1 + 4\boldsymbol{\varepsilon}_{rel})\sqrt{x} + \boldsymbol{\varepsilon}_{abs})\ \wedge$$
$$\left[\begin{array}{l} r > 0 \to \left(x \oslash r \in \mathbb{F} \wedge \left(r'_1 = x \oslash r \to r \oplus r'_1 \in \mathbb{F}\ \wedge\ \left(r'_2 = r \oplus r'_1 \to r'_2 \circledast \frac{1}{2} \in \mathbb{F}\ \wedge\right.\right.\right. \\ \left.\left.\left.\left(r' = r'_2 \circledast \frac{1}{2} \to (r' \neq r \to r' > 0)\ \wedge\ (r' = r \to r' \in (1 + 4\boldsymbol{\varepsilon}_{rel})\sqrt{x} + \boldsymbol{\varepsilon}_{abs})\right)\right)\right)\right) \end{array} \right]$$

which may be re-written by judicious application of $\phi_1 \to (\phi_2 \to \phi_3) \Leftrightarrow \phi_1 \wedge \phi_2 \to \phi_3$:

$$\begin{aligned}
x \in [0,1] \;\to\; (x \neq 0 \;&\to\; x > 0) \wedge \\
(x = 0 \;&\to\; x \in (1 + 4\boldsymbol{\varepsilon}_{rel})\sqrt{x} + \boldsymbol{\varepsilon}_{abs}) \wedge \\
(r > 0 \;&\to\; x \oslash r \in \mathbb{F}) \wedge \\
(r > 0 \wedge r'_1 = x \oslash r \;&\to\; r \oplus r'_1 \in \mathbb{F}) \wedge \\
(r > 0 \wedge r'_1 = x \oslash r \wedge r'_2 = r \oplus r'_1 \;&\to\; r'_2 \circledast \tfrac{1}{2} \in \mathbb{F}) \wedge \\
(r > 0 \wedge r'_1 = x \oslash r \wedge r'_2 = r \oplus r'_1 \wedge r' = r'_2 \circledast \tfrac{1}{2} \wedge r' \neq r \;&\to\; r' > 0) \wedge \\
(r > 0 \wedge r'_1 = x \oslash r \wedge r'_2 = r \oplus r'_1 \wedge r' = r'_2 \circledast \tfrac{1}{2} \wedge r' = r \;&\to\; r' \in (1 + 4\boldsymbol{\varepsilon}_{rel})\sqrt{x} + \boldsymbol{\varepsilon}_{abs})
\end{aligned}$$

5.5 Concluding remarks

There is no simple set of rules for finding an appropriate specification for a program. In the field of information engineering heuristics such as *requirements elicitation* help implementers to formalise essential properties that users of a program expect to hold. Such heuristics can help to focus the specification design task by fixing a language in which the specification is to be expressed. Also the method by which the validation of of proof obligations is conducted can help to further constrain the expression of specifications by eliminating specifications that lead to correctness theorems that carry an unacceptable proof effort.

In the case of *floating point* properties, *i. e*. accuracy and absence of numerical exceptions, the first question must be how the program will be used. If the program is the top level procedure, then the specification may be relaxed maximally, as long as it still satisfies the requirements of the end users. If, however, the program is to be used deep within other programs, then it becomes essential to provide as tight properties as possible, since any approximation will carry an information loss cost similar to that of interval arithmetic discussed previously. Successive approximations magnify the initial information loss possibly leading to only unacceptably weak properties being provable at the top level.

It is questionable if numerically intensive subprograms can be proved using automatic methods. Library programs intended as subroutines may have to be equipped with properties that can only be proved using interactive symbolic provers. However, if only weak properties, such as exception freedom or loose error bounds, are sought then the type

of specifications described in this chapter may suffice and yield correctness theorems amenable to automated proof.

The exact choice of specification must thus take into consideration both the intended users of the program, and the intended method for discharging the generated proof obligations. The process of finding an appropriate expression may consist of a loop starting with a choice of specification, followed by correctness theorem generation, and ending in a proof attempt. If the proof attempt fails a new specification is chosen, and the loop is traversed again. It is not clear if this method scales. The complexity in program size is similar to that for backtracking exhaustive search algorithms in search space size. The method is however the one often used in practice and it is questionable if a better one currently exists. With advances in automated specification generation, such as the generation of specifications for sub-programs from top level specifications, the problem of predicting the necessary strength of subprogram specifications may be eliminated, but currently it remains open.

In the particular case of the two example programs `erf` and `sqrt` the contracts were found by using interval approximations, as described in this chapter, which led to fairly tight accuracy properties. In the case of `erf` it was guessed that doubling of the error bound given in literature [2] would suffice. The guess was based on the error bound for the approximation of the integral form with the rational function provided by [2], and a straightforward computation of the maximal thickness of the interval approximation of the rational function over the interval $[0, 4]$. The interval constraint for `sqrt` was found in a similar way, by estimation of the maximal width of the interval approximation of the expression for the returned value at the fixpoint. In both cases the compositional nature of interval approximations made it possible to derive the error bounds with relative ease.

6 Implementation in SPARK

CONTENTS

SPARK Ada was given in the introduction to the thesis as an example of a programming language deployed industrially in the high integrity applications area. The SPARK Ada approach is based on *subsetting* of Ada to a core language by eliminating parts of the full language that are difficult to formally reason about. Thus, *e.g*. recursion and pointers are eliminated in the SPARK Ada core language. Restrictions are also imposed that guarantee predictable space usage of programs. This is particularly important for applications in embedded systems, which often run on hardware with very limited memory.

One of our stated goals has been to propose an extension to the SPARK Ada annotation language that would introduce capabilities for expressing and verifying functional properties of floating point code. This chapter presents one such extension, based on the work presented in the previous chapters, formulated by presenting a SPARK Ada version of the Ada.Numerics.Elementary_Functions package and SPARK Ada implementations of the example programs from Chapter 5.

6.1 The SPARK Ada language

The SPARK Ada *language* consists of a sublanguage of Ada equipped with an annotation language equivalent to first order predicate logic over the real numbers. The SPARK Ada *framework* consist of the language and a tool suite comprising the Examiner verification condition generator, the Simplifier automated theorem prover and an interactive theorem prover called the Proof Checker. Currently, SPARK Ada has not fully supported verification of floating point programs and consequently, the tools lack proper support for handling floating point property annotations and verification conditions arising from floating point programs.

6.1.1 Syntax of SPARK annotations

The syntax and semantics of SPARK Ada are well documented in [10]. An annotation is a special comment denoted by an additional symbol following the usual Ada comment declaration, with the default being `--#`. As a result, the operational semantics of SPARK Ada is simply that of the Ada core language.

Imported packages are declared using the `inherit` annotation and main programs are declared using the `main_program` annotation. The `derives` annotation enables the Ex-

aminer to perform data and information flow analysis. In subprograms, the `global` annotation declares variables of the main program the subprogram interacts with. As mentioned in Chapter 1, the declaration of global variables makes SPARK Ada *referentially transparent* as an imperative language, facilitating analysis by making all effects explicit through variables declared in the specification of a program.

- `inherit` declares the import of a package.
- `main_program` declares a procedure as top level program.
- `derives` declares for each output variable which input variables its new value depends on.
- `global` declares variables of a program that a subprogram reads from or writes to.

Functional properties of the program are asserted using invariants. These may be declared using the keywords

- `pre` declaring the precondition, a predicate on the input values;
- `post` declaring the postcondition, a predicate on the output values;
- `assert` declaring a predicate on the program variables at some point inside the subprogram.

The primitives above are used to annotate executable code with predicates allowing for the generation of correctness theorems. The annotation language is extensible by *abstract functions* which are declared in the same way as Ada functions but within annotations. They may then be used within subsequent annotations and are left uninterpreted by the Examiner. It is possible to provide rules for abstract functions to be used by the theorem provers in the toolset, it is however difficult to predict when and what rules the Simplifier will apply. In our case we prefer the abstract functions to remain uninterpreted by the SPARK Ada toolset, using them to represent new language elements.

6.1.2 An example program

As an illustration of the syntax of SPARK Ada we give the code of the example program `Twice` in Figure 6.1. The precondition assures that the doubled value stays within the range defined by the `Integer` type, while the postcondition states that the output value is the double of the input value.

```
procedure Twice (X : in Integer; R : out Integer)
--# derives R from X;
--# pre  2*X in Integer;
--# post R = 2*X;
is begin
    R := X+X;
end Twice;
```

Figure 6.1: Example program specification

6.2 The SPARK tool set

As mentioned above, the SPARK Ada language comes with a tool set comprising a verification condition generator and automated and interactive theorem provers.

6.2.1 The Examiner

The SPARK Examiner is an analyser tool that produces correctness theorems for SPARK Ada programs. Each theorem is a conjunction of verification conditions (VCs) corresponding to execution paths in the program. The VCs are obtained through a Hoare-type backward analysis called *hoisting*. One equips the program code with invariants (i.e. pre- and postconditions and loop cuts) in the form of comments and the invariants are propagated backwards (i.e. hoisted) between program points. The resulting VCs are of the form

$$H_1 \wedge \cdots \wedge H_n \rightarrow C_1 \wedge \cdots \wedge C_m$$

where the H_i's are called hypotheses and the C_j's are called conclusions.

Hoisting over Assignments.

The analysis is performed by the Examiner through hoisting. It entails propagating an invariant back along an execution path. When passing an assignment of an expression to a variable each occurrence of the variable in the invariant is replaced by the expression, *e.g.* when hoisting `R=2*X` over `R:=X+X`, it becomes `X+X=2*X`.

Hoisting over Procedure Calls.

When hoisting an invariant past a procedure call the implicit assumption is made that the procedure is correct. Thus the precondition of the procedure holds at the point of call and its out parameters satisfy the procedure's postcondition. Procedure calls are thus modelled by:

- an assignment to each of the procedure's out parameters,
- a requirement that the precondition holds at the calling point and
- an assumption that the postcondition holds below the calling point.

The requirement generates an additional VC in the correctness theorem.

Modular Analysis.

The SPARK analysis is modular, in the sense that subprograms need not be re-proved. This means that when changes are made to a program that leave subprograms unchanged, then only the VCs for the changed code will need to be proved.

Current Treatment of Floating Point Computation.

Presently the Examiner treats floating point numbers and operations as exact real numbers and functions. This is clearly dangerous and there is a warning given with each correctness theorem, stating that the analysis is approximate.

6.2.2 The Simplifier

The SPARK tool set contains an automated theorem prover called the Simplifier. It simplifies VCs by eliminating redundant hypotheses and transforming conclusions. It often manages to reduce conclusions or whole VCs to trivial tautologies and removes them.

6.2.3 The Proof Checker

The Proof Checker is an interactive proof assistant which is used to obtain mechanically checkable proofs of verification conditions that the Simplifier cannot discharge. Once a proof has been performed, it may be re-run automatically, thus providing added confidence in the correctness proof.

6.3 SPARK.Numerics

In this section we present an extension to the SPARK Ada annotation language with the integral operator. The method has been described in depth in the previous chapters so we will only quickly review it and refer back to Chapters 3, 4 and 5 for further details.

Our approach to verification condition generation for SPARK floating point programs is to equip all floating point operations, functions and programs with the appropriate *specifications* with preconditions ensuring exception freedom and postconditions specifying functional properties. We first begin by defining abstract functions representing the mathematical analogues of the elementary functions we shall use, along with new functions implementing the integral operator and interval arithmetic extensions to the annotation language, as described in Chapter 5.

6.3.1 `Erf` example

In this section we present a SPARK Ada implementation of the `erf` example program from Section 5.4.2. We provide abstract functions for elementary operations and annotation language extensions described in Section 5.4.1 in the package `Exact`, shown in Figure 6.2.

```
package Exact is

  --# function Pi return Float;
  --# function Sqrt (X : Float) return Float;
  --# function Exp (X : Float) return Float;
  --# function FreshVariable return Float;
  --# function Integral (A,B,Integrand,Variable : Float) return Float;

end Exact;
```

Figure 6.2: `Exact` package specification fragment

Note the use of abstract functions in the package `Exact`, declaring new annotation language elements. The numeric ground types in the SPARK Ada annotation language are integers and real numbers. Thus, the type `Float` is a synonym for the reals $\mathbb{R}$ in annotations, and in particular in abstract function declarations.

In SPARK Ada there is no way to introduce new variables into annotations, other than when bound by propositional quantifiers. We circumvent this limitation by using the `FreshVariable` abstract function, providing one unique fresh variable. If more are needed we could have represented a family of fresh variables by declaring `FreshVariable` as a function taking an integer parameter, seen as the index of the represented variable. The need for additional variables is due to the introduction of the `Integral` operator, which requires an integrand expression and a variable with respect to which the integrand is to be integrated. In effect we have introduced expressions:

$$\int_a^b \tau \, dx$$

where τ is any term in FORM and x is the fresh variable. This approach is clearly unsatisfactory and we would prefer to allow a third kind of binding, other than through existential and universal quantification, which would enable the introduction of fresh variable names by users within integrals. Then, occurrences of expressions:

$$\texttt{Integral}(\texttt{A}, \texttt{B}, \texttt{T}, \texttt{X})$$

would be checked for freshness of `X` preventing accidental binding of program variables and use of arbitrary expressions for `X` and then introduce the new variable for use within `T`. Function wrappers for floating point operations and elementary functions are specified by the package `Numeric`, shown in Figure 6.3. We equip each function with a precondition guaranteeing that the result will be within the range defined by the type `Float`.

The implementation of the `Erf` program is shown in Figure 6.4. We have used the Ada notation `E in A..B` for the simultaneous inequalities $\texttt{A} \leq \texttt{E} \leq \texttt{B}$. Other than that it is a faithful implementation of the PEA program show in Figure 5.1 on page 72. Note the use of the `Numeric.Value` when introducing floating point constants. The reason is that the underlying representation of `Float` is in base 2, which means that values such as 0.1 cannot be exactly represented. Therefore, any assignment of a value that is not clearly representable must be wrapped in a function representing the rounding error that may be incurred when casting the decimal float to a binary one.

6.3.2 `Sqrt` example

In this section we present a SPARK Ada implementation of the `square_root` example program from Section 5.4.4. We provide abstract functions for elementary operations and

```
package Numeric is

  function Value (X : Float) return Float;

  function Add (X,Y : Float) return Float;
  --# pre Add(X,Y) in Float;

  function Multiply (X,Y : Float) return Float;
  --# pre Multiply(X,Y) in Float;

  function Divide (X,Y : Float) return Float;
  --# pre Divide(X,Y) in Float;

  function Power (X : Float; N : Integer) return Float;
  --# pre Power(X,N) in Float;

  function Exp (X : Float) return Float;
  --# pre Exp(X) in Float;

end Numeric;
```

Figure 6.3: `Numeric` package specification fragment

annotation language extensions described in Section 5.4.3 in the package `Exact`, shown in Figure 6.2. The relation `Ni` corresponds to the relation $\in$ on terms in ANOT and `Sqrt` corresponds to the exact square root. Procedure wrappers for floating point operations and elementary functions are provided in the package `Numeric`. They are equipped with preconditions guaranteeing that the result is in the range defined by the type `Float` and postconditions expressing bounds on the result values. Note that we have eliminated rounded operations and rounding operators from the specifications. This results in verification conditions that contain only real intervals and operations and relations on them. The abstract functions `EpsAbsi` and `EpsReli` denote the intervals $\boldsymbol{\varepsilon}_{abs} = [-\varepsilon_{abs}, \varepsilon_{abs}]$ and $\boldsymbol{\varepsilon}_{rel} = [-\varepsilon_{rel}, \varepsilon_{rel}]$ respectively. We have not needed to introduce the interval constructor $[\tau_1, \tau_2]$ as the Ada range notation is extended in SPARK to include any abstract expression as bounds. The limitation is that arithmetic of range expressions is not permitted, which limits the utility of this notation for our purposes. An extension of the `E in A..B`

notation to permit expressions such as:

```
E in (A..B)+(C..D)
```

would make it possible to write the specifications in Figure 6.6 in the following form:

```
--# R in (1.0-EpsRel*(-1.0..1.0))*(X+Y)+EpsAbs*(-1.0..1.0)
```

In such cases one would probably introduce the *unit interval* `Unit` in the `Exact` package, making the specification above slightly more readable:

```
--# R in (1.0+EpsRel*Exact.Unit)*(X+Y)+EpsAbs*Exact.Unit
```

Currently, the Examiner rewrites range memberships into the corresponding pair of inequalities, but as pointed out in Chapter 5, there are advantages to keeping them coupled. We would prefer to have the option not to eliminate the `in` relation during verification condition generation, but rather deferring it to the simplification phase.

Due to the handling of postconditions of functions in the freely available version of the Examiner we have had to use procedures to implement correct handling of floating point operations and functions. In the latest commercial version of the tool, specifications for functions are handled in a similar manner to procedures. This means that future versions of the `Numeric` package presented in this section will be implemented using function wrappers as in Section 6.3.1, but using the specification from the current section.

The SPARK implementation of the `Sqrt` example program is shown in Figure 6.7. We are using the new realtion `Ni` in place of the extended Ada range membership `in` proposed above. Note that we have had to unroll the rounded operation in the loop guard, as described in Section 3.1.2, making sure that the latest values of the program variables `R` and `S` are used when evaluating the exit condition.

```
with Exact,Numeric;
--# inherit Exact,Numeric;
--# main_program;
procedure Erf (X : in Float; R : out Float)
--# derives R from X;
--# pre  X in 0.0..4.0;
--# post R in 2.0/Exact.Sqrt(Exact.Pi)
--#           *Exact.Integral
--#              (0.0,
--#               X,
--#               Exact.Exp(-Exact.FreshVariable**2),
--#               Exact.FreshVariable)
--#           -0.00005 ..
--#           2.0/Exact.Sqrt(Exact.Pi)
--#           *Exact.Integral
--#              (0.0,
--#               X,
--#               Exact.Exp(-Exact.FreshVariable**2),
--#               Exact.FreshVariable)
--#           +0.00005;
is
  A1,A2,A3,P,T : Float;
begin
  A1 := Numeric.Value(0.3480242);
  A2 := Numeric.Value(-0.0958798);
  A3 := Numeric.Value(0.7478556);
  P := Numeric.Value(0.47047);
  T := Numeric.Add
         (1.0,Numeric.Multiply(P,X));
  R := Numeric.Add
         (1.0,-Numeric.Multiply
                 (Numeric.Exp
                    (-Numeric.Power(X,2)),
                  Numeric.Add
                    (Numeric.Add
                       (Numeric.Divide(A1,T),
                        Numeric.Divide
                          (A2,Numeric.Power(T,2))),
                     (Numeric.Divide
                       (A3,Numeric.Power(T,3))))));
end Erf;
```

Figure 6.4: `Erf` program code

```
package Exact is

  --# function Ni (X,Y : Float) return Boolean;
  --# function Sqrt (X : Float) return Float;

end Exact;
```

Figure 6.5: `Exact` package specification fragment

```
with Exact;
--# inherit Exact;
package Numeric is

  function EpsAbs return Float;

  --# function EpsAbsi return Float;

  function EpsRel return Float;

  --# function EpsReli return Float;

  procedure Assign (X : in Float; R : out Float);
  --# derives R from X;
  --# pre  (1.0+EpsReli)*X in Float;
  --# post Exact.Ni(R,(1.0+EpsReli)*X+EpsAbsi);

  procedure Add (X,Y : in Float; R : out Float);
  --# derives R from X,Y;
  --# pre  (1.0+EpsReli)*(X+Y) in Float;
  --# post Exact.Ni(R,(1.0+EpsReli)*(X+Y)+EpsAbsi);

  procedure Multiply (X,Y : in Float; R : out Float);
  --# derives R from X,Y;
  --# pre  (1.0+EpsReli)*(X*Y) in Float;
  --# post Exact.Ni(R,(1.0+EpsReli)*(X*Y)+EpsAbsi);

  procedure Divide (X,Y : in Float; R : out Float);
  --# derives R from X,Y;
  --# pre  (1.0+EpsReli)*(X/Y) in Float;
  --# post Exact.Ni(R,(1.0+EpsReli)*(X/Y)+EpsAbsi);

end Numeric;
```

Figure 6.6: `Numeric` package specification fragment

```
with Exact,Numeric;
--# inherit Exact,Numeric;
--# main_program;
procedure Sqrt (X : in Float; R : out Float)
--# derives R from X;
--# pre  X in 0.0 .. 1.0;
--# post Exact.Ni(R,(1.0+4.0*Numeric.EpsReli)*Exact.Sqrt(X));
is
  R1,R2,R3,S : Float;
begin
  R := X;
  S := 1.0;
  while R /= S loop
    --# assert R > 0.0;
    S := R;
    Numeric.Multiply(0.5,S,R1);
    Numeric.Divide(X,S,R2);
    Numeric.Multiply(0.5,R2,R3);
    Numeric.Add(R1,R3,R);
  end loop;
end Sqrt;
```

Figure 6.7: `Sqrt` program code

7 Background

CONTENTS

The present chapter provides the mathematical background needed for the discussion of correctness proof automation that follows in Chapter 8. It builds on and expands the framework of approximation in the interval poset described in Chapter 2.

7.1 Approximation continued

Sections 2.1 and 2.2 outlined a framework for approximation of elements of a poset R within an associated poset of subsets of R. We now specialise the discussion to approximation of the two point extended reals $(\mathbb{R}^{\pm\infty}, \leq)$ given in Example 2.1.13 within the associated poset $(\mathbb{I}(\mathbb{R}^{\pm\infty}), \sqsubseteq)$ of $\mathbb{R}^{\pm\infty}$- intervals with the refinement order.

7.1.1 Approximation of real numbers

Since the poset $(\mathbb{R}^{\pm\infty}, \leq)$ has a least element $-\infty$ and a greatest element $+\infty$, it follows that the interval poset $(\mathbb{I}(\mathbb{R}^{\pm\infty}), \sqsubseteq)$ has a least element $\mathbb{R}^{\pm\infty} = [-\infty, +\infty]$. The maximal elements in $\mathbb{I}(\mathbb{R}^{\pm\infty})$ are the *singletons*, *i. e.* intervals with identical endpoints. The correspondence $\mathbb{R}^{\pm\infty} \ni x \leftrightarrow [x, x] \in \mathbb{I}(\mathbb{R}^{\pm\infty})$ embeds the extended reals within the extended real intervals and we say that the interval $\boldsymbol{x}$ *approximates* the real x whenever $\boldsymbol{x} \sqsubseteq [x, x]$ holds. In such cases we extend the notation by writing $\boldsymbol{x} \sqsubseteq x$.

Definition 2.2.4 gives a canonical lifting of relations on a poset to relations on the associated interval poset. In particular, the order relation $\leq$ on $\mathbb{R}^{\pm\infty}$ lifts to the relation $\leq_{\mathbb{I}}$ on $\mathbb{I}(\mathbb{R}^{\pm\infty})$, and the induced relation *approximates* the original relation in the following sense: if $x, y \in \mathbb{R}^{\pm\infty}$ and $\boldsymbol{x}, \boldsymbol{y} \in \mathbb{I}(\mathbb{R}^{\pm\infty})$ are such that $\boldsymbol{x} \sqsubseteq x$ and $\boldsymbol{y} \sqsubseteq y$, then $\boldsymbol{x} \leq_{\mathbb{I}} \boldsymbol{y}$ implies $x \leq y$.

7.1.2 Approximation of total real functions

In the previous chapters we have modelled floating point functions by extended real functions. In this section we describe approximation of total extended real functions and relate it to approximation of extended reals described in the section above.

The set of $(\mathbb{R}^{\pm\infty})^n \to \mathbb{R}^{\pm\infty}$ of total extended real valued functions taking n extended real arguments ordered pointwise is an instance of a poset of poset-valued functions from Example 2.1.6. Since the poset $\mathbb{R}^{\pm\infty}$ is a complete lattice, it follows that $(\mathbb{R}^{\pm\infty})^n \to \mathbb{R}^{\pm\infty}$ is a complete lattice with pointwise lifted operations.

As the lattice is complete, the associated poset of intervals is a complete semilattice with operations given in Definition 2.2.7 and $[\lambda x_1 \cdots x_n.-\infty, \lambda x_1 \cdots x_n.+\infty]$ as bottom element. Members $[f, g]$ of $\mathbb{I}((\mathbb{R}^{\pm\infty})^n \to \mathbb{R}^{\pm\infty})$ are intervals with functions as endpoints, satisfying $f(x_1, \ldots, x_n) \leq g(x_1, \ldots, x_n)$ for all $(x_1, \ldots, x_n) \in (\mathbb{R}^{\pm\infty})^n$.

Functions are approximated by function intervals in the same way as reals are approximated by real intervals, *i. e.* a function interval $\boldsymbol{f}$ *approximates* the function f whenever $\boldsymbol{f} \sqsubseteq f$. The analogy with reals carries over to approximation of inequalities between functions as well: if $f, g : (\mathbb{R}^{\pm\infty})^n \to \mathbb{R}^{\pm\infty}$ and $\boldsymbol{f}, \boldsymbol{g} \in \mathbb{I}((\mathbb{R}^{\pm\infty})^n \to \mathbb{R}^{\pm\infty})$ are such that $\boldsymbol{f} \sqsubseteq f$ and $\boldsymbol{g} \sqsubseteq g$, then $\boldsymbol{f} \dot{\leq}_{\mathbb{I}} \boldsymbol{g}$ implies $f \dot{\leq} g$.

In this chapter we have so far considered *exact* computation. But real numbers, intervals and functions cannot generally be stored or computed with on a digital computer. In the following section we shall recall some concepts from our previous discussion of finitary approximations.

7.1.3 Finitary approximation of numbers

Chapter 4 presented floating point numbers as a means of representing real numbers in a way that may be effectively stored and computed with on digital systems. To obtain *safe* approximations of real numbers we approximate numbers with intervals as described above, but restrict ourselves to *floating point intervals*, *i. e.* intervals with floating point endpoints. Thus, an irrational real, such as π, is approximated by an interval $[x, y]$ whenever $[x, y] \sqsubseteq \pi$, *i. e.* by a pair of numbers x and y such that $x \leq \pi$ and $\pi \leq y$. Taking x and y to be floating point numbers, say $x = 3.1$ and $y = 3.2$, we have a finitary and safe approximation $[3.1, 3.2] \sqsubseteq \pi$ of the irrational real π.

In Section 5.1 rounding operators were described taking a real number in $\{x \in \mathbb{R} \mid \mathbb{F}_{\min} \leq x \wedge x \leq \mathbb{F}_{\max}\}$ to a floating point neighbour in $\mathbb{F}$, for some floating point format $\mathbb{F}$. We can extend the rounding operators to map from $\mathbb{R}^{\pm\infty}$ to $\mathbb{F}^{\pm\infty}$, where $\mathbb{F}^{\pm\infty}$ denotes the subset of $\mathbb{R}^{\pm\infty}$ consisting of floating point numbers in the format $\mathbb{F}$ and the infinities, by mapping numbers outside the domain of the original operator to the appropriate bound of $\mathbb{F}$ or an infinity. The *directed rounding operators* $\mathcal{R}_{-\infty}$ and $\mathcal{R}_{+\infty}$ then provide a lower bound $\underline{x}$ and an upper bound $\overline{x}$, respectively, for any extended real $x \in \mathbb{R}^{\pm\infty}$. We have obtained the following approximation of x in $\mathbb{I}(\mathbb{F}^{\pm\infty})$: $[\mathcal{R}_{-\infty}(x), \mathcal{R}_{+\infty}(x)] \sqsubseteq x$.

The approximation in the preceding paragraph is based on selecting an *effectively representable* subset $\mathbb{F}^{\pm\infty}$ of $\mathbb{R}^{\pm\infty}$ and directed rounding operations. In the remaining discussion we shall denote the result of *rounding down* an element x by $x\downarrow$ and the result of *rounding up* the element x by $x\uparrow$. We shall not explicitly distinguish roundings on different sets unless they cannot be identified from the context. With this notation we have: $\mathcal{R}_{-\infty}(x) = x\downarrow$ and $\mathcal{R}_{+\infty}(x) = x\uparrow$ in $\mathbb{R}^{\pm\infty}$. Directed roundings $\downarrow,\uparrow : \mathbb{R}^{\pm\infty} \to \mathbb{F}^{\pm\infty}$ induce the downward rounding $\downarrow : \mathbb{I}(\mathbb{R}^{\pm\infty}) \to \mathbb{I}(\mathbb{F}^{\pm\infty})$ defined in the obvious way: $[x,y]\downarrow = [x\downarrow, y\uparrow]$, and we clearly have $[x,y]\downarrow \sqsubseteq [x,y]$.

7.1.4 Finitary approximation of total functions

The notation given in the section above for roundings on numbers carries over to functions:

Example 7.1.1 (Rounding to constant)**.** *Let $f : X \to \mathbb{R}^{\pm\infty}$ be a member of the poset of extended real valued functions on a set X and define the functions $\downarrow,\uparrow : R \to \mathbb{R}^{\pm\infty}$ by:*

$$f\downarrow = \inf_{x\in X} f(x) \text{ and } f\uparrow = \sup_{x\in X} f(x)$$

taking a function to its lower or upper constant approximation. Then $\downarrow$ and $\uparrow$ are directed roundings *in the sense of Section 7.1.3 as $f\downarrow \dot{\leq} f \dot{\leq} f\uparrow$ holds for all $f : X \to \mathbb{R}^{\pm\infty}$.*

The roundings in the example above provide valid but *coarse* approximations in the sense that $f\downarrow = g\downarrow$ and $f\uparrow = g\uparrow$ may hold although $f \dot{<} g$. Another example of roundings on functions is provided by rounding of smooth functions to polynomials:

Example 7.1.2 (Rounding smooth functions to polynomials)**.** *Let $C^\infty(\mathbb{R}^{\pm\infty})$ be the set of extended real valued smooth functions on $\mathbb{R}^{\pm\infty}, f(x) = \sum_{k=0}^{\infty} \frac{f^{(k)}(0)}{k!}x^k$ be the Maclaurin expansion of $f \in C^\infty(\mathbb{R}^{\pm\infty})$ and $M_n(f) = \sum_{k=0}^{n} \frac{f^{(k)}(0)}{k!}x^k$ and $R_n(f) = \sum_{k=n+1}^{\infty} \frac{f^{(k)}(0)}{k!}x^k$ be the order n Maclaurin polynomial and rest term, respectively. We define the family of polynomial valued directed roundings $\downarrow,\uparrow : C(\mathbb{R}^{\pm\infty}) \times \mathbb{N} \to \mathbb{R}^{\pm\infty}[x]$ by*

$$f\downarrow n = M_n(f) + R_n(f)\downarrow \text{ and } f\uparrow n = M_n(f) + R_n(f)\uparrow$$

where the roundings $\downarrow$ and $\uparrow$ are the constant roundings from Example 7.1.1.

Thus $\downarrow n$ and $\uparrow n$ round smooth functions to n^{th}-degree polynomials. In effect, each polynomial rounding *refines* the corresponding constant rounding and the polynomial round-

ings for smaller degrees, with the two coinciding for zero degree polynomials, *i. e.* constants: $\downarrow 0 = \downarrow$ and $\uparrow 0 = \uparrow$.

When the function to be approximated is *bounded* over the domain in question, then we may use the subset of polynomials with extended real coefficients as codomain of the rounding operations:

Example 7.1.3 (Rounding bounded functions to polynomials)**.** *Let $K \subset \mathbb{R}$ be a compact set of reals and $C(K)$ be the set of continuous real valued functions on K. Then each member f of $C(K)$ is bounded and thus there are polynomials $p, q \in \mathbb{R}[x]$ such that $p(x) \leq f(x) \leq q(x)$ for each $x \in K$. By the Axiom of Choice we may pick two such polynomials p_f and q_f for each f in $C(K)$, thus defining roundings $\downarrow, \uparrow : C(K) \to \mathbb{R}[x]$ given by $f\downarrow = p_f$ and $f\uparrow = q_f$.*

So far the roundings defined in the present section have not provided finitary approximations. The call to the Axiom of Choice did not yield concrete approximations but merely shows that such roundings are plausible. Even the constant rounding yields real constants as bounds, which need further approximation. The solution is to define roundings that yield polynomials of bounded degree and *floating point* coefficients. We shall provide a description of one such rounding in the following chapter.

7.1.5 Approximation of total interval functions

In Section 7.1.2 we saw that function intervals approximate real functions in the way real intervals approximate real numbers. In the following section we shall see how function intervals may be used in the approximation of interval functions.

The poset of extended real intervals $\mathbb{I}(\mathbb{R}^{\pm\infty})$ may be embedded in a poset of extended real valued function intervals $\mathbb{I}(X \to \mathbb{R}^{\pm\infty})$ by the *constant* map:

$$\mathbb{I}(\mathbb{R}^{\pm\infty}) \hookrightarrow \mathbb{I}(X \to \mathbb{R}^{\pm\infty}) \text{ given by } [a, b] \mapsto [\lambda x.a, \lambda x.b] \tag{7.1}$$

taking an interval $[a, b]$ to the function interval with bound functions given by the constant functions corresponding to the bounds a and b. Whenever there is no risk of confusion we shall identify the image of an interval under this embedding with the interval itself, as we did in the case of the singleton embedding mapping poset elements to singleton intervals.

Function intervals may be approximated from the outside by constant intervals through the *outer constant approximation* $\underline{\text{const}}$ defined below. In the language of the preceding sections $\underline{\text{const}}$ yields a downward rounding $\downarrow : \mathbb{I}((\mathbb{R}^{\pm\infty})^n \to \mathbb{R}^{\pm\infty}) \to (\mathbb{I}(\mathbb{R}^{\pm\infty})^n \to \mathbb{I}(\mathbb{R}^{\pm\infty}))$:

Definition 7.1.4 (Outer constant approximation)**.** *Each* $\boldsymbol{f} = [\underline{f}, \overline{f}] \in \mathbb{I}((\mathbb{R}^{\pm\infty})^n \to \mathbb{R}^{\pm\infty})$ *induces an interval function* $\underline{\text{const}}(\boldsymbol{f}, \cdot) : \mathbb{I}(\mathbb{R}^{\pm\infty})^n \to \mathbb{I}(\mathbb{R}^{\pm\infty})$ *called the* outer constant approximation *of* $\boldsymbol{f}$ *at* $\boldsymbol{x}$*. It is defined at each* $\boldsymbol{x} \in \mathbb{I}(\mathbb{R}^{\pm\infty})^n$ *by:*

$$\underline{\text{const}}(\boldsymbol{f}, \boldsymbol{x}) = \inf_{x \in \boldsymbol{x}} [\underline{f}(x), \overline{f}(x)] \tag{7.2}$$

The constant map (7.1) gives a representation of $\mathbb{I}(\mathbb{R}^{\pm\infty})$ *literals* within $\mathbb{I}(X \to \mathbb{R}^{\pm\infty})$. One advantage of working in $\mathbb{I}(X \to \mathbb{R}^{\pm\infty})$ over $\mathbb{I}(\mathbb{R}^{\pm\infty})$ is that interval *variables* may be represented *exactly*:

Definition 7.1.5 (Projection)**.** *Let* $\boldsymbol{x} = (\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n) \in \mathbb{I}(\mathbb{R}^{\pm\infty})^n$ *be a tuple of extended real intervals. The function interval* $[\lambda x.x_i, \lambda x.x_i] \in \mathbb{I}(\boldsymbol{x} \to \mathbb{R}^{\pm\infty})$ *is called the* i^{th} projection *over* $\boldsymbol{x}$ *and denoted* $\text{proj}_i(\boldsymbol{x})$.

Using projections to represent variables means that relationships between variables are retained. Since function intervals are a richer set than number intervals there are also more ways to represent intermediate values during the evaluation of expressions. These differences lead to function interval evaluation of interval expressions generally yielding more accurate approximations. As an illustration of such a situation consider the following simple example:

Example 7.1.6 (Function intervals vs. number intervals)**.** *Let* $F : \mathbb{I}(\mathbb{R}^{\pm\infty}) \to \mathbb{I}(\mathbb{R}^{\pm\infty})$ *be the interval function given by* $\lambda x.2x - x$*. Evaluating* F *at* $[-1, 1]$ *in* $\mathbb{I}(\mathbb{R}^{\pm\infty})$ *proceeds as follows:* $F([-1, 1]) = 2[-1, 1] - [-1, 1] = [-2, 2] - [-1, 1] = [-3, 3]$*. Evaluation in* $\mathbb{I}([-1, 1] \to \mathbb{R}^{\pm\infty})$ *using projections gives:* $F([-1, 1]) = \underline{\text{const}}(2[\lambda x.x, \lambda x.x] - [\lambda x.x, \lambda x.x]) = \underline{\text{const}}([\lambda x.2x, \lambda x.2x] - [\lambda x.x, \lambda x.x]) = \underline{\text{const}}([\lambda x.2x - x, \lambda x.2x - x]) = \underline{\text{const}}([\lambda x.x, \lambda x.x])$*. Evaluating the result at* $[-1, 1]$ *yields:* $\underline{\text{const}}([\lambda x.x, \lambda x.x], [-1, 1]) = \inf_{x \in [-1,1]}[x, x] = [-1, 1]$ *which is more accurate than* $[-3, 3]$.

When we partially evaluate $\underline{\text{const}} : \mathbb{I}((\mathbb{R}^{\pm\infty})^n \to \mathbb{R}^{\pm\infty}) \times \mathbb{I}(\mathbb{R}^{\pm\infty})^n \to \mathbb{I}(\mathbb{R}^{\pm\infty})$ at a tuple $\boldsymbol{x}$ we obtain an *outer* or *downward* rounding $\downarrow = \underline{\text{const}}(\cdot, \boldsymbol{x}) : \mathbb{I}((\mathbb{R}^{\pm\infty})^n \to \mathbb{R}^{\pm\infty}) \to \mathbb{I}(\mathbb{R}^{\pm\infty})$. The dual *inner* or *upward* rounding is readily defined by taking the *supremum* rather than infimum in (7.2). The problem is that this supremum need not exist, making the resulting

function partial. The solution is to extend the notion of interval, as we shall do in the following section.

7.1.6 Inner approximations

So far we have considered one type of approximation, namely *outer approximation*, which allows us to safely decide *inequalities* between expressions in a poset. We shall now consider approximation for deciding *inclusions* between intervals over the poset, which will lead us to the notion of *inner approximation* and that of *generalised intervals*.

How do we approximate the inclusion of intervals $\boldsymbol{x} \subseteq \boldsymbol{y}$ in $\mathbb{I}(\mathbb{R}^{\pm\infty})$ within $\mathbb{I}(\mathbb{F}^{\pm\infty})$? The inclusion, being equivalent with reverse refinement, is equivalent with a pair of inequalities:

$$[a, b] \subseteq [c, d] \Leftrightarrow c \leq a \wedge b \leq d$$

so we can round the endpoints in the appropriate directions to obtain an inclusion of floating point intervals:

$$c\uparrow \leq a\downarrow \wedge b\uparrow \leq d\downarrow \Leftrightarrow [a, b]\downarrow \subseteq [c, d]\uparrow$$

where $\boldsymbol{y}\uparrow$ is called *upward* or *inner* rounding of intervals and is dual to the *downward* or *outer* rounding of intervals induced by outward rounding of endpoints introduced in the previous section. The floating point interval inclusion thus obtained implies the original inclusion:

$$[a, b]\downarrow \subseteq [c, d]\uparrow \Rightarrow [a, b] \subseteq [c, d]$$

and we have a way of safely and efficiently approximating extended real interval inclusions.

There are however two problems hiding in the definition of inner approximation. Namely, how does one approximate a singleton interval from within and how does one safely combine inner approximations? Consider the following example:

$$[3, 4] \subseteq [-1, 1] + \pi \tag{7.3}$$

To obtain an inner approximation for the right hand side of (7.3) will need an inner approximation the singleton interval π. When using real intervals the singleton itself can be taken as an inner approximation but if we are using finitary approximations, π can no

longer be used as an endpoint. However, there are no candidate endpoints x and y that give an inner approximation $\pi \sqsubseteq [x, y]$. The solution is to generalise the definition of interval so that inner approximations of singletons may be represented, which is the topic of the following section.

7.1.7 Generalised intervals

In his seminal work [73, 74] Warmus noted that interval arithmetic, as presented in Section 2.2, has some undesirable properties. In particular, additive inverses do not exist for non-singleton intervals, and as a solution *generalised intervals* were introduced as alternative approximations for real numbers.

Definition 7.1.7 (Generalised interval)**.** *Let* $(R, \leq)$ *be a poset. A* generalised interval *over* R *is given by a pair of elements* $a, b \in R$*, called the* left *and* right endpoint *of the interval, respectively, and we write the interval as* $[a, b]$*. The* set of generalised intervals *is denoted by* $\mathbb{J}(R)$*.*

Since an order interval is given by an ordered pair $a \leq b$ of elements of R and a generalised interval is given by *any* two elements of R we may view $\mathbb{I}(R)$ as a subset of $\mathbb{J}(R)$. By extending the refinement relation $\sqsubseteq$ *syntactically* from $\mathbb{I}(R)$ to $\mathbb{J}(R)$ we obtain the *generalised interval poset*:

Definition 7.1.8 (Generalised interval poset)**.** *Let* $(R, \leq)$ *be a poset and define the* refinement partial order $\sqsubseteq$ *on* $\mathbb{J}(R)$ *by* $[a, b] \sqsubseteq [c, d] \equiv a \leq c \wedge d \leq b$*. Then* $(\mathbb{J}(R), \sqsubseteq)$ *is a poset and we call it the* directed interval poset *over* R*.*

It follows that extending the identity on $\mathbb{I}(R)$ to $\mathbb{I}(R) \hookrightarrow \mathbb{J}(R)$ order-embeds order intervals within the generalised intervals. Generalised intervals do not in general have an interpretation as a set of elements. Instead they may be thought of as an analogy of negative integers, added to the set of order intervals to complete the addition operation. The analogue of *elements* of an order interval is given for generalised intervals by the *singletons* that are in refinement relation with the given interval:

Definition 7.1.9 (Singleton of a generalised interval)**.** *Let* $\mathbb{J}(R)$ *be the set of generalised intervals for some poset* R *and* $\boldsymbol{x} \in \mathbb{J}(R)$*. The* singletons of $\boldsymbol{x}$ *are the singletons* $x \in \mathbb{J}(R)$ *such that* $\boldsymbol{x} \sqsubseteq x$ *or* $x \sqsubseteq \boldsymbol{x}$*. We shall extend the notation for membership and write* $x \in \boldsymbol{x}$ *when* x *is a singleton of* $\boldsymbol{x}$*.*

We extend the notation for singletons of a generalised interval to singletons $x \in R^n$ of tuples $\boldsymbol{x} \in \mathbb{J}(R)^n$ of generalised intervals by writing $x \in \boldsymbol{x}$ if each coordinate x_i of x is a singleton of the corresponding coordinate $\boldsymbol{x}_i$ of $\boldsymbol{x}$, *i. e.* whenever $x_i \in \boldsymbol{x}_i$ for each $i \in \{1, .., n\}$.

The generalised intervals $\mathbb{J}(R)$ complete more than the additive structure on $\mathbb{I}(R)$, they also provide the missing elements needed to complete the lattice structure missing in the interval semilattice $\mathbb{I}(L)$ over lattices L as given in Definition 2.2.7 on page 26 :

Definition 7.1.10 (Generalised interval lattice)**.** *Let* $(L, \sqcup, \sqcap)$ *be a lattice and* $(L, \leq)$ *the associated poset. Then* $(\mathbb{J}(L), \bar{\sqcup}, \bar{\sqcap})$ *is the lattice associated with the refinement partial order* $\sqsubseteq$ *on* $\mathbb{J}(L)$ *with the lattice operations* $\bar{\sqcup}$ *and* $\bar{\sqcap}$ *defined by:*

$$[a, b] \bar{\sqcup} [c, d] = [a \sqcup c, b \sqcap d] \;\; \textit{and} \;\; [a, b] \bar{\sqcap} [c, d] = [a \sqcap c, b \sqcup d]$$

Definition 7.1.10 implies that whenever L is complete then so is $\mathbb{J}(L)$, with $\top_{\mathbb{J}(L)} = [\top_L, \bot_L]$ and $\bot_{\mathbb{J}(L)} = [\bot_L, \top_L]$.

Example 7.1.11 (Generalised extended real intervals)**.** *An example of a complete generalised interval lattice is given by* $\mathbb{J}(\mathbb{R}^{\pm\infty})$*, the lattice of generalised intervals over the lattice of two-point extended real numbers* $\mathbb{R}^{\pm\infty}$ *from Example 2.1.13 on page 23, with top and bottom elements being* $[\infty, -\infty]$ *and* $[-\infty, \infty]$*, respectively.*

The constant embedding $\mathbb{I}(\mathbb{R}^{\pm\infty}) \hookrightarrow \mathbb{I}(X \to \mathbb{R}^{\pm\infty})$ in (7.1) may be syntactically extended to an embedding of generalised intervals $\mathbb{J}(\mathbb{R}^{\pm\infty}) \hookrightarrow \mathbb{J}(X \to \mathbb{R}^{\pm\infty})$. Likewise, we may lift $\underline{\text{const}}$ from a map taking order function intervals to interval functions to a map taking generalised function intervals to generalised interval functions. Generalised intervals form a lattice and as pointed out above, whenever the base lattice is complete, then so is the generalised interval lattice. From Example 2.1.20 we know that if L is complete, then so is the lattice of L-valued functions. Hence, $\mathbb{J}(X \to \mathbb{R}^{\pm\infty})$ is a complete lattice and the definition of the dual upward rounding $\overline{\text{const}}$ corresponding to $\underline{\text{const}}$ gives a total function:

Definition 7.1.12 (Inner constant approximation)**.** *Each* $\boldsymbol{f} = [\underline{f}, \overline{f}] \in \mathbb{J}((\mathbb{R}^{\pm\infty})^n \to \mathbb{R}^{\pm\infty})$ *induces an interval function* $\overline{\text{const}}(\boldsymbol{f}) : \mathbb{J}(\mathbb{R}^{\pm\infty})^n \to \mathbb{J}(\mathbb{R}^{\pm\infty})$ *called the* inner constant approximation *of* $\boldsymbol{f}$ *at* $\boldsymbol{x}$*. It is defined at each* $\boldsymbol{x} \in \mathbb{J}(\mathbb{R}^{\pm\infty})^n$ *by:*

$$\overline{\text{const}}(\boldsymbol{f}, \boldsymbol{x}) = \sup_{x \in \boldsymbol{x}} [\underline{f}(x), \overline{f}(x)]$$

Note that the supremum in the definition on the previous page is taken over *singletons* as defined in Definition 7.1.9 on page 102.

We may evaluate $\overline{\text{const}} : \mathbb{J}((\mathbb{R}^{\pm\infty})^n \to \mathbb{R}^{\pm\infty}) \times \mathbb{J}(\mathbb{R}^{\pm\infty})^n \to \mathbb{J}(\mathbb{R}^{\pm\infty})$ partially at an interval tuple as we did with the order interval version of $\underline{\text{const}}$ at the end of Section 7.1.5 to obtain an *inner* or *upward* rounding $\uparrow = \overline{\text{const}}(\cdot, \boldsymbol{x}) : \mathbb{J}((\mathbb{R}^{\pm\infty})^n \to \mathbb{R}^{\pm\infty}) \to \mathbb{J}(\mathbb{R}^{\pm\infty})$ of generalised function intervals onto generalised intervals.

The *width* of real order intervals may be generalised to arbitrary interval posets by giving a *decreasing* function $w : \mathbb{I}(R) \to \mathbb{R}^{\pm\infty}$, satisfying $w(x) = 0$ for singletons x. It follows that any order interval has non-negative width, but with generalised intervals this is no longer the case.

Definition 7.1.13 (Conjugate interval)**.** *Let R be a poset and $\boldsymbol{x} \in \mathbb{J}(R)$. The* conjugate $\boldsymbol{x}^*$ *of $\boldsymbol{x}$ is obtained by* reversing *the* endpoints *of the interval:* $[a, b]^* = [b, a]$.

We have $\mathbb{J}(\mathbb{R}^{\pm\infty}) = \mathbb{I}(\mathbb{R}^{\pm\infty}) \cup \mathbb{I}(\mathbb{R}^{\pm\infty})^*$, where $\mathbb{I}(\mathbb{R}^{\pm\infty})^* = \{\boldsymbol{x}^* \mid \boldsymbol{x} \in \mathbb{I}(\mathbb{R}^{\pm\infty})\}$ is the set of directed extended real intervals with nonpositive width. The sign of the width of an interval is sometimes called the interval's *direction*. Thus, $\mathbb{I}(\mathbb{R}^{\pm\infty})$ may be viewed as the sub-poset of $\mathbb{J}(\mathbb{R}^{\pm\infty})$ made from intervals with nonnegative direction. Note however that $\mathbb{J}(R)$ cannot in general be obtained from $\mathbb{I}(R)$ by conjugation:

Example 7.1.14 ($\exists R.\mathbb{J}(R) \neq \mathbb{I}(R) \cup \mathbb{I}(R)^*$)**.** *A counterexample is given by choosing for R the poset $(X \to \mathbb{R}^{\pm\infty}, \dot{\leq})$ of extended real valued functions, where e.g. $\boldsymbol{f} = [\lambda x.x, \lambda x.-x]$ is in $\mathbb{J}(X \to \mathbb{R}^{\pm\infty})$, but neither $\boldsymbol{f}$ nor $\boldsymbol{f}^*$ is in $\mathbb{I}(X \to \mathbb{R}^{\pm\infty})$.*

Example 7.1.15 (Conjugate approximation)**.** *Any* outer *approximation $\boldsymbol{x}$ of a* singleton x *in $\mathbb{J}(\mathbb{R}^{\pm\infty})$ corresponds to an order interval, since $\boldsymbol{x} = [\underline{x}, \overline{x}] \sqsubseteq x \Leftrightarrow \underline{x} \leq x \wedge x \leq \overline{x} \Rightarrow \underline{x} \leq \overline{x}$. Therefore the conjugate $\boldsymbol{x}^* = [\overline{x}, \underline{x}]$ of $\boldsymbol{x}$ gives an inner approximation of x: $x \sqsubseteq \boldsymbol{x}^*$. In particular, any outer approximation of π, such as $[3.1, 3.2] \sqsubseteq \pi$ from Section 7.1.3, yields an inner approximation $\pi \sqsubseteq [3.2, 3.1]$ of π by conjugation.*

Definition 2.2.8 in Section 2.2 introduced the containment property characterising outer interval extensions. We reformulate and extend the definition in terms of generalised intervals:

Definition 7.1.16 (Inner and outer generalised containment property)**.** *Let R be a poset and $f : R^n \to R$ be a function. A function $\boldsymbol{g} : \mathbb{J}(R)^n \to \mathbb{J}(R)$ satisfies the* inner generalised containment property *with respect to f if $x_i \sqsubseteq \boldsymbol{x}_i$ implies $f(x) \sqsubseteq \boldsymbol{g}(\boldsymbol{x})$ for each $i \in$*

$\{1, \ldots, n\}$, $x \in R^n$ *and* $\boldsymbol{x} \in \mathbb{J}(R)^n$. *Dually,* $\boldsymbol{g}$ *satisfies the* outer generalised containment property *with respect to* f *if* $\boldsymbol{x}_i \sqsubseteq x_i$ *implies* $\boldsymbol{g}(\boldsymbol{x}) \sqsubseteq f(x)$ *for each* $i \in \{1, \ldots, n\}$, $x \in R^n$ *and* $\boldsymbol{x} \in \mathbb{J}(R)^n$.

Note that $\boldsymbol{g}$ satisfies the inner or outer containment property with respect to f if and only if $f \dot{\sqsubseteq} \boldsymbol{g}$ or $\boldsymbol{g} \dot{\sqsubseteq} f$, *i. e.* whenever $\boldsymbol{g}$ is pointwise above or below f, respectively. Functions satisfying a containment property are called *generalised extensions*:

Definition 7.1.17 (Inner and outer generalised interval extension)**.** *Let R,* f *and* $\boldsymbol{g}$ *be as in Definition 7.1.16. If* $\boldsymbol{g}$ *satisfies the* inner generalised containment property *with respect to* f*, then* $\boldsymbol{g}$ *is called an* inner generalised interval extension *of* f *and if* $\boldsymbol{g}$ *satisfies the* outer generalised containment property *with respect to* f*, then* $\boldsymbol{g}$ *is called an* outer generalised interval extension *of* f.

Functions that satisfy both the inner and outer containment property with respect to a function are said to satisfy the *generalised containment property*. Such functions are called *generalised interval extensions*.

7.2 Partial functions

In the preceding discussion we have only considered *total* functions. In practice, many functions will only give meaningful results for a strict subset of their domain. Functions which are *undefined* over a nonempty subset of their domain are called *partial*.

Definition 7.2.1 (Partial function)**.** *Let X and Y be sets and let the relation* $f \subseteq X \times Y$ *satisfy the following property: for any* $x \in X$ *there is* at most one $y \in Y$ *related to x by* f. *Whenever this is the case we call* f *a* partial function *from X to Y and write* $f : X \rightharpoonup Y$.

When each member of X is related to *precisely one* member of Y we call the relation a *total function*. If some $x \in X$ is related to *two or more* members of Y we call the relation *multi-valued* at x. There is a projection of multi-valued relations onto partial functions obtained by removing from the relation all points (x, y) for which the relation is multi-valued at x. We shall see concrete examples of such partial functions in the section below.

7.2.1 Partial extensions

In Section 2.3 we noted that partial functions may arise as *extensions* of total functions. Extensions need not be partial, one example of a total extension is given by arithmetic negation extended by infinities:

Example 7.2.2 (Negation total on $\mathbb{R}^{\pm\infty}$)**.** *Negation is total on* $\mathbb{R}$*, we therefore only need to define the negation on the infinities:* $-(\infty) = -\infty$ *and* $-(-\infty) = \infty$*. Thus, the extension of negation to* $\mathbb{R}^{\pm\infty}$ *is total.*

Note that the infinitie values $-\infty$ and ∞ correspond to the IEEE signed infinities described in Section 4.2.2. However, the extension of $\mathbb{R}$ to $\mathbb{R}^{\pm\infty}$ does not include values corresponding to the signed zeros. As a first example of a partial function consider the extension of addition on $\mathbb{R}$ to $\mathbb{R}^{\pm\infty}$:

Example 7.2.3 (Addition partial on $\mathbb{R}^{\pm\infty}$)**.** *We wish to extend the addition operation on* $\mathbb{R}$ *to an operation on* $\mathbb{R}^{\pm\infty}$*. Since addition is total on* $\mathbb{R}$ *we only need to specify values for* $x + \alpha$ *and* $\alpha + \beta$ *for* $x \in \mathbb{R}$ *and* $\alpha, \beta \in \{-\infty, \infty\}$*. We may define* $x + \alpha = \alpha$ *and* $\alpha + \alpha = \alpha$*, but there is no clear candidate value for* $\alpha - \alpha$*. Leaving addition* undefined *for* $(\alpha, -\alpha)$ *extends the operation* partially *to* $\mathbb{R}^{\pm\infty}$*.*

In the above example the *obstruction* to totality of the extended operation lies entirely inside the set by which the domain of the operation is extended. Multiplication provides an example where the obstruction relates an element of the original domain with the new elements:

Example 7.2.4 (Multiplication partial on $\mathbb{R}^{\pm\infty}$)**.** *Let* $x \in \mathbb{R}_{\neq 0}$ *be a nonzero real number and* $\alpha \in \{-\infty, \infty\}$*. We extend multiplication from* $\mathbb{R}$ *to* $\mathbb{R}^{\pm\infty}$ *by defining* $x\alpha = \operatorname{sign}(x)\alpha$*, there is however no clear candidate value for* 0α*. By leaving multiplication* undefined *at* $(0, \alpha)$ *and* $(\alpha, 0)$ *we obtain a* partial *operation on* $\mathbb{R}^{\pm\infty}$*.*

One of the most familiar partial functions on real numbers is division, which when seen as a real valued function is undefined for zero divisors. We may however extend division by infinities as we did with addition in the example above:

Example 7.2.5 (Division partial on $\mathbb{R}^{\pm\infty}$)**.** *Division by zero is undefined on all of* $\mathbb{R}$*. Let* $x \in \mathbb{R}_{\neq 0}$ *be a nonzero real number and* $\alpha, \beta \in \{-\infty, \infty\}$*. We may attempt to extend division to an operation* $(\mathbb{R}^{\pm\infty})^2 \to \mathbb{R}^{\pm\infty}$ *by defining* $0/x = x/\alpha = 0$ *and* $x/0 = \alpha/x = \operatorname{sign}(x)\operatorname{sign}(\alpha)\infty$*,*

but there are no clear candidate values for $0/0$ *or* α/β*. By leaving division* undefined *for* $(0,0)$ *and* (α,β) *division extends* partially *to* $\mathbb{R}^{\pm\infty}$.

Examples 7.2.3–7.2.5 above used a general strategy for extending continuous functions to the closure of their domain. $\mathbb{R}^{\pm\infty}$ may be seen as the topological closure of $\mathbb{R}$ obtained by taking the union of $\mathbb{R}$ with its *boundary* $\partial\mathbb{R} = \{-\infty, \infty\}$: $\mathbb{R}^{\pm\infty} = \mathbb{R} \cup \partial\mathbb{R}$. The strategy for extending a continuous function $f : X \to \mathbb{R}$ by a point x in the boundary ∂X of X is to *extend by limits* in the following sense: if each sequence $\{x_k\}_{k\geq 0} \subset X$ such that $x_k \to x$ as $k \to \infty$ satisfies $f(x_k) \to r$ as $x_k \to x$, then we define $f(x) = r$. If there is no such r or if two sequences converging to x yield different rs then the extension is undefined at x and thus a partial function.

We can now return to the examples on the previous page and formally identify the obstructions to totality for extended addition, multiplication and division. In the case of addition we wish to define $\infty - \infty$ as some value in $\mathbb{R}^{\pm\infty}$. We can approach the point $(\infty, -\infty) \in (\mathbb{R}^{\pm\infty})^2$ along the sequences $(x_k, y_k) = (k, -\infty)$ and $(x'_k, y'_k) = (\infty, -k)$, but since $k - \infty = -\infty$ and $\infty - k = \infty$ for all $k \in \mathbb{N}$. Thus, $x_k + y_k \to -\infty$ while $x'_k + y'_k \to \infty$ as $k \to \infty$, and so the extension becomes *multi-valued* at $(\infty, -\infty)$.

Multiplication was readily extended to all of $(\mathbb{R}^{\pm\infty})^2$ except the points containing a zero *and* an infinite coordinate. We may approach the point $(0, \infty)$ along the sequences $(x_k, y_k) = (0, k)$ and $(x'_k, y'_k) = (1/k, \infty)$, but $x_k y_k = 0$ while $x'_k y'_k = \infty$ for each $k \geq 1$ and thus multiplication becomes *multi-valued* at $(0, \infty)$.

In the case of division we wish to define $0/0$ as some value in $\mathbb{R}^{\pm\infty}$. As we did in the preceding paragraphs, we may approach the point $(0,0)$ along the sequences $(x_k, y_k) = (0, 1/k)$ and $(x'_k, y'_k) = (1/k, 0)$. Clearly, $x_k/y_k = 0$ and $x'_k/y'_k = \infty$ for each $k \geq 1$ and so division becomes *multi-valued* at $(0,0)$.

It follows from Definition 7.2.1 on page 105 that partial functions may arise as *inverses* of total functions. The *inverse* ρ^{-1} of a *relation* $\rho \subseteq X \times Y$ is the relation $\{(y, x) \in Y \times X \mid (x, y) \in \rho\}$. Partiality from *multi-valuedness* may arise when inverses are taken of *non-injective* functions. The typical example is the function that squares a real number. Since each positive real number $x \in \mathbb{R}_{>0}$ is the square of both $\sqrt{x}$ and $-\sqrt{x}$, it follows that the inverse of the square function would be *multi-valued* at each positive real. To circumvent this obstruction, the nonnegative branch is chosen, *i. e.* the *positive root* is chosen as the

value of the square root, making it a total function on $\mathbb{R}_{\geq 0}$. An obstruction remains to the totality of $\sqrt{}$ on all of $\mathbb{R}$, namely that the function it is the inverse of is not surjective.

Partiality from *undefinedness* occurs for inverses of *non-surjective* functions. The square root has multiple obstructions to totality and we covered one in the previous paragraph. To give an example that relies entirely on non-surjectivity we consider the exponential and logarithm functions. Since the exponential e^x of a real number x is positive it follows that the natural logarithm is *undefined* for all negative reals.

7.2.2 Making partial functions total

Any partial function $f : X \rightharpoonup Y$ can be extended to a total one by adding values to the codomain Y that f can map values in $X \setminus f^{-1}(Y)$ to. In the simplest case one unique value is used to represent undefinedness:

Definition 7.2.6 (∅-extension)**.** *Let X and Y be sets and let $X_\varnothing$ and $Y_\varnothing$ denote the disjoint union of the empty set with X and Y, respectively. There is a canonical extension of partial functions $f : X \rightharpoonup Y$ to total functions $f_\varnothing : X_\varnothing \to Y_\varnothing$ defined by $f_\varnothing = f$ on $f^{-1}(Y)$ and $f_\varnothing = \varnothing$ on $X_\varnothing \setminus f^{-1}(Y)$. We call $f_\varnothing$ the* canonical total extension *of f.*

Partial functions $f : X \rightharpoonup Y$ may be retrieved from their extensions $f_\varnothing$ by removing each member of $f_\varnothing$ containing $\varnothing$: $f = f_\varnothing \setminus \{(x, y) \in X \times Y \mid x = \varnothing \vee y = \varnothing\}$ and we can thus go between partial functions and canonical extensions. We shall therefore drop the subscript $\varnothing$ whenever it is clear from the context that we are using ∅-extensions.

Interval extensions of partial functions may also be total, this is because an interval may approximate many values in the codomain of the extended function simultaneously. The partial extension of addition in Example 7.2.3 has a total outer interval extension, in the sense of Definition 7.1.17, given by:

$$[\underline{x}, \overline{x}] + [\underline{x}', \overline{x}'] = \begin{cases} [-\infty, \infty] & \text{if } \underline{x} = -\underline{x}' = \infty \text{ or } \underline{x} = -\underline{x}' = -\infty \\ & \text{or } \overline{x} = -\overline{x}' = \infty \text{ or } \overline{x} = -\overline{x}' = -\infty \text{ and} \\ [\underline{x} + \underline{x}', \overline{x} + \overline{x}'] & \text{otherwise} \end{cases} \tag{7.4}$$

If we chose to return $[\infty, -\infty]$ when the operations on the bounds are undefined, rather than $[-\infty, \infty]$ as in (7.4), then the resulting interval function would be an inner interval extension of extended real addition.

This choice always arises when a relation f is multi-valued over some set X and a total

function which has a maximal interval extension outside X. Then the choice of interval approximation for the set $\{y \mid (x, y) \in f \wedge x \in X\}$, of values y associated by f to some member $x \in X$, determines the type of extension. As long as all approximations are chosen consistently, *i. e.* all inner or all outer, then the resulting interval function is a total inner or outer interval extension, respectively. In particular, total interval extensions for the partial extensions of multiplication in Example 7.2.4 and division in Example 7.2.5 may be obtained in the described manner from maximal interval extensions for addition on $\mathbb{R}$ and division on $\mathbb{R}$ with nonzero divisors.

7.2.3 Many valued logics

Let $\mathbb{B}$ denote the set $\{\mathtt{t}, \mathtt{f}\}$ of Boolean truth values where `t` denotes true and `f` denotes false. Predicates may be defined as total $\mathbb{B}$-valued functions. When they are composed with partial functions we obtain *partial* $\mathbb{B}$-valued functions. Consider the *nonnegativity* predicate p defined by $x \mapsto x \geq 0$ and the square root function $x \mapsto \sqrt{x}$ and their composition $p \circ \sqrt{\cdot}$ given by $x \mapsto \sqrt{x} \geq 0$. The composition $p \circ \sqrt{\cdot}$ is a partial $\mathbb{B}$-valued function on $\mathbb{R}$ and therefore yields a total $\mathbb{B}_\varnothing$-valued $\varnothing$-extension. We may view the value $\varnothing$ in $\mathbb{B}_\varnothing$ as a *third* truth value `e`.

Interpreting undefinedness as a truth value leads to a three-valued logic $\mathbb{B}_\mathtt{e} = \{\mathtt{e}, \mathtt{t}, \mathtt{f}\}$, called *partial Boolean logic*, with the values `e`, `t` and `f` called undefined, true and false, respectively. One can retrieve the two-valued Boolean logic by mapping `e` to either `t` or `f`. An example of the latter choice is given by the handling of the NaN value by predicates in Section 3.3.

Definition 2.2.4 introduced interval extensions of relations on posets. Given a function $f : X \to Y$ we define the *set extension* $f^\wp$ of f as the set-valued function $f^\wp : \wp(X) \to \wp(Y)$ given by $A \mapsto \{f(x) \mid x \in A\}$. Set extensions of relations $\rho : X^2 \to \mathbb{B}$ become partial functions $\rho^\wp : \wp(X)^2 \to \wp(\mathbb{B})$ where $\wp(\mathbb{B}) = \{\varnothing, \{\mathtt{t}\}, \{\mathtt{f}\}, \{\mathtt{t}, \mathtt{f}\}\}$ may be viewed as a set of *four* truth values $\{\mathtt{e}, \mathtt{t}, \mathtt{f}, \mathtt{u}\}$. We interpret the singleton values as the corresponding Boolean value and call the values $\mathtt{e} = \varnothing$ and $\mathtt{u} = \{\mathtt{t}, \mathtt{f}\}$ *undefined* and *undecided*, respectively.

Note that total relations yield non-surjective set extensions. Since each pair in X^2 maps to some member of $\mathbb{B}$ it follows that the extension maps sets of pairs to nonempty members of $\wp(\mathbb{B})$ effectively resulting in a *three*-valued logic $\{\mathtt{t}, \mathtt{f}, \mathtt{u}\}$. Ordering the set under the restriction of the relation $\lesssim$, given in Definition 8.1.7 on page 120, gives us the poset

commonly known as the *flat Booleans* and denoted by $\mathbb{B}_{\mathtt{u}}$.

Set extensions of partial predicates may yield the empty truth value $\mathtt{e}$ representing application of the predicate entirely outside its domain of definition. An example is given by the partial predicate $p \circ \sqrt{\cdot}$ applied to the set $\mathbb{R}_{<0}$ of negative real numbers. Clearly there is no $x \in \mathbb{R}_{<0}$ such that $\sqrt{x} \geq 0$ so $\mathbb{R}_{<0} \mapsto \varnothing$ by $(p \circ \sqrt{\cdot})^{\wp}$. Applying $(p \circ \sqrt{\cdot})^{\wp}$ to a general set of reals yields *sets of partial Boolean* values. We will not consider the full eight valued logic $\wp(\{\mathtt{t}, \mathtt{f}, \mathtt{e}\})$, instead we interpret the Boolean singletons $\{\mathtt{t}\}$ and $\{\mathtt{f}\}$ as the corresponding Boolean values $\mathtt{t}$ and $\mathtt{f}$, the singleton $\{\mathtt{e}\}$ and the empty set $\varnothing$ as $\mathtt{e}$ and the remaining subsets as $\mathtt{u}$.

The four-valued logic $\{\mathtt{e}, \mathtt{t}, \mathtt{f}, \mathtt{u}\}$ generalises and unifies the three valued logics $\mathbb{B}_{\mathtt{e}}$ and $\mathbb{B}_{\mathtt{u}}$ by differentiating undefined and undecided truth values enabling the identification of causes for non Boolean results. Undefined results from partiality while undecided should be thought of as arising from multi-valuedness, *e.g.* by application of a set extension to a set containing members of both the support of the predicate and its complement.

Binary Boolean logic may be retrieved from $\wp(\mathbb{B})$ by mapping both undecided and undefined values to singleton values. A concrete example is given by Example 2.2.5 where a poset structure on $\mathbb{I}(R)$ compatible with the original order on the poset $(R, \leq)$ is recovered from the set extension of $\leq$ by mapping the non singleton truth values to false.

8 Automated numerical theorem prover

CONTENTS

In the following chapter we present a prototype automated numerical theorem prover for correctness theorems resulting from the analysis in Chapter 3. The algorithm uses a standard approach in automated numerical theorem proving, based on *safe numerical approximation* and *domain subdivision*, but uses a novel interval arithmetic. Automation is achieved through *approximation* which comes at the cost of *completeness*, in other words, we prove some theorems automatically but cannot prove them all. The class of theorems whose proofs are amenable to automation using approximate reasoning consists of theorems without *touching*. Touching occurs when the boundaries of supports of predicates in a theorem are not disjoint. A simple example of a theorem exhibiting touching is given by:

Example 8.0.7 (Touching)**.** *Let φ be defined by $D_1 \subseteq D_2$, where the discs D_1 and D_2 are given by $D_1 = \{(x, y) \in \mathbb{R}^2 \mid \sqrt{x^2 + (y-1)^2} \leq 1\}$ and $D_2 = \{(x, y) \in \mathbb{R}^2 \mid \sqrt{x^2 + (y-2)^2} \leq 2\}$. The boudaries of D_1 and D_2 are tangential at the origin and we say that φ exhibits* touching *at* $(0, 0)$.

8.1 Approximation of predicates

A *solution* of a predicate $p : X \to \mathbb{B}$ is an $x \in X$ such that $p(x) = \mathtt{t}$ and a *counterexample* to p is an $x \in X$ such that $p(x) = \mathtt{f}$. To *prove* p over A is to show that A is contained in the set $p^{-1}(\mathtt{t})$ of solutions of p and to *disprove* p over A is to show that a counterexample to p lies in A. Recall from Section 7.2.3 that we may view the values of the three valued logic $\{\mathtt{t}, \mathtt{f}, \mathtt{u}\}$ as *sets* of Boolean values $\{\{\mathtt{t}\}, \{\mathtt{f}\}, \{\mathtt{t}, \mathtt{f}\}\}$. A function $\mathcal{A}_p : \wp(X) \to \{\mathtt{t}, \mathtt{f}, \mathtt{u}\}$ is called a *safe approximation* of p if $p(x) \in \mathcal{A}_p(A)$ for all $x \in A$ and $A \in \wp(X)$ and $\mathcal{A}_p(A)$ is then called a *safe approximation of p over A*. In particular, safe approximations satisfy the following property: if $\mathcal{A}_p(A) = \mathtt{t}$ then $A \subseteq p^{-1}(\mathtt{t})$ and if $\mathcal{A}_p(A) = \mathtt{f}$ then $A \subseteq p^{-1}(\mathtt{f})$, *i. e.* safe approximations allow us to *prove* or *disprove* predicates.

By restricting safe approximations to a subset S of $\wp(X)$ we obtain safe S-approximations. Approximations $\wp(R) \to \{\mathtt{t}, \mathtt{f}, \mathtt{u}\}$ of predicates over a poset R may be restricted to $\mathbb{I}(R)$-approximations $\mathbb{I}(R) \to \{\mathtt{t}, \mathtt{f}, \mathtt{u}\}$, also called order interval approximations. As an example we give the safe interval approximation of the inequality relation $\leq$ on $\mathbb{R}^{\pm\infty}$:

Example 8.1.1 (Safe interval approximation of $\leq$)**.** *Let $\leq$ be the canonical ordering on $\mathbb{R}^{\pm\infty}$ and $\mathcal{A}_{\leq}$ be the interval relation $\leq_{\mathbb{I}}$ from Example 2.2.5 induced by $\leq$. Then $\mathcal{A}_{\leq}$ is a*

safe $\mathbb{I}(\mathbb{R}^{\pm\infty})$*-approximation of* $\leq$ *in each argument.*

Note that considering the three-valued logic values as sets of Boolean values allows them to be ordered by reverse inclusion $\supseteq$, making $(\{\mathtt{t}, \mathtt{f}, \mathtt{u}\}, \supseteq)$ into the poset where $\mathtt{t}$ and $\mathtt{f}$ are maximal and $\mathtt{u}$ is the least element, *i. e.* the flat Boolean poset $(\mathbb{B}_{\mathtt{u}}, \lesssim)$ from Section 7.2.3. We can re-formulate the definition of safe approximation for predicates as: $\mathcal{A}_p : \wp(X) \to \mathbb{B}_{\mathtt{u}}$ is a *safe approximation* of p if $\mathcal{A}_p(A) \lesssim p(x)$ for each $x \in A$ and each $A \subseteq X$. This definition is generalised to $\mathbb{B}_{\mathtt{eu}}$-valued predicates and extensions in Definition 8.1.13 on page 123.

8.1.1 Safe numerical approximation

Safe approximations for real predicates are traditionally computed by *natural interval approximations* of the predicates. Natural approximations are obtained from real predicates by replacing the constituent real functions and relations with corresponding interval extensions. The simplest case is provided by constant interval extensions:

Example 8.1.2 (Constant interval approximation)**.** *Consider the extended real predicate* $p : \mathbb{R}^{\pm\infty} \to \mathbb{B}$ *given by* $x \mapsto \frac{5x}{8} \leq \frac{3+5x}{8}$*. A safe approximation* $\mathcal{A}_p$ *of* p *may be computed using the outer interval extensions for addition and multiplication described in Section 7.2.2. We get* $\mathcal{A}_p([0,1]) = \mathcal{A}_{\leq}([0, \frac{5}{8}], [\frac{3}{8}, 1]) = \mathtt{u}$*, where* $\mathcal{A}_{\leq}$ *is the safe approximation of* $\leq$ *from Example 8.1.1 on the previous page. We interpret the result* $\mathtt{u}$ *as the approximation* $\mathcal{A}_p$ *being too weak to decide* p *over* $[0, 1]$*.*

The undecided result of Example 8.1.2 shows that safe approximations may severely overapproximate and often fail to decide seemingly trivially provable predicates. When we are using *isotone* interval extensions in the approximation, *i. e.* such extensions $\boldsymbol{f}$ of constituent functions f of p that $\boldsymbol{x} \sqsubseteq \boldsymbol{x}'$ implies $\boldsymbol{f}(\boldsymbol{x}) \sqsubseteq \boldsymbol{f}(\boldsymbol{x}')$, then we may increase the precision of the approximation by reducing the domain of the predicate. By dividing the decision problem over the initial domain into decision problems over subsets, tighter approximations of numerical expressions are obtained, leading to better approximations of the predicate over the sub-domains. The following section describes this approach.

8.1.2 Domain subdivision

Domain subdivision is a method for improving an approximation of a predicate by reducing the domain of the predicate. Consider the predicate p from Example 8.1.2. We were unable to decide p over $[0, 1]$ because the interval approximation $[0, \frac{5}{8}]$ of the left hand side and $[\frac{3}{8}, 1]$ of the right hand side of the inequality intersect. By splitting the domain $[0, 1]$ of p into the two halves $[0, \frac{1}{2}]$ and $[\frac{1}{2}, 1]$ and then using $\mathcal{A}_p$ to approximate p over each half we obtain: $\mathcal{A}_p([0, \frac{1}{2}]) = \mathcal{A}_{\leq}([0, \frac{5}{16}], [\frac{3}{8}, 1]) = \mathtt{t}$ and $\mathcal{A}_p([\frac{1}{2}, 1]) = \mathcal{A}_{\leq}([\frac{5}{16}, \frac{5}{8}], [\frac{11}{16}, 1]) = \mathtt{t}$.

We have proved that p is true over sets covering $[0, 1]$ and thus true over all of $[0, 1]$ using the same approximation $\mathcal{A}_p$ as in Example 8.1.2, but because the reduction in domain size led to an improvement in the approximation of p we managed to decide the predicate, paying the cost of evaluating p twice. The fact that iterated subdivision creates an exponential number of sub-problems means that the improvements in approximation come at an exponential computational cost. Since implementations necessarily use finitary representations and bounded space and time, it follows that many predicates become practically undecidable with this approach. Consider the following example:

Example 8.1.3 (Tight inequality)**.** *Consider the extended real predicate* $p : \mathbb{R}^{\pm\infty} \to \mathbb{B}$ *given by* $x \mapsto x \leq x + 2^{-n}$ *and its natural constant interval approximation* $\mathcal{A}_p$*. Clearly,* p *holds over all of* $\mathbb{R}^{\pm\infty}$ *and therefore over any subset, however evaluating* $\mathcal{A}_p$ *over a generic interval* $[a, b]$ *yields:* $\mathcal{A}_p([a, b]) = \mathcal{A}_{\leq}([a, b], [a + 2^{-n}, b + 2^{-n}])$ *which holds if and only if the width of the interval is sufficiently small:* $\mathcal{A}_p([a, b]) = \mathtt{t} \Leftrightarrow b - a \leq 2^{-n}$.

The order in which the set of sub-problems is processed has an effect on the expected complexity of the algorithm. Processing the set as a *queue*, *i. e.* in first in, first out order, corresponds to a *breadth-first* search. Dually, processing the set as a *stack*, *i. e.* in last in, first out order, corresponds to *depth-first* search.

Deciding p from Example 8.1.3 over the interval $[0, 1]$ with bisection as subdivision method requires n bisections, generating 2^n sub-problems, regardless of search strategy. The following example shows that the choice of search strategy can reduce the complexity class of a decision problem:

Example 8.1.4 (Search strategy affects complexity)**.** *Consider deciding the extended real predicate* p *given by* $x \mapsto x > 2^{-n}$ *over the interval* $[0, 1]$*. As in Example 8.1.3 sub-*

problems with domain width larger than 2^{-n} *yield the undecided value* $\mathtt{u}$*. Using breadth-first search generates a best-case exponential number of sub-problems before identifying a counterexample. On the other hand, a left-baised depth-first search descends directly towards 0 and decides* p *by returning* $\mathtt{f}$ *for the interval* $[0, 2^{-n}]$*, thus having proved the predicate false by exhibiting a nonempty set of counterexamples, while generating a linear number of sub-problems.*

While the predicates in Examples 8.1.2, 8.1.3 and 8.1.4 may, given enough resources, be decided using constant approximations and bisection, many cannot. A simple example is given by taking the limit as $n \to \infty$ in Example 8.1.4:

Example 8.1.5 (Undecided predicate)**.** *Let the predicate* $p : [0, 1] \to \mathbb{B}$ *be given by* $x \mapsto x > 0$ *and* $\mathcal{A}_p$ *be the natural constant interval approximation of* p*. Altough* $\mathcal{A}_p(\boldsymbol{x}) = \mathtt{t}$ *for each sub-interval* $\boldsymbol{x}$ *of* $[0, 1]$ *that does not contain 0, we have* $\mathcal{A}_p([0, 2^{-n}]) = \mathtt{u}$ *for each* n*.* $\mathcal{A}_p$ *can therefore not decide* p *over* $[0, 1]$ *within a finite number of bisections.*

We have seen that the choice of search strategy can have a dramatic effect on the performance of a decision procedure. Consider a variation on Example 8.1.5 where the predicate p is given by $x \mapsto x < 0$. $\mathcal{A}_p(\boldsymbol{x}) = \mathtt{f}$ for each interval $\boldsymbol{x} \in \mathbb{I}([0, 1])$ such that $0 <_{\mathbb{I}} \boldsymbol{x}$. If left-baised depth-first strategy is chosen, then the decision problem will not be decided within a finite number of subdivisions. Any other search strategy will decide the predicate as soon as the strategy deviates from left-baised depth-first, thus encountering an interval of counterexamples witin a finite number of bisections.

Note that as a consequence of the above example any *depth-first* strategy may lead to a *non-terminating* computation. In this particular example it is caused by a repeated failure to generate a thin interval by bisecting non-thin ones. To eliminate non-termination due to search strategy choice one should always choose a strategy that revisits each sub-problem within a finite number of steps. The simplest among such strategies are the breadth-first search strategies.

When considering subdivision for predicates with more than one variable, there is another strategy choice beyond sub-problem processing order. All variable domains may be subdivided at once, but this leads to a rapid increase in the size of the sub-problem set potentially exhausting the available space resources. Usually, one variable domain is subdivided at a time, which introduces the choice of a *subdivision direction selection*

strategy.

The subdivision direction selection strategy chosen determines the *shape* of the domain of sub-problems. To see this consider a decision problem with two variables $x_1, x_2 \in [0, 1]$ ranging over the same interval. The domain of the problem is clearly a unit square, but the domains of its immediate sub-problems are rectangular. In the case of bisecting say x_1 we get a *left* sub-problem with $x_1 \in [0, \frac{1}{2}]$ and $x_2 \in [0, 1]$ and a *right* sub-problem with $x_1 \in [\frac{1}{2}, 1]$ and $x_2 \in [0, 1]$. Clearly, if x_1 is split more often than x_2, then the generated sub-problems will have domains that are thin and long while if x_2 is split more often than x_1, then sub-problems with wide and short domains are generated.

Considering a predicate with many variables that is *constant* in one or more variable we see that, much as in the case of search strategies, subdivision has to revisit each variable within a finite number of iterations. Otherwise the computation may *fail to terminate* for decidable problems due to the choice of subdivision strategy. To see this suppose that our algorithm splits only the variables that the predicate is constant in. Since we are splitting, the predicate was not decided before and since we are splitting along a variable that does not affect the approximation, the splitting cannot improve the approximation obtained for the sub-problems, resulting in a non-terminating sequence of splittings. The standard way of eliminating non-termination due to splitting direction selection the *round-robin* or the *largest-first* splitting strategies may be employed.

The round-robin splitting strategy amounts to ordering the variables in a circular queue, resulting in no variable being split more than once more than any other. Round-robin therefore maintains the proportions of the initial domain. The largest-first splitting strategy chooses the variable with greatest domain width to split. As a consequence, largest-first splitting leads to sub-problem domains with roughly the same width for each variable.

8.1.3 Approximation of Boolean functions

As we saw in the previous sections, approximations of predicates are $\{\mathtt{e}, \mathtt{t}, \mathtt{f}, \mathtt{u}\}$-valued functions, *e.g.* total predicates are approximated by $\{\mathtt{t}, \mathtt{f}, \mathtt{u}\}$-valued functions.

The way we lift the propositional connectives to $\{\mathtt{t}, \mathtt{f}, \mathtt{u}\}$ has consequences to the deductive system we are implicitly defining. The danger is that rules defined with a specific interpretation in mind may yield a system which has surprising properties, *e.g.* theorems

¬	
t	f
f	t
u	u

∨	t	f	u
t	t	t	t
f	t	f	u
u	t	u	u

Figure 8.1: Standard extension of propositional Boolean logic by the undecided value u.

¬	
t	f
f	t
u	u
e	e

∨	t	f	u	e
t	t	t	t	e
f	t	f	u	e
u	t	u	u	e
e	e	e	e	e

Figure 8.2: Propositional Boolean logic extended conservatively to {e, t, f, u}.

that should not hold may become provable. In Figure 8.1 we present the traditional extension of propositional logic by *undecidedness*. The interaction between u and the Boolean values is determined by considering the set extensions of the connectives and mapping the resulting sets back to {t, f, u}. The remaining standard connectives ∧ and → are obtained the usual way from:

$$\varphi \wedge \psi \equiv \neg\left((\neg\varphi) \vee (\neg\psi)\right) \text{ and } \varphi \rightarrow \psi \equiv (\neg\varphi) \vee \psi \tag{8.1}$$

As noted in Section 7.2.3 partial functions lead to partial predicates, and the *undefined* truth value e. We may further extend the three valued logic presented in Figure 8.1 to the four valued logic {e, t, f, u}, as shown in Figure 8.2, again obtaining the remaining connectives from (8.1). Note that the extension is *conservative* in the sense that partial formulas not provable in {t, f, u} are still not provable. The *undefined* truth value e is treated as an exception, signalling that nonsensical results are being obtained, such as attempted evaluation of log(−1). Thus, as soon as an undefined value is computed for a predicate, it is propagated through the Boolean structure and returned.

Section 11.6 of the ARM [1] describes cases when exceptional behaviour may be altered by an Ada implementation, and conditions under which operations with undefined operations may be omitted.

¬	
t	f
f	t
u	u
e	e

∨	t	f	u	e
t	t	t	t	t
f	t	f	u	e
u	t	u	u	u
e	t	e	u	e

∧	t	f	u	e
t	t	f	u	e
f	f	f	f	f
u	u	f	u	u
e	e	f	u	e

→	t	f	u	e
t	t	f	u	e
f	t	t	t	t
u	t	u	u	u
e	t	e	u	e

Figure 8.3: Propositional Boolean logic extended to {e, t, f, u}.

The conservative extension of propositional logic to {e, t, f, u} has one drawback, namely that it does not satisfy the *rule of least surprise*. The rule states that if there is no general consensus on the solution to a problem, then those solutions should be adopted that result in the most intuitive or natural system. To illustrate this vague notion consider a concrete example:

Example 8.1.6 (Mysterious evaluation order)**.** *Let the predicate p_1 be given by $x \mapsto x < 0$ and the partial predicate p_2 be given by $x \mapsto \sqrt{x} \geq 0$. We may then form $p \equiv p_1 \vee p_2$ and consider the validity of p over $\mathbb{R}$. People tend to give two different solutions to this problem: one group uses strict evaluation of the predicates and the conservative extension from Figure 8.2 concluding that the problem is* ill posed *and therefore has no solution. The other group inspect the predicate, realising that for any given value, either p_1 or p_2 returns* t*, and since $\vee$ should hold whenever any of its argument does, they conclude that p is true over all of $\mathbb{R}$.*

The *strict* or *innermost-first* evaluation order used by the first group in the above example always generates the undefined value e whenver possible. The conservative extensions of the Boolean connectives immediatly propagate e, so whenever a function is applied outside its domain of definition this evaluation strategy yields undefinedness *regardless of the context in which the application appears*. In contrast, when a *non-strict* evaluation order is used, one may use the context in which the evaluation occurs to safely ignore some applications with undefined results.

Note that the predicate p from Example 8.1.6 may be rewritten as $x \geq 0 \rightarrow \sqrt{x} \geq 0$. Presented in this form p is generally believed to be true, even for negative values of x. Again, an evaluation order is used that allows for short-circuiting the decision procedure as soon as a Boolean result may be concluded. This order could be called *non-Boolean-last* as it

evaluates all expressions that yield a Boolean value and then decides the formula lazily, only computing non-Boolean values when failing to decide the formula using Boolean values. It is interesting that re-writing a partial predicate may alter its perceived validity, which illustrates that the rule of least surprise is a heuristic and does not generally offer a solution to design problems. Nevertheless, it is instructive to consider extensions of propositional logic that come closer to satisfying the expectations of people, such as the validity of p over $\mathbb{R}$.

One such extension is presented in Figure 8.3. Since the logic allows for proving formulas with partial predicates outside their domain of definition, this could lead to erroneous conclusions about the exceptional behaviour of programs. However, the precondition check VC generated for a fall to a function `f`, specified as described in Chapter 5, contains a boundedness constraint of the form $\mathtt{f}(\ldots) \in \mathbb{F}$ in its conclusion. If the call to `f` yields `e`, then the VC can only be true if one of its hypotheses is false. The only possible cases are then that the calling program's precondition, a loop invariant, the postcondition for a subprogram in a preceding call or a Boolean conditional or loop guard appearing as hypothesis in the VC is false. If the call is reachable only the first three are possible, from which it follows that either a loop VC is false or that a subprogram call or the program call itself is illegal. If either of the first two is the case, then we may repeat the previous line of reasoning and otherwise the precondition for the main program must be false for the values yielding `e`. Thus, the extended logic in Figure 8.3 is safe for proving exception freedom properties as formalised using the specifications in Chapter 5 and logical semantics from Chapter 3.

8.1.4 Partial functions continued

In Section 8.1.1 we considered approximations for extended real predicates. Our aim however is to prove correctness theorems involving partial functions and interval relations over real variables, meaning that we need to handle both undecidedness and undefinedness. We are therefore concerned with approximations $\mathcal{B}_p$ of *partial poset predicates* $p : R \to \{\mathtt{e}, \mathtt{t}, \mathtt{f}, \mathtt{u}\}$ for posets R.

Definition 8.1.7 (Flat partial Booleans)**.** *The partial order* $\lesssim$ *on* $\{\mathtt{e}, \mathtt{t}, \mathtt{f}, \mathtt{u}\}$ *determined by* $\mathtt{u} \lesssim \mathtt{e}, \mathtt{u} \lesssim \mathtt{t}$ *and* $\mathtt{u} \lesssim \mathtt{f}$ *is called the* flat partial order *on* $\{\mathtt{e}, \mathtt{t}, \mathtt{f}, \mathtt{u}\}$*. The resulting poset* $(\{\mathtt{e}, \mathtt{t}, \mathtt{f}, \mathtt{u}\}, \lesssim)$ *is called the* flat partial Boolean *poset and denoted by* $\mathbb{B}_{\mathtt{eu}}$.

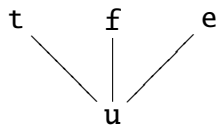


Figure 8.4: Flat partial Booleans

In Section 2.3 we equipped interval posets $\mathbb{I}(R)$ acting as domains for interval extensions of partial functions with a *bottom element* $\bot$ acting as result for application of the extension outside the domain of definition of the underlying function. We extend this idea further in analogy with the extension of $\mathbb{B}$ to $\mathbb{B}_{\mathtt{eu}}$, providing values that propagate partiality and multivaluedness information to constituent relations in the predicate:

Definition 8.1.8 (Flat partial poset)**.** *Let* $X_{\mathtt{eu}}$ *denote the set* X *extended with the* distinguished *values* $\mathtt{e}$ *and* $\mathtt{u}$*, called* erroneous *and* undetermined*, respectively. The partial order* $\lesssim$ *on* $X_{\mathtt{eu}}$ *determined by* $\mathtt{u} \lesssim \mathtt{e}$ *and* $\mathtt{u} \lesssim x$ *for each* $x \in X$ *is called the* flat partial order *on* X*. The resulting poset* $(X_{\mathtt{eu}}, \lesssim)$ *is called the* flat partial poset on X*.*

By the calling the values $\mathtt{e}$ and $\mathtt{u}$ *distinguished* we mean that they can only be added once. Consequently, the operation of taking the flat partial poset is *idempotent*, *i. e.* $(X_{\mathtt{eu}})_{\mathtt{eu}} = X_{\mathtt{eu}}$. Note that the flat order relates any value to itself and $\mathtt{u}$ *only*. In particular, the flat order $\lesssim$ on posets $(R, \leq)$ generally has no relation to the partial order $\leq$.

Partial functions from X to Y lift trivially to partial functions from $X_{\mathtt{eu}}$ to $Y_{\mathtt{eu}}$ and may then be *canonically* extended to *total* functions, *i. e.* partial functions extend canonically to total functions between the flat partial posets on their carrier sets:

Definition 8.1.9 (Flat partial function extension)**.** *Let* $\{X_i\}_{i=1}^n$ *be a family of sets,* Y *be a set and* $f : \prod_{i=1}^n (X_i)_{\mathtt{eu}} \rightharpoonup Y_{\mathtt{eu}}$ *be a partial function. The canonical total function* $f_{\mathtt{eu}} : \prod_{i=1}^n (X_i)_{\mathtt{eu}} \to Y_{\mathtt{eu}}$ *is defined by:*

$$f_{\mathtt{eu}}(a) = \begin{cases} f(a) & \text{if } f \text{ is defined at } a \\ \mathtt{u} & \text{if } f \text{ is undefined at } a \text{ and } \mathtt{u} \in \{a_1, \ldots, a_n\} \text{ and } \mathtt{e} \notin \{a_1, \ldots, a_n\} \\ \mathtt{e} & \text{otherwise} \end{cases} \tag{8.2}$$

and called the flat partial extension *of* f*.*

Taking the flat poset does not generally commute with set operations and an important counterexample is given by the powerset operation:

Definition 8.1.10 (Flat partial powerset)**.** *Let X be a set. The powerset $\wp(X_{\mathsf{eu}})$ of the flat partial poset over X is projected onto the flat poset $\wp(X)_{\mathsf{eu}}$ over the powerset of X by the surjective map $\pi^{\wp}_{\mathsf{eu}} : \wp(X_{\mathsf{eu}}) \twoheadrightarrow \wp(X)_{\mathsf{eu}}$ defined by:*

$$\pi^{\wp}_{\mathsf{eu}}(A) \;=\; \begin{cases} A & \text{if } A \subseteq X \\ \mathsf{e} & \text{if } A = \{\mathsf{e}\} \\ \mathsf{u} & \text{otherwise} \end{cases} \tag{8.3}$$

The composition of the above projection with the powerset operation is called the flat partial powerset operation *and denoted by $\wp_{\mathsf{eu}}$.*

It follows from the definition of $\wp_{\mathsf{eu}}$ that $\wp_{\mathsf{eu}}(X_{\mathsf{eu}}) = \wp(X)_{\mathsf{eu}}$. Recall that the definition of *set extensions* $f^{\wp}$ was given on page 109, for *total* functions $f : X \to Y$, as the *image* $f^{\wp}(A) = \{f(x) \in Y \mid x \in A\}$ of A under f, for $A \subseteq X$. The generalisation to set extensions $f^{\wp} : \wp(X)^n \to \wp(Y)$ of functions $f : X^n \to Y$ on *products* is defined identically, but still yields *partial* extensions for partial functions. However, partial functions lift *canonically* to *total* functions on the *flat* powersets of their carrier sets:

Definition 8.1.11 (Flat partial set extension)**.** *Let X_i, Y and $f : \prod_{i=1}^{n} (X_i)_{\mathsf{eu}} \rightharpoonup Y_{\mathsf{eu}}$ be a partial function. Then the canonical total function $f^{\wp_{\mathsf{eu}}} : \prod_{i=1}^{n} \wp(X_i)_{\mathsf{eu}} \to \wp(Y)_{\mathsf{eu}}$ is defined as the composition $f^{\wp_{\mathsf{eu}}} = \pi^{\wp}_{\mathsf{eu}} \circ (f_{\mathsf{eu}})^{\wp}$, called the* flat partial set extension *of f.*

Example 8.1.12 (Flat set extension of the logarithm)**.** *The logarithm function on real numbers $\log : \mathbb{R} \rightharpoonup \mathbb{R}$ is defined for $x \in \mathbb{R}_{>0}$. In particular it is a partial function $\mathbb{R}_{\mathsf{eu}} \rightharpoonup \mathbb{R}_{\mathsf{eu}}$ with flat partial extension $\log_{\mathsf{eu}} : \mathbb{R}_{\mathsf{eu}} \to \mathbb{R}_{\mathsf{eu}}$ given by:*

$$\log_{\mathsf{eu}}(a) \;=\; \begin{cases} \log(a) & \text{if } a > 0 \\ \mathsf{u} & \text{if } a = \mathsf{u} \\ \mathsf{e} & \text{otherwise} \end{cases}$$

The total function $\log_{\mathsf{eu}}$ has the set extension $(\log_{\mathsf{eu}})^{\wp} : \wp(\mathbb{R}_{\mathsf{eu}}) \to \wp(\mathbb{R}_{\mathsf{eu}})$ given by:

$$\begin{aligned} (\log_{\mathsf{eu}})^{\wp}(A) &= \{\log_{\mathsf{eu}}(a) \mid a \in A\} \\ &= \log^{\wp}(A \cap \mathbb{R}_{>0}) \cup \{\mathsf{e} \mid A \cap \mathbb{R}_{\leq 0} \neq \varnothing \vee \mathsf{e} \in A\} \cup \{\mathsf{u} \mid \mathsf{u} \in A\} \end{aligned}$$

and the composition $\pi^{\wp}_{\mathsf{eu}} \circ (\log_{\mathsf{eu}})^{\wp}$ yields the flat partial set extension $\log^{\wp_{\mathsf{eu}}} : \wp(\mathbb{R})_{\mathsf{eu}} \to \wp(\mathbb{R})_{\mathsf{eu}}$ of the logarithm on real numbers:

$$\log^{\wp_{\mathsf{eu}}}(A) \;=\; \begin{cases} \log^{\wp}(A) & \text{if } A \subseteq \mathbb{R}_{>0} \\ \mathsf{e} & \text{if } A \subseteq \mathbb{R}_{\leq 0} \cup \{\mathsf{e}\} \\ \mathsf{u} & \text{otherwise} \end{cases}$$

The above example shows how to compute the domain over which set extensions of partial numeric functions have numeric results. Flat set extensions over posets immediately yield flat order interval extensions and by defining general interval extensions by the corresponding equations we arrive at flat generalised interval extensions.

Using the flat ordering on $\mathbb{B}_{\mathtt{eu}}$ we order predicates pointwise as in Example 2.1.6 on page 22 and formalise the approximation of partial poset predicates in terms of isotonic functions:

Definition 8.1.13 (Safe interval approximation)**.** *Let R be a poset and $p : R^n \to \mathbb{B}_{\mathtt{eu}}$ be a function. An* isotonic *function $\mathcal{B}_p : \mathbb{I}(R)^n \to \mathbb{B}_{\mathtt{eu}}$ is called a* safe interval approximation *of p whenever:*

$$\boldsymbol{r}_i \sqsubseteq r_i \Rightarrow \mathcal{B}_p(\boldsymbol{r}) \lesssim p(r) \tag{8.4}$$

holds for each $i \in \{1, \ldots, n\}$, each $r \in \boldsymbol{r}$ and each $\boldsymbol{r} \in \mathbb{I}(R)^n$.

Property (8.4) guarantees that $\mathcal{B}_p(\boldsymbol{x})$ is $\lesssim$*-maximal* only if p is *constant* over $\boldsymbol{x}$ and that a maximal $\mathcal{B}_p(\boldsymbol{x})$ is equal to $p(x)$ for $x \in \boldsymbol{x}$. Put another way, safe extensions $\mathcal{B}_p$ *decide* p as *true, false* or *undefined* over a box only if p is *everywhere true*, *everywhere false* or *everywhere undefined* over the box.

8.1.5 Polynomial function intervals

As previously noted interval approximations cannot always decide a predicate. Often the problem is that the chosen approximations are too coarse, meaning they over-approximate ranges for expressions over a domain, forcing an exponentially costly subdivision loop.

One solution is to make the accuracy of the approximation variable. In such cases failing to decide does not immediately induce subdivision, instead the accuracy of the approximation is increased and the approximation re-computed, now standing a better chance of deciding a Boolean value for the problem. Clearly this approach will be computationally less costly, provided that increasing the accuracy has polynomial cost. The prototype prover presented in this chapter is intended to test this assumption. There are many methods for increasing the accuracy of constant interval approximations, however they all use interval arithmetic to some degree, meaning that there are bounds on the accuracy improvement imposed by wrapping effects and the dependency problem.

Our approach is to use *polynomial function intervals*, also called *polynomial function en-*

closures (PFEs), whose bounds are polynomials of bounded size with arbitrary precision coefficients, to construct safe approximations for the floating point and interval functions and relations that appear in correctness theorems for numerical programs specified in the way presented in Chapters 3–6.

Recalling the definition of floating point formats from Section 4.1 where we denoted by $\mathbb{F}_{\beta,p,e_{\min},e_{\max}}$ the format given by base β, precision p and maximum and minimum exponents $e_{\min}$ and $e_{\max}$, respectively. We specialise to the binary format $\mathbb{F}_g^{\pm\infty} = \mathbb{F}^{\pm\infty}_{2,g,2^{g+1}-1,1-2^{g+1}}$ parametrised by the single natural number g, called the *granularity* of the format.

Formally, a PFE $\boldsymbol{f}$ is a generalised function interval, bounded by polynomials in m real variables with $\mathbb{F}_g$-coefficients and degree $\leq n$, written as $\boldsymbol{f} \in \mathbb{J}(\mathbb{F}_g^{\pm\infty}[x_1, \ldots, x_m]_{\leq n})$, where $\mathbb{F}_g^{\pm\infty}[x_1, \ldots, x_m]_{\leq n} = \bigoplus_{i=0}^{n} \mathbb{F}_g^{\pm\infty}[x_1, \ldots, x_m]_i$ is set of $\mathbb{F}_g$-polynomials of degree 0 to n. The bound on the degree is necessary to limit the complexity of the arithmetic as iterated multiplications of non-constant polynomials generate a sequence of polynomials of strictly increasing degree. To enforce the bound it is necessary to interleave arithmetic operations that potentially increase the degree of the bound functions with directed roundings $\downarrow n, \uparrow n : \mathbb{F}_g^{\pm\infty}[x_1, \ldots, x_m] \to \mathbb{F}_g^{\pm\infty}[x_1, \ldots, x_m]_{\leq n}$, for $n \in \mathbb{N}$, taking polynomials f of arbitrary degree to polynomials of degree $\leq n$ that bound f from below and above: $f \downarrow n \dot{\leq} f \dot{\leq} p \uparrow f$. As we saw in Chapter 7 directed roundings on the base poset induce directed roundings on intervals over the poset, so we have directed roundings $\downarrow n, \uparrow n : \mathbb{J}(\mathbb{F}_g^{\pm\infty}[x_1, \ldots, x_m]) \to \mathbb{J}(\mathbb{F}_g^{\pm\infty}[x_1, \ldots, x_m]_{\leq n})$ taking arbitrary polynomial intervals $\boldsymbol{f}$ to intervals $\boldsymbol{f} \downarrow n$ and $\boldsymbol{f} \uparrow n$ with bounds of degree $\leq n$, approximating $\boldsymbol{f}$ from outside and inside, respectively.

The *maxdegree* parameter n and granularity g of PFEs provide a way of increasing approximation effort before subdividing that generalises the common approach of constant interval approximations with arbitrary precision coefficients. Any interval in $\mathbb{J}(\boldsymbol{x} \to \mathbb{R}^{\pm\infty})$, with continuous endpoint functions and domain $\boldsymbol{x} \in \mathbb{I}(\mathbb{R})^m$, may be approximated by a PFE in m variables and that the approximation can be made arbitrarily accurate by choosing sufficiently high degree and coefficient granularity. In particular, PFEs approximate exact real interval and continuous floating point expressions over interval boxes.

8.2 Implementation

Recall the formula sublanguage of ANOT from Section 3.2.1, used to represent proof obligations for PEA programs. Using the exact real and interval expressions provided by the language we can conveniently approximate floating point expressions appearing in the program code, the conditions under which execution of the program cannot generate numeric exceptions and accuracy properties for the resulting values. The following sections show how we approximate the exact formulas using safe numeric approximations, making it possible to automate some correctness proofs.

The implementation is a straightforward translation of the semantics of interval expressions and relations that has been developed in the preceding chapters. Floating point functions are viewed as partial real functions and approximated by intervals safely bounding the function in the interval lattice.

8.2.1 The correctness theorem language CTL

Chapters 3–5 outlined a method for specification and correctness theorem generation for algorithms expressed in the PEA language from Section 3.2.2. PEA consists of the procedural language PROC from Section 3.1.1 annotated by formulas in the ANOT language from Section 3.2.1. The denotational semantics described in Section 3.4 translates the PEA program into an ANOT formula encapsulating the correctness condition for the program. We will not address the proving of formulas with nested quantifiers, rather we will be proving propositional formulas closed by outermost universal quantification. We also restrict our attention to the functions and relations appearing in the example programs `Erf` and `Sqrt` from Sections 5.4.2 and 5.4.4. Thus, the grammar of the correctness theorems under consideration is the sub-grammar of the ANOT language given by:

$$
\begin{aligned}
\tau \quad ::= \quad & q \mid e \mid \pi \mid x \mid [\tau_1, \tau_2] \mid -\tau \mid \sqrt{\tau} \mid \tau_1 + \tau_2 \mid \tau_1 * \tau_2 \mid \tau_1/\tau_2 \mid \tau_1^{\tau_2} \mid \int_{x\in[\tau_1,\tau_2]} \tau_3 \mid \\
& \varepsilon_{abs} \mid \varepsilon_{rel} \mid \boldsymbol{\varepsilon}_{abs} \mid \boldsymbol{\varepsilon}_{rel} \mid \mathtt{exp}(\tau) \mid \tau_1 \oplus \tau_2 \mid \tau_1 \circledast \tau_2 \mid \tau_1 \oslash \tau_2 \\
\varphi \quad ::= \quad & \top \mid \bot \mid \tau_1 < \tau_2 \mid \tau_1 \leq \tau_2 \mid \tau_1 = \tau_2 \mid \tau_1 \geq \tau_2 \mid \tau_1 > \tau_2 \mid \tau_1 \in \tau_2 \mid \tau_1 \subseteq \tau_2 \mid \\
& \neg\varphi \mid \varphi_1 \wedge \varphi_2 \mid \varphi_1 \vee \varphi_2 \mid \varphi_1 \rightarrow \varphi_2
\end{aligned}
$$

where $q \in \mathbb{Q}$. Correctness theorems are formulas φ with all its free variables closed by a universal quantifier. The domain of each variable is obtained from the obligatory

declarations provided by PEA programs. Correctness theorems then have the form:

$$\forall x_1 \in \boldsymbol{x}_1 \cdots \forall x_m \in \boldsymbol{x}_m \,.\, \varphi \tag{8.5}$$

where $\{x_1, \ldots, x_m\}$ is the set of free variables in φ and $\boldsymbol{x} = (\boldsymbol{x}_1, \ldots, \boldsymbol{x}_m) \in \mathbb{I}(\mathbb{R})^m$ is an interval tuple, also called the *domain box* of the correctness theorem. We call the language of formulas of the form (8.5) the *correctness theorem language* and abbreviate it as CTL. We will also refer to CTL formulas without quantifiers as CTL' formulas. The following section describes how the framework for safe approximation of exact numerical, floating point and Boolean expressions described previously is applied to obtain safe approximations for CTL formulas.

8.2.2 Exact and approximate semantics for CTL

The *exact* semantics of the grammar given on the previous page is expressed in terms of *safe generalised function interval approximations* of predicates as defined on page 123. The exact semantics, used to approximate floating point expressions, is in turn approximated in the prover by means of *safe* PFE *approximations*, as defined in the preceding chapter. Formally, the *exact* interpretation of a formula φ is given by the map $[\![\cdot]\!] : \mathrm{CTL} \to \mathbb{B}_{\mathtt{eu}}$:

$$[\![\forall x_1 \in \boldsymbol{x}_1 \cdots \forall x_m \in \boldsymbol{x}_m \,.\, \varphi]\!] \;=\; [\![\varphi]\!]_{\boldsymbol{x}} \tag{8.6}$$

where $\boldsymbol{x} = (\boldsymbol{x}_1, \ldots, \boldsymbol{x}_m)$ and $[\![\varphi]\!]_{\boldsymbol{x}}$ is defined recursively over the grammar of CTL' as follows:

$$\begin{aligned} [\![\varphi_1 \; c \; \varphi_2]\!]_{\boldsymbol{x}} &= \mathcal{B}_c([\![\varphi_1]\!]_{\boldsymbol{x}}, [\![\varphi_2]\!]_{\boldsymbol{x}}) \\ [\![\neg\varphi]\!]_{\boldsymbol{x}} &= \mathcal{B}_\neg([\![\varphi]\!]_{\boldsymbol{x}}) \end{aligned} \tag{8.7}$$

for Boolean connectives $c \in \{\wedge, \vee, \rightarrow\}$ with $\mathcal{B}_\neg$ and $\mathcal{B}_c$ given by the $\{\mathtt{e}, \mathtt{t}, \mathtt{f}, \mathtt{u}\}$-extensions of $\neg$ and c in Figure 8.3 on page 119, respectively. Continuing, the *containment* relation $\subseteq$ and the *thin containment* relation $\in$ both have *exact* semantics $\mathcal{B}_\subseteq = \mathcal{B}_\in$ given by the *reverse refinement* relation $\sqsupseteq$ on *flat partial generalised extended real function intervals*:

$$[\![\tau_1 \subseteq \tau_2]\!]_{\boldsymbol{x}} \;=\; [\![\tau_1 \in \tau_2]\!]_{\boldsymbol{x}} \;=\; [\![\tau_2]\!]_{\boldsymbol{x}} \sqsubseteq [\![\tau_1]\!]_{\boldsymbol{x}} \tag{8.8}$$

and are *approximated* in terms of flat PFEs by:

$$[\![\tau_2]\!]_{\boldsymbol{x}} \sqsubseteq [\![\tau_1]\!]_{\boldsymbol{x}} \;\gtrsim\; \begin{cases} \mathtt{t} & \text{if } [\![\tau_2]\!]_{\boldsymbol{x}}^{\mathrm{PFE}\uparrow} \sqsubseteq^{\mathrm{PFE}} [\![\tau_1]\!]_{\boldsymbol{x}}^{\mathrm{PFE}\downarrow} \\ \mathtt{f} & \text{if } [\![\tau_2]\!]_{\boldsymbol{x}}^{\mathrm{PFE}\downarrow} \not\sqsubseteq^{\mathrm{PFE}} [\![\tau_1]\!]_{\boldsymbol{x}}^{\mathrm{PFE}\uparrow} \\ \mathtt{e} & \text{if } \mathtt{e} \in \left\{ [\![\tau_1]\!]_{\boldsymbol{x}}^{\mathrm{PFE}\uparrow}, [\![\tau_1]\!]_{\boldsymbol{x}}^{\mathrm{PFE}\downarrow}, [\![\tau_2]\!]_{\boldsymbol{x}}^{\mathrm{PFE}\uparrow}, [\![\tau_2]\!]_{\boldsymbol{x}}^{\mathrm{PFE}\downarrow} \right\} \\ \mathtt{u} & \text{otherwise} \end{cases} \tag{8.9}$$

where the flat inner and outer PFE semantics $[\![\tau]\!]_{\boldsymbol{x}}^{\mathrm{PFE}\uparrow}$ and $[\![\tau]\!]_{\boldsymbol{x}}^{\mathrm{PFE}\downarrow}$ of terms are described below and $\sqsubseteq^{\mathrm{PFE}}$ and $\not\sqsubseteq^{\mathrm{PFE}}$ are safe PFE approximations of $\sqsubseteq$ and $\not\sqsubseteq$, respectively. Order relations $r \in \{<, \leq, =, \geq, >\}$ and Boolean literals have *exact* semantics $\mathcal{B}_r, \mathcal{B}_\bot$ and $\mathcal{B}_\top$ given by:

$$\begin{aligned} [\![\tau_1\ r\ \tau_2]\!]_{\boldsymbol{x}} &= [\![\tau_1]\!]_{\boldsymbol{x}}\ r_{\mathbb{I}}\ [\![\tau_2]\!]_{\boldsymbol{x}} \\ [\![\bot]\!]_{\boldsymbol{x}} &= \mathtt{f} \\ [\![\top]\!]_{\boldsymbol{x}} &= \mathtt{t} \end{aligned} \tag{8.10}$$

where the interval relation $r_{\mathbb{I}}$ induced by r is *approximated* by:

$$[\![\tau_1]\!]_{\boldsymbol{x}}\ r_{\mathbb{I}}\ [\![\tau_2]\!]_{\boldsymbol{x}} \;\gtrsim\; \begin{cases} \mathtt{t} & \text{if } [\![\tau_1]\!]_{\boldsymbol{x}}^{\mathrm{PFE}\downarrow}\ r_{\mathbb{I}}^{\mathrm{PFE}}\ [\![\tau_2]\!]_{\boldsymbol{x}}^{\mathrm{PFE}\downarrow} \\ \mathtt{f} & \text{if } [\![\tau_1]\!]_{\boldsymbol{x}}^{\mathrm{PFE}\downarrow}\ (\neg r)_{\mathbb{I}}^{\mathrm{PFE}}\ [\![\tau_2]\!]_{\boldsymbol{x}}^{\mathrm{PFE}\downarrow} \\ \mathtt{e} & \text{if } \mathtt{e} \in \left\{ [\![\tau_1]\!]_{\boldsymbol{x}}^{\mathrm{PFE}\downarrow}, [\![\tau_2]\!]_{\boldsymbol{x}}^{\mathrm{PFE}\downarrow} \right\} \\ \mathtt{u} & \text{otherwise} \end{cases} \tag{8.11}$$

where $r_{\mathbb{I}}^{\mathrm{PFE}}$ is a safe flat PFE approximation of the interval relation $r_{\mathbb{I}}$ and $\neg r$ is the *strict opposite* version of the exact relation r, with *e.g.* $\neg(\leq)$ being $>$.

The refinement and interval order relation approximations in (8.9) and (8.11) use the *flat outer* PFE *semantics* $[\![\tau]\!]_{\boldsymbol{x}}^{\mathrm{PFE}\downarrow}$. The *exact* outer semantics $\mathcal{B}_{\circledcirc}^{\downarrow}$ and $\mathcal{B}_{\mathtt{exp}}^{\downarrow}$ of *floating point* operations and functions is given by the generalised outward rounding operator $\mathcal{S}_{out}$ from Section 5.1. It approximates floating point expressions from below by pessimistically assuming that maximal rounding errors arise from each constituent floating point function. The exact outer semantics $[\![\tau]\!]_{\boldsymbol{x}}^{\downarrow}$ is *approximated* from below by the flat outer PFE semantics $[\![\tau]\!]_{\boldsymbol{x}}^{\mathrm{PFE}\downarrow}$ defined for floating point terms τ by:

$$\begin{aligned} [\![\tau_1 \circledcirc \tau_2]\!]_{\boldsymbol{x}}^{\mathrm{PFE}\downarrow} &= \mathcal{B}_{\circledcirc}^{\mathrm{PFE}\downarrow}([\![\tau_1]\!]_{\boldsymbol{x}}^{\mathrm{PFE}\downarrow}, [\![\tau_1]\!]_{\boldsymbol{x}}^{\mathrm{PFE}\downarrow}) \\ [\![\mathtt{exp}(\tau)]\!]_{\boldsymbol{x}}^{\mathrm{PFE}\downarrow} &= \mathcal{B}_{\mathtt{exp}}^{\mathrm{PFE}\downarrow}([\![\tau]\!]_{\boldsymbol{x}}^{\mathrm{PFE}\downarrow}) \\ [\![\boldsymbol{\varepsilon}]\!]_{\boldsymbol{x}}^{\mathrm{PFE}\downarrow} &= [\lambda x. -\varepsilon, \lambda x. \varepsilon] \\ [\![\varepsilon]\!]_{\boldsymbol{x}}^{\mathrm{PFE}\downarrow} &= \lambda x. \varepsilon \end{aligned} \tag{8.12}$$

where $\circledcirc \in \{\oplus, \circledast, \oslash\}$, $\boldsymbol{\varepsilon} \in \{\boldsymbol{\varepsilon}_{abs}, \boldsymbol{\varepsilon}_{rel}\}$ and $\varepsilon \in \{\varepsilon_{abs}, \varepsilon_{rel}\}$. $\mathcal{B}_{\circledcirc}^{\text{PFE}\downarrow}$ and $\mathcal{B}_{\texttt{exp}}^{\text{PFE}\downarrow}$ are given in terms of the natural flat outer PFE extension $\mathcal{S}_{out}^{\text{PFE}\downarrow}$ of the exact operator $\mathcal{S}_{out}$:

$$\begin{aligned}
\mathcal{B}_{\circledcirc}^{\text{PFE}\downarrow}(\boldsymbol{f}_1, \boldsymbol{f}_2) &= \mathcal{S}_{out}^{\text{PFE}\downarrow}\Big([\![\varepsilon_{\circledcirc}]\!]_{\boldsymbol{x}}^{\text{PFE}\downarrow}, \boldsymbol{f}_1 \circ^{\text{PFE}\downarrow} \boldsymbol{f}_2 \Big) \\
&= \mathcal{R}_{out}^{\text{PFE}\downarrow}\Big((1 +^{\text{PFE}\downarrow} [\![\varepsilon_{\circledcirc}]\!]_{\boldsymbol{x}}^{\text{PFE}\downarrow} *^{\text{PFE}\downarrow} [-1, 1]) *^{\text{PFE}\downarrow} (\boldsymbol{f}_1 \circ^{\text{PFE}\downarrow} \boldsymbol{f}_2) \Big) \\
&= (1 +^{\text{PFE}\downarrow} [-\varepsilon_{rel}, \varepsilon_{rel}]) *^{\text{PFE}\downarrow} \Big((1 +^{\text{PFE}\downarrow} [\![\varepsilon_{\circledcirc}]\!]_{\boldsymbol{x}}^{\text{PFE}\downarrow} *^{\text{PFE}\downarrow} [-1, 1]) \\
&\quad *^{\text{PFE}\downarrow} (\boldsymbol{f}_1 \circ^{\text{PFE}\downarrow} \boldsymbol{f}_2) \Big) +^{\text{PFE}\downarrow} [-\varepsilon_{abs}, \varepsilon_{abs}]
\end{aligned} \tag{8.13}$$

where $\circ \in \{+, *, /\}$ is the exact operation corresponding to $\circledcirc$ and $\circ^{\text{PFE}\downarrow}$ is an *isotonic outer generalised* PFE *extension* of $\circ$ in the sense of Chapters 2 and 7 and:

$$\mathcal{B}_{\texttt{exp}}^{\text{PFE}\downarrow}(\boldsymbol{f}) = \mathcal{S}_{out}^{\text{PFE}\downarrow}\Big([\![\varepsilon_{\texttt{exp}}]\!]_{\boldsymbol{x}}^{\text{PFE}\downarrow}, \mathcal{B}_{e}^{\text{PFE}\downarrow}(\boldsymbol{f}) \Big) \tag{8.14}$$

where $\mathcal{B}_{e}^{\text{PFE}\downarrow}$ is an isotonic outer generalised interval extension of the extended real exponential function. In particular, the exact outer semantics approximates *floating point functions* over an interval $\boldsymbol{x}$ with a function interval in $\mathbb{J}(\boldsymbol{x} \to \mathbb{R}^{\pm\infty})$.

The *exact* interpretation $\mathcal{B}_{\gamma}$ of *exact* functions γ is given by the corresponding interval function and *approximated* in terms of PFE extensions:

$$\begin{aligned}
[\![\gamma(\tau_1, \ldots, \tau_k)]\!]_{\boldsymbol{x}}^{\text{PFE}\downarrow} &= \mathcal{B}_{\gamma}^{\text{PFE}\downarrow}([\![\tau_1]\!]_{\boldsymbol{x}}^{\text{PFE}\downarrow}, \ldots, [\![\tau_k]\!]_{\boldsymbol{x}}^{\text{PFE}\downarrow}) \\
\Big[\!\Big[\textstyle\int_{x' \in [\tau_1, \tau_2]} \tau_3 \Big]\!\Big]_{\boldsymbol{x}}^{\text{PFE}\downarrow} &= \mathcal{B}_{\int}^{\text{PFE}\downarrow}([\![\tau_3]\!]_{\boldsymbol{x}'}^{\text{PFE}\downarrow}) \circ_{i_{x'}}^{\text{PFE}\downarrow} [\![\tau_2]\!]_{\boldsymbol{x}}^{\text{PFE}\downarrow} -^{\text{PFE}\downarrow} \\
&\quad \mathcal{B}_{\int}^{\text{PFE}\downarrow}([\![\tau_3]\!]_{\boldsymbol{x}'}^{\text{PFE}\downarrow}) \circ_{i_{x'}}^{\text{PFE}\downarrow} [\![\tau_1]\!]_{\boldsymbol{x}}^{\text{PFE}\downarrow} \\
[\![[\tau_1, \tau_2]]\!]_{\boldsymbol{x}}^{\text{PFE}\downarrow} &= \left[\underline{[\![\tau_1]\!]_{\boldsymbol{x}}^{\text{PFE}\downarrow}}, \overline{[\![\tau_2]\!]_{\boldsymbol{x}}^{\text{PFE}\downarrow}} \right] \\
[\![x]\!]_{\boldsymbol{x}}^{\text{PFE}\downarrow} &= \text{proj}_{i_{\boldsymbol{x}}}(\boldsymbol{x})
\end{aligned} \tag{8.15}$$

where $\underline{[a, b]} = a$, $\overline{[a, b]} = b$ and $\boldsymbol{x}'$ is the domain box $\boldsymbol{x}$ extended by an interval for the variable x' such that:

$$\text{proj}_{i_{x'}}(\boldsymbol{x}') = [\![[\tau_1, \tau_2]]\!]_{\boldsymbol{x}}^{\text{PFE}\downarrow} \downarrow 0 \tag{8.16}$$

and $\circ_{i_{x'}}^{\text{PFE}\downarrow}$ denotes *outer* PFE *approximation* of *composition* in the variable $i_{x'}$. The outer approximations $\mathcal{B}_{\gamma}^{\text{PFE}\downarrow}$ are given by isotonic flat outer PFE extensions of the possibly partial exact functions $\gamma \in \{-, \sqrt{}, +, *, /, \hat{}, \int\}$, where $\hat{}$ denotes the binary exponentiation operator. The *exact* semantics of exact *literals* $l \in \{e, \pi\} \cup \mathbb{Q}$ is given by the embedding of real numbers within function intervals, $[\![l]\!]_{\boldsymbol{x}} = [\lambda x.l, \lambda x.l]$, and *approximated* by *outward rounding*:

$$[\![l]\!]_{\boldsymbol{x}}^{\text{PFE}\downarrow} = [\lambda x.l\downarrow, \lambda x.l\uparrow] \tag{8.17}$$

Note that this is in fact how the floating point epsilons were handled in (8.12), but since they are *representable*, the resulting interval is *thin*.

The containment relation approximations in (8.9) use both the outer PFE semantics $[\![\tau]\!]_{\boldsymbol{x}}^{\text{PFE}\downarrow}$ and the *inner* PFE semantics $[\![\tau]\!]_{\boldsymbol{x}}^{\text{PFE}\uparrow}$ of terms. The latter is defined *dually* to $[\![\tau]\!]_{\boldsymbol{x}}^{\text{PFE}\downarrow}$ and may be obtained from (8.12)–(8.17) by:

- reversing left and right interval bounds, *i. e.* $\underline{x}$ becomes $\overline{x}$ and vice versa, in the interval constructor equation in (8.15)
- reversing all downward arrows to upward arrows, except those in (8.16)
- replacing the isotonic outer generalised interval extensions with *isotonic inner generalised interval extensions*
- using the *conjugate* of the *outward* semantics for interval floating point literals:

$$[\![\varepsilon]\!]_{\boldsymbol{x}}^{\uparrow} \quad = \quad [\lambda x.\varepsilon, \lambda x. -\varepsilon] \tag{8.18}$$

- and replacing generalised outward rounding $\mathcal{S}_{out}$ by *generalised inward rounding*:

$$\mathcal{S}_{in}(e, x) \;=\; \mathcal{R}_{in}\left((1 + e[1, -1])x\right) \tag{8.19}$$

where $\mathcal{S}_{in}$ is defined in terms of the inward rounding operator $\mathcal{R}_{in}$ given by:

$$\mathcal{R}_{in}(x) \;=\; (1 + [\varepsilon_{rel}, -\varepsilon_{rel}])x + [\varepsilon_{abs}, -\varepsilon_{abs}] \tag{8.20}$$

8.2.3 Note on the approximation of $\int_a^b f(x)dx$

Equation (8.15) describes how outer PFE approximations of definite integrals are obtained. First, the integration variable domain is approximated by *constant outer* approximation of the interval given by outer approximations of the integration bound expressions, as described by (8.16), which guarantees that the PFE approximation of the primitive function will be safe. The domain box $\boldsymbol{x}$ is then extended by the domain of the integration variable to the box $\boldsymbol{x}'$ and an outer PFE approximation of the primitive function of the integrand expression is constructed over $\boldsymbol{x}'$. Finally, the resulting enclosure is composed in the integration variable with outer approximations of the bound expressions and the safe difference is taken. Note that while the outer approximation of the integral is obtained using outer approximations of the constituent expressions, an *inner* approximation

of the integral still approximates the integration domain from below, using *outer* approximations of the bound expressions, while composing *inner* approximations of the primitive function and the bound expressions.

Currently, the implementation only supports integration of expressions with *monotone* primitive functions, due to the same restriction on the composition operator.

The implementation also provides a parameter for the integration operation, called *integration depth*, intended to provide a means of increasing approximation precision for computational effort. If the integration depth is set to n, then the integration variable domain is bisected n times and approximations of the integrand expression are computed over each sub-interval, effectively yielding a *piece-wise* PFE approximation over the original domain. The piece-wise approximations are then safely combined over the range of each integration bound, hopefully resulting in tighter approximations of the bounds, and thus of the definite integral.

In the following section we investigate how solving times for the functional correctness VC of the `erf` example program are affected by enclosure degree and integration depth. The expectation is that increasing either parameter should initially decrease solving times as improved precision allows earlier decisions and then increase as redundant precision no longer allows earlier decisions while incurring additional cost. Eventually, increasing either parameter should incur precision loss, due to superfluous rounded operations, yielding no improvement in precision at increased computational cost.

8.3 Experiments

8.3.1 Motivation

The task our work set out to address is the automation of floating point verification. We have shown how to specify accuracy properties for floating point programs and how to generate proof obligations for programs specified in this way. Further, we have described the various pitfalls facing algorithms automating the proof of such proof obligations, referring to these obstructions collectively as the *information loss problem*. We proposed a way to reduce information loss in algorithms combining interval evaluation with domain subdivision by increasing the information stored in the intermediate values during eval-

uation. Our solution uses *polynomial bounds* in intervals to store approximations to the dependency relations between variables and to reduce information loss due to wrapping.

The hypothesis is that the computational overheads that come from computing with polynomials over floating point numbers, rather than just with floating point numbers, are sufficiently outweighed by increased approximation accuracy, leading to earlier decision of relations, and thereby to a reduction in domain splitting. Since the main cost of subdivision based algorithms comes from the exponential cost of splitting, we conjecture that problems where the boundaries of relations lie sufficiently close to force a state explosion in algorithms using traditional interval evaluation, our higher precision arithmetic can succeed. The hypothesis was tested by using a variable precision arithmetic with polynomial degree as one of the parameters. This allowed us to compare proof attempts using higher degree polynomial bound evaluation and evaluation with constant polynomial bounds, *i. e.* using traditional intervals. We could also see the effect of increasing polynomial degree on proof effort and identified, within a given range, the optimal degree for the given problem.

8.3.2 Experimental setup

The experiments were conducted to probe the assumption that VCs for floating point properties require higher degree enclosures in order to minimise approximation error during solving. The argument is that since precision loss in the arithmetic may force subdivision, it follows that precision enhancement at sub-exponential cost may delay subdivision and yield overall reduced solving effort.

We have not formally proved that the four-valued logic from Figure 8.3 is suitable for discharging the exception freedom proof obligations in the experiments below. However, since all predicates appearing in the examples are total over the prescribed domains, the logic used by the prover is in effect the usual three-valued logic from Figure 8.1, which is perfectly safe to use in this context.

The correctness theorems generated for the `erf` example program in Figure 5.1 on page 72 and the `square_root` example program in Figure 5.3 on page 78 are shown on page 74 and on page 80 respectively. The CTs were simplified by iterated application of $\phi_1 \rightarrow (\phi_2 \wedge \phi_3) \Rightarrow (\phi_1 \rightarrow \phi_2) \wedge (\phi_1 \rightarrow \phi_3)$ and $\phi_1 \rightarrow (\phi_2 \rightarrow \phi_3) \Rightarrow (\phi_1 \wedge \phi_2) \rightarrow \phi_3$ and replacing of equal expressions. The resulting sets of verification conditions are shown

in Section 8.3.3 and 8.3.6 respectively and correspond one-to-one with verification conditions generated by the Examiner and Simplifier tools for the corresponding SPARK implementation of `erf` on page 91 and of `square_root` on page 94.

The prover attempts to *decide* the VC over the prescribed domain, meaning that it can identify counterexamples if the VC is false over an entire sub-box of its domain. To test this functionality we modify the specification of the `erf` program by inverting the signs of the additive constant 0.00005 in the postcondition, obtaining (8.35) a false version of the functional correctness VC (8.34).

Timing data for each proof obligation was generated on an Intel Core2 Duo 1.86 GHz machine with 3GB RAM and 6MB cache, running Kubuntu 10.10, and timeout set to one hour. Proving would also be halted when attempting to split an interval with neighbouring maximum granularity numbers as endpoints. The prover was compiled with the Glasgow Haskell Compiler (GHC) version 6.12.3. The results are presented and discussed in Section 8.3.4

Solving was timed for enclosure degrees between 0 and 16 and the results presented below indicate that increasing the degree of PFEs can help improve the overall performance of subdivision based solving. Note however that the presented experiments are only sufficient to give an *indication* of performance as a function of enclosure degree. A systematic study for a suite of elementary problems will be needed before any general conclusions can be deduced about the real life performance of the system. To further make claims about the performance of the approach for *floating point proof obligations* it is essential to obtain a suite of industrial code examples of sufficient size to substantiate basic statistical methods. Then it may be possible to deduce approximations of the amortized performance of the algorithm.

8.3.3 `erf` correctness theorem

The proof obligations below were derived from the correctness theorem shown on page 74, with the variable x ranging over the interval [0,4]:

$$0.47047 \circledast x \in \mathbb{F} \tag{8.21}$$

$$1.0 \oplus (0.47047 \circledast x) \in \mathbb{F} \tag{8.22}$$

to make the formulas more readable we let $T = 1.0 \oplus (0.47047 \circledast x)$ below:

$$T \circledast T \in \mathbb{F} \tag{8.23}$$

$$T \circledast (T \circledast T) \in \mathbb{F} \tag{8.24}$$

$$x \circledast x \in \mathbb{F} \tag{8.25}$$

$$\texttt{exp}(-x \circledast x) \in \mathbb{F} \tag{8.26}$$

$$0.3480242 \oslash T \in \mathbb{F} \tag{8.27}$$

$$0.0958798 \oslash (T \circledast T) \in \mathbb{F} \tag{8.28}$$

$$0.3480242 \oslash T \ominus 0.0958798 \oslash (T \circledast T) \in \mathbb{F} \tag{8.29}$$

$$0.7478556 \oslash (T \circledast (T \circledast T)) \in \mathbb{F} \tag{8.30}$$

$$0.3480242 \oslash T \ominus 0.0958798 \oslash (T \circledast T) \oplus 0.7478556 \oslash (T \circledast (T \circledast T)) \in \mathbb{F} \tag{8.31}$$

we also let $S = 0.3480242 \oslash T \ominus 0.0958798 \oslash (T \circledast T) \oplus 0.7478556 \oslash (T \circledast (T \circledast T))$ below:

$$\texttt{exp}(-x \circledast x) \circledast S \in \mathbb{F} \tag{8.32}$$

$$1.0 \ominus \texttt{exp}(-x \circledast x) \circledast S \in \mathbb{F} \tag{8.33}$$

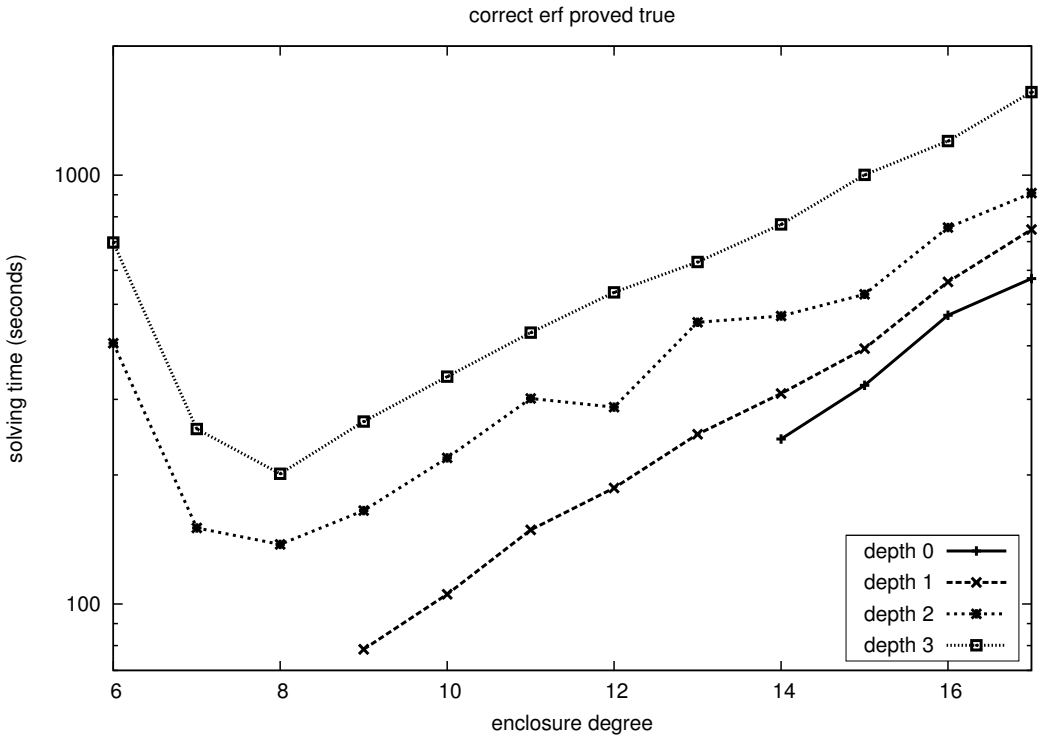
$$\frac{2}{\sqrt{\pi}} \int_{t \in [0,x]} e^{-t^2} - 0.00005 \leq 1.0 \ominus \texttt{exp}(-x \circledast x) \circledast S \leq \frac{2}{\sqrt{\pi}} \int_{t \in [0,x]} e^{-t^2} + 0.00005 \tag{8.34}$$

where $a \leq b \leq c$ abbreviates $a \leq b \wedge b \leq c$.

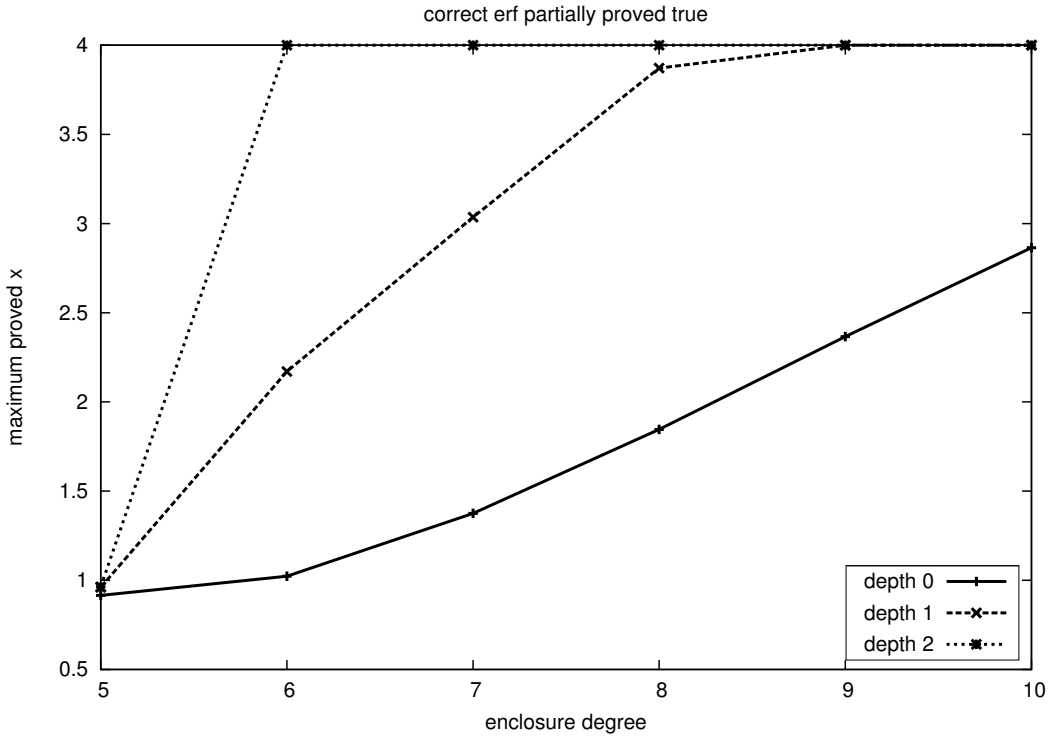
8.3.4 `erf` proving results

The thirteen exception freedom VCs in (8.21)–(8.33) were all proved within 10 seconds for all enclosure degrees. Minimal solving times were obtained for degree 0 enclosures, for which all exception freedom VCs were proved within 10 ms. Higher degrees yielded monotonically increasing solving times, indicating that the the additional precision and computational cost of higher degree enclosures was wasted.

The solving times for the functional correctness VC (8.34) are presented in Figures 8.5a and 8.5b. The missing data points in Figure 8.5a for integration depths 0 and 1 correspond to the solving being terminated due to attempted splitting of an interval with neighbouring floating point endpoints, indicating that a prohibitive number of sub-problems may have to be evaluated to decide the VC over the initial domain. We conclude that for these



(a) Solving time, as function of maximum enclosure degree, for integration depths 0 to 3.



(b) Maximum proved x, as function of maximum enclosure degree, for integration depths 0 to 2.

Figure 8.5: Proving the true `erf` functional correctness VC.

enclosure degrees the approximation precision is insufficient, forcing splitting and leading to exponential growth of the sub-problem queue. We used depth-first search to reach these atomic intervals as breadth-first search results in timeout.

Figure 8.5a presents the solving efforts for enclosure degrees 5 to 10 and integration depths 0 to 2. Plotting the right bound x of the domain $[0, x]$ for which the prover succeeds in proving the VC as a function of enclosure degree clearly shows that the fraction of the domain for which the prover succeeds in proving the VC increases monotonically with enclosure degree and integration depth.

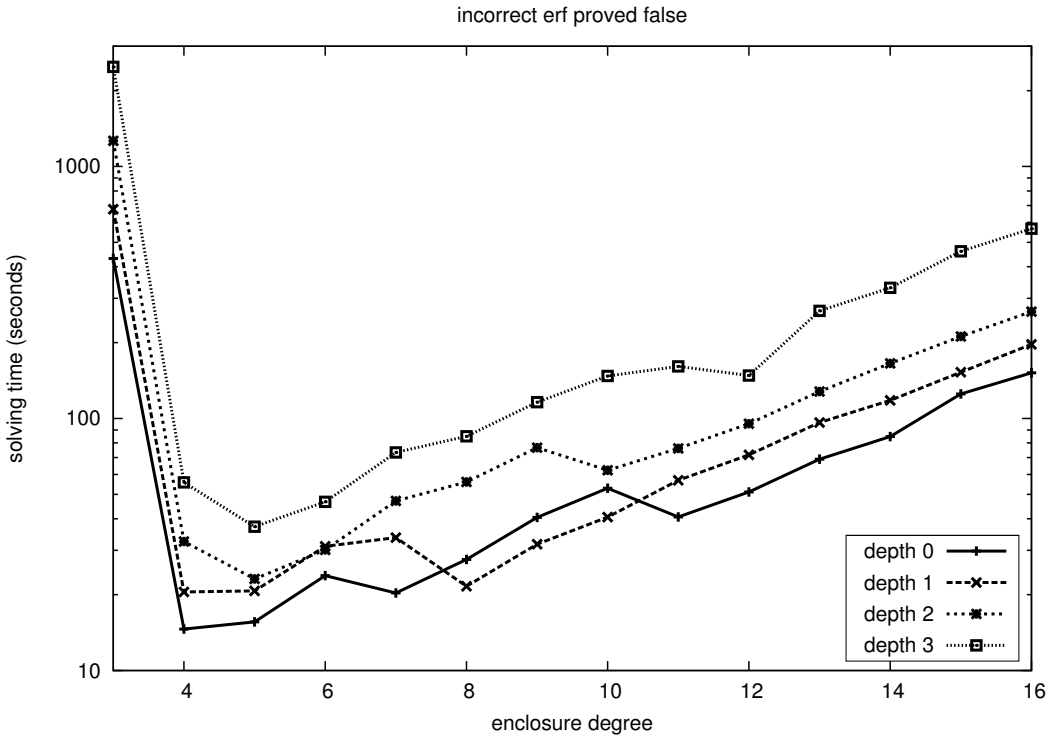
As expected, the timing graphs have a roughly convex shape. The shape is explained in terms of two main sources of computational cost: average evaluation time per sub-problem and the number of computed sub-problems. Increasing enclosure degree leads to increased computational effort and, at least initially, to improved precision. Improved precision leads to earlier decisions, reducing the total amount of generated sub-problems. As long as the number of generated sub-problems decreases faster than the average per-problem computation time, as functions of enclosure degree, then the solving time graph should decrease as maximum enclosure degree increases. Eventually, an optimal enclosure degree should be reached, after which additional degrees add comparatively little precision, while incurring additional computational cost, meaning the graph should level off and then start increasing.

The graphs in Figure 8.5a roughly follow the above prediction. The exponential cost of precision improvement through domain subdivision is dramatically reduced by increasing enclosure degree, reaching a minimum after which increased computational effort is wasted. We see that increasing the integration depth can have a beneficial effect, reducing the minimum enclosure degree for which the minimum solving time is achieved.

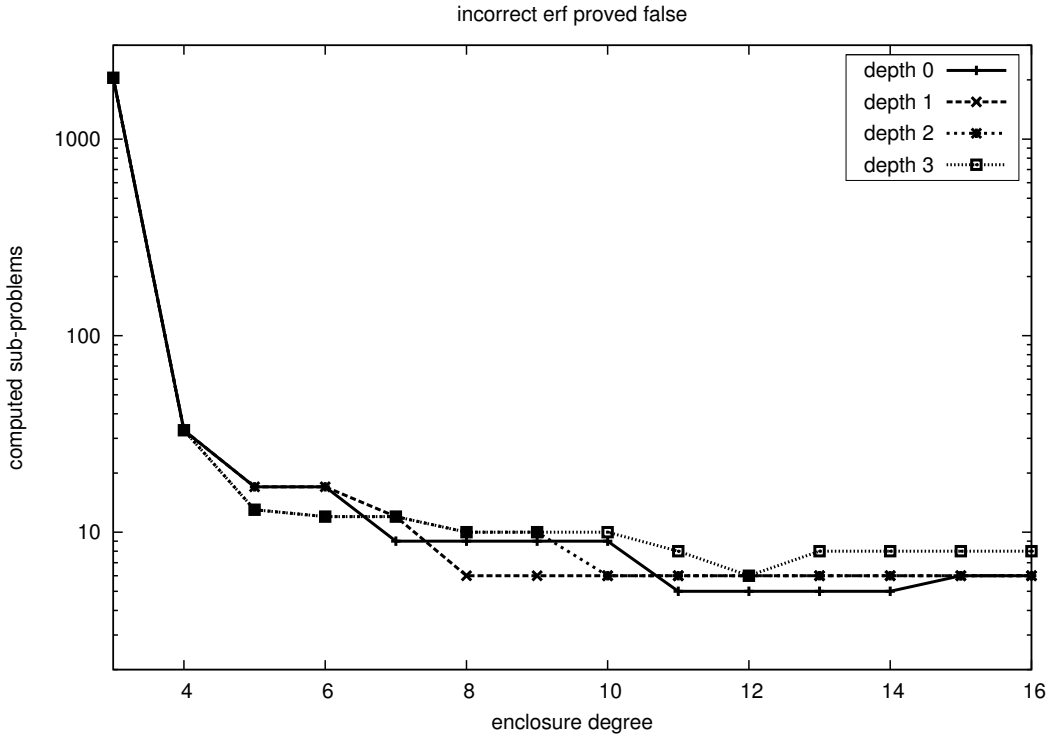
The conclusion drawn from the above experiment is that higher degree polynomial enclosures can improve the overall performance of a bisection search algorithm, at least when compared to naive arithmetic using constant and low degree enclosures.

8.3.5 `erf` counterexample discovery

To evaluate how the prover performs at counterexample discovery, the postcondition was changed in a way modelling what was felt as a reasonable implementation mistake,



(a) Solving time as function of maximum enclosure degree.



(b) The number of computed `Erf` sub-problems as function of maximum enclosure degree.

Figure 8.6: Solving the false `Erf` functional correctness VC until counterexample discovery, for integration depths 0 to 3.

namely the reversal of signs in a formula. The resulting false version of the functional correctness VC (8.34) is shown below.

$$\frac{2}{\sqrt{\pi}} \int_{t\in[0,x]} e^{-t^2} + 0.00005 \leq 1.0 \ominus \texttt{exp}(-x \circledast x) \circledast S \leq \frac{2}{\sqrt{\pi}} \int_{t\in[0,x]} e^{-t^2} - 0.00005 \tag{8.35}$$

The solving results for (8.35) are presented in Figure 8.6 on the preceding page. Figure 8.6a and Figure 8.6b show solving time and the number of computed sub-problems, respectively, as functions of enclosure degree. The solving time graphs in Figure 8.6a are similar to the ones in Figure 8.5a but shifted towards lower degrees. This is to be expected, since a counterexample can be identified anywhere, while proving requires searching the entire domain. Since the integral is initially near constant it should follow that lower degree enclosures suffice to approximate the integral closely enough.

The shapes of the graphs are again roughly convex, reflecting the characteristics of the prover algorithm as described in the discussion of the proof of the correct `erf` functional correctness VC above.

Graphs of the number of computed sub-problems until counterexample identification are presented in Figure 8.6b. Since the number of generated sub-problems depends on the approximation precision, we can interpret the graphs as showing the initial benefit of increasing approximation effort and with it precision. As approximation effort is increased further, there is a decrease in the improvement, seemingly tending asymptotically to zero. At this point added approximation effort yields little or no improvement and the additional effort translates to computational overhead.

8.3.6 `square_root` correctness theorem

The proof obligations below were derived from the correctness theorem on page 80, with the variable x ranging over the interval [0,1] and the variable r ranging over $\mathbb{F}$:

$$x \neq 0 \rightarrow x > 0 \tag{8.36}$$

$$x = 0 \rightarrow x \in (1 + 4\boldsymbol{\varepsilon}_{rel})\sqrt{x} + \boldsymbol{\varepsilon}_{abs} \tag{8.37}$$

$$r > 0 \rightarrow x \oslash r \in \mathbb{F} \tag{8.38}$$

$$r > 0 \rightarrow r \oplus (x \oslash r) \in \mathbb{F} \tag{8.39}$$

$$r > 0 \rightarrow 0.5 \circledast (r \oplus (x \oslash r)) \in \mathbb{F} \tag{8.40}$$

$$r > 0 \wedge 0.5 \circledast (r \oplus (x \oslash r)) \neq r \rightarrow 0.5 \circledast (r \oplus (x \oslash r)) > 0 \tag{8.41}$$

$$r > 0 \wedge 0.5 \circledast (r \oplus (x \oslash r)) = r \rightarrow 0.5 \circledast (r \oplus (x \oslash r)) \in (1 + 4\boldsymbol{\varepsilon}_{rel})\sqrt{x} + \boldsymbol{\varepsilon}_{abs} \tag{8.42}$$

Note that the boundedness checks appearing as conclusions for the exception freedom VCs in (8.38)–(8.40) also appear as hypotheses in each following VC in conjunction with the hypothesis $r > 0$ of (8.38). We therefore omit these checks from the hypotheses of the VCs in (8.39)–(8.42) to improve readability.

8.3.7 Revised `square_root` program

The VCs obtained for the `square_root` example program can not all be proved using the current prover. In fact, they are not even all true. The exception freedom VC in (8.38) states that the result of dividing an x between 0 and 1 by *any* positive floating point number r results in a number bounded by $\mathbb{F}_{max}$. We have however not included any *concrete* lower bound and we can *e.g.* assume $x = 1$ and $r < {}^{1}/_{\mathbb{F}_{max}}$, from which it follows that ${}^{x}/_{r} > \mathbb{F}_{max}$.

To obtain a correct program it is necessary to change the code in Figure 5.3 on page 78 so that the resulting VCs are true. Furthermore, we wish to produce VCs that are *provable* using the *approximate* and *sound* techniques employed in the prototype prover.

In our case we have chosen to change the range for x in the precondition in order to eliminate the troublesome cases for x near 0. This is perfectly reasonable, since any call to `square_root` may be preceded with a conditioning step where the values in the original range for x are multiplied with an appropriate power of 4, resulting in a call to `square_root` with $2^{2k}x$ rather than x, for some integer k. The result will then lie near $2^k\sqrt{x}$, which is multiplied with 2^{-k} to obtain the sought approximation of $\sqrt{x}$. Note that when using floating point arithmetic in base b multiplication and division with powers of b is exact, given that no overflow or underflow occurs. Therefore the conditioning and normalisation steps introduce the least possible error.

Changing the input range for the program eliminates potential denormalised values for x, which makes it possible to tighten the postcondition slightly to $r \in (1 + 4\boldsymbol{\varepsilon}_{rel})\sqrt{x}$.

In order to propagate the positivity property for r through successive iterations of the loop we need to provide a loop invariant encoding this information. As a first idea we may change $r > 0$ for $r \in [x, 1]$. Doing this we would however introduce a less obvious prob-

```
proc square_root(in x ∈ 𝔽; out r ∈ 𝔽)
    pre   x ∈ [0.5, 2]
    post  r ∈ (1 + 4ε_rel) √x
is  ...
    s ∈ 𝔽
begin
    s := x;
    r := 0.5 ⊛ x ⊕ 0.5;
    while r ≠ s assert r ∈ [−x²/4 + x, x²/4 + 1] do
        s := r;
        r := 0.5 ⊛ (s ⊕ (x ⊘ s))
    od
end
```

Figure 8.7: Revised implementation of the square root function in PEA specified using the new precondition and loop invariant.

lem, namely that of placing the fixpoint of the loop body given by $x = 1$ and $r = 1$ on the boundary of the loop invariant. The problem with this is that VCs for the path around the loop require us to show that states satisfying the loop invariant are mapped to states that satisfy the loop invariant. To prove this for a fixpoint it would require using *exact* reasoning, which we have traded for automation by employing *approximate* techniques. The solution is to use a loop invariant that has no fixpoints on its boundary. A sufficient condition is that the support of the invariant is mapped *properly* into itself by the loop body. One such region is given by r that are bounded from above by $\frac{x^2}{4} + 1$ and from below by $-\frac{x^2}{4} + x$, yielding the loop invariant $r \in [-\frac{x^2}{4} + x, \frac{x^2}{4} + 1]$.

Our final concern is to make the initial values of r and s validate the post condition whenever they do the loop exit condition and that they otherwise validate the loop invariant. The solution is to change the initial value of s to x and that of r to $0.5 \circledast x \oplus 0.5$, effectively unrolling the loop one iteration. The revised PEA program is shown in Figure 8.7 and the resulting proof obligations are presented below:

$$0.5 \circledast x \oplus 0.5 \neq x \rightarrow 0.5 \circledast x \oplus 0.5 \in [-\frac{x^2}{4} + x, \frac{x^2}{4} + 1] \tag{8.43}$$

$$0.5 \circledast x \oplus 0.5 = x \rightarrow 0.5 \circledast x \oplus 0.5 \in (1 + 4\varepsilon_{rel})\sqrt{x} \tag{8.44}$$

$$r \in [-\frac{x^2}{4} + x, \frac{x^2}{4} + 1] \rightarrow x \oslash r \in \mathbb{F} \tag{8.45}$$

$$r \in [-\frac{x^2}{4} + x, \frac{x^2}{4} + 1] \rightarrow r \oplus (x \oslash r) \in \mathbb{F} \tag{8.46}$$

$$r \in [-\frac{x^2}{4} + x, \frac{x^2}{4} + 1] \rightarrow 0.5 \circledast (r \oplus (x \oslash r)) \in \mathbb{F} \tag{8.47}$$

$$r \in [-\frac{x^2}{4} + x, \frac{x^2}{4} + 1] \wedge 0.5 \circledast (r \oplus (x \oslash r)) \neq r \rightarrow 0.5 \circledast (r \oplus (x \oslash r)) \in [-\frac{x^2}{4} + x, \frac{x^2}{4} + 1] \tag{8.48}$$

$$r \in [-\frac{x^2}{4} + x, \frac{x^2}{4} + 1] \wedge 0.5 \circledast (r \oplus (x \oslash r)) = r \rightarrow 0.5 \circledast (r \oplus (x \oslash r)) \in (1 + 4\boldsymbol{\varepsilon}_{rel}) \sqrt{x} \tag{8.49}$$

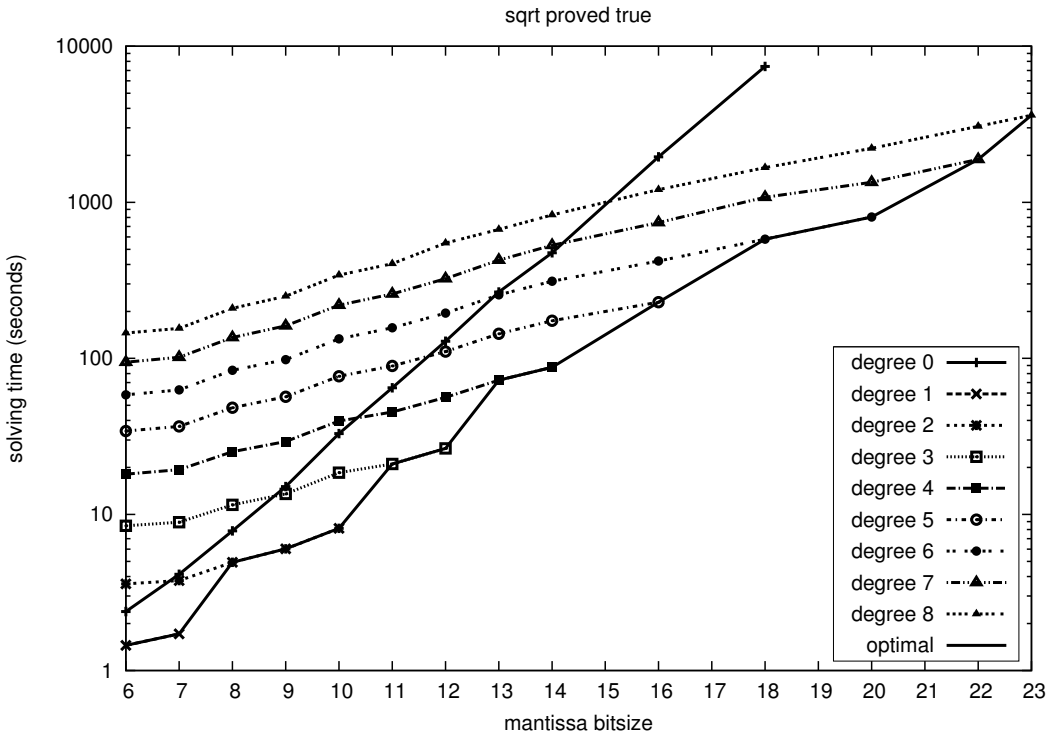
Note that boundedness checks appearing as conclusions for the exception freedom VCs in (8.45)–(8.47) also appear as hypotheses in each following VC in conjunction with the hypothesis $r \in [-\frac{x^2}{4} + x, \frac{x^2}{4} + 1]$ of (8.45). We therefore omit these checks from the hypotheses of the VCs in (8.46)–(8.49) to improve readability.

8.3.8 Revised `square_root` proving results

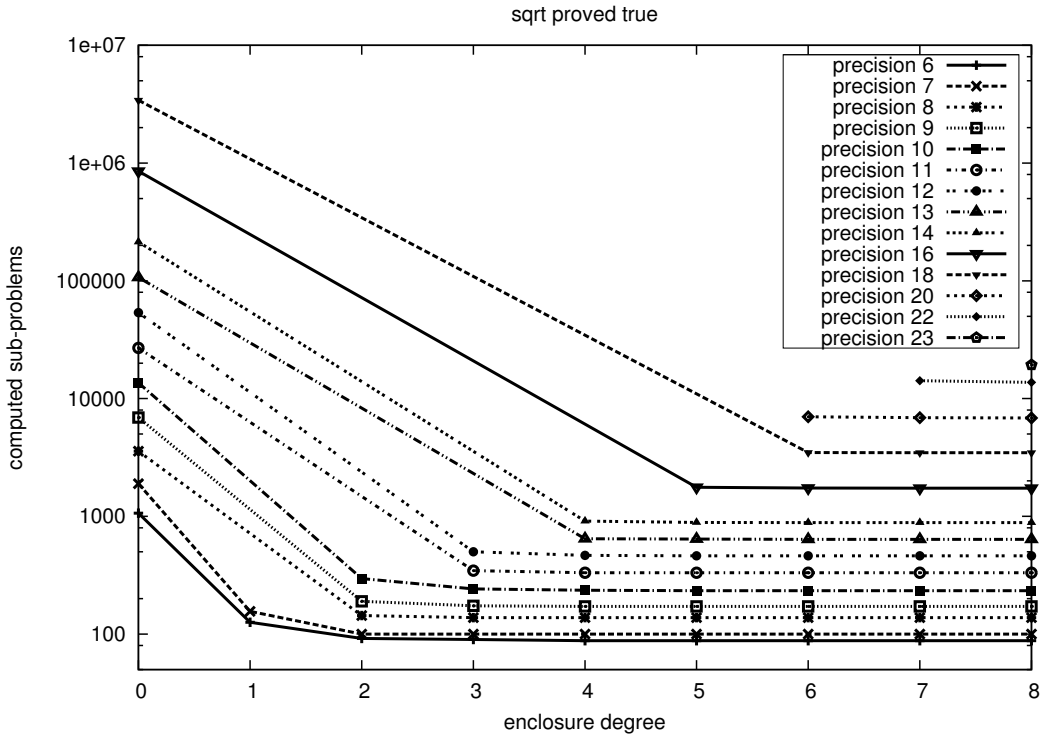
All proofs were performed for values of x in $[0.5, 2]$ and values of r in $[0, 3]$. The prover successfully discharged each VC in (8.43)–(8.48) in well under a second using degree 0 enclosures. Using higher degree enclosures decided the VC in (8.43), for the path from start into the loop, without performing splitting and slightly outperforming the constant enclosures. Affine enclosures did show consistently lower solving times for the exception freedom VCs in (8.45)–(8.47), however the difference was within measurement error. With the exception of the VC in (8.49), the remaining cases showed higher degrees leading to monotonically increasing solving times, which was expected as our revision of the program eliminated the values for which information loss may have been important.

The remaining VC, for the path from the loop invariant to the postcondition, shown in (8.49), is the one requiring the most from the approximations used in the prover. It states that on exit from the loop, *i. e.* when the loop action makes a negligible change to the value of r, the value is very near $\sqrt{x}$. This statement is formalised in the postcondition constraint and we expect its proof to show the most benefits of using higher enclosure degrees.

Initial experiments using degree 0 enclosures either timed out after several hours, or given 24 hour timeouts ran out of memory. To obtain data for analysis we simulated lower precision floating point types by increasing the floating point epsilon to reflect a smaller mantissa bit size than the 23 bits of single precision IEEE-754 floating point numbers. Proofs were attempted for precisions 6 to 23 and enclosure degrees 0 to 8 and the results are summarised in Figure 8.8 on the next page. The missing data points in the graphs



(a) Solving time, as function of mantissa bit size, for maximum enclosure degree 0 to 8.



(b) The number of computed sub-problems as function of maximum enclosure degree.

Figure 8.8: Proving the revised `square_root` functional correctness VC for floating point types with precision 6 to 23.

denote that the proof was terminated prematurely as subdivision had generated a domain box with intervals whose endpoints are double precision floating point neighbours.

Figure 8.8a shows the graphs of solving time, as a function of floating point type precision, for each degree. The final graph, labelled optimal, chooses the minimum solving time for the given precision. Clearly, it is possible to choose a positive degree for each precision for which the prover performs better than when using constant intervals. This validates our expectations that the VC in (8.49) requires very precise approximation and that higher degree enclosures give an overall benefit to the prover algorithm. The improvement in solving time is due to the ability of higher degree enclosures to decide the VC over larger sub-domains, leading to less splitting and thus less computation. Since the additional cost of computing with higher degree polynomials is lesser than the exponential cost of splitting, the overall performance was improved.

Figure 8.8b shows the number of computed sub-problems for each precision, as a function of enclosure degree. Since this number is proportional to the space complexity of the proof the graph gives a picture of the asymptotic cost of precision and the ability of higher degree enclosures to mitigate it. We see that, for each precision, using higher degrees leads to significantly smaller numbers of sub-problems, with improvements ranging between one and three orders of magnitude. As precision is increased, higher degrees are required for successful completion of the proof, with the full 23 bit precision proof only succeeding when using degree 8 enclosures.

9 Conclusions and further work

CONTENTS

9.1 Summary

The work presented in the previous chapters was aimed at investigating the automation of correctness proofs for floating point programs. Many high integrity applications, such as systems in the avionics, railway and military sectors, rely on floating point computation for the implementation of real number algorithms. Since the values computed by such implementations necessarily diverge from the sought real number values it becomes imperative to quantify these deviations in order to show that the behaviour of the resulting system conforms with the expectations of its users.

The analyses involved with such quantification typically require a great amount of effort due to the size and complexity of said systems. Thus, it is desirable to provide tool support that automates parts of such analyses. In our case specification is left as a manual task, but the remaining process is automated, as far as possible.

9.2 Contributions

In the following sections we outline the contributions made and the conclusions drawn from the results of our investigation.

9.2.1 Defining referential transparency for statement languages

Referential transparency was first defined formally by Søndergaard and Sestoft in [70] for purely applicative languages, *i. e.* for expressions without side-effects. The problem of defining referential transparency in the context of imperative languages was considered by Daniels in unpublished work [28]. He gave the notion for an expression language where functions are defined imperatively, *i. e.* using statements including assignments and procedures. In Section 3.6 the notion is generalised to procedural languages, by treating statements as generalised expressions and lifting the standard definition accordingly. Through this generalisation we make formal the intuition that SPARK Ada shares an essential property of pure languages, which explains why the particular restrictions imposed on the Ada subset facilitate formal analysis.

9.2.2 Automated generation of floating point proof obligations

The problem this thesis has addressed is that of automation of correctness proofs for imperative floating point programs in general and SPARK Ada floating point programs in particular. The approach taken was by automated generation of correctness theorems followed by automated theorem proving. Part one dealt with the task of generating proof obligations for exception freedom and functional properties, achieved by presenting, in Chapter 3, a logical semantics which translates a program-specification pair into a correctness theorem. The process was proved sound in Theorem 3.5.1, where it was shown that the generated correctness theorem implies the correctness theorem derived using a structural operational semantics, closely modelling an intuitive notion of correctness. The novelty with this work has been the simplification of the usual operational semantics made possible due to the *referential transparency* of the modelled language which eliminates the need for a top level environment that is usually introduced for bookkeeping of global variables.

9.2.3 Introduction of the integration operator

The addition of the integration operator to the expression sublanguage made the specification language more expressive. Given in Section 5.4.1, the extension allowed us to implement the error function in Section 5.4.2 with a specification using the exact integral that defines the computed function.

9.2.4 Introduction of interval expressions

The extension of the specification language with interval expressions and relations was described in Section 5.4.3. The extension allowed us to give an implementation of a widely used algorithm for the square root function with a specification that is composable, making it easier to reason about the program when treated as a "black box".

To our knowledge these extensions are novel in the context of specification languages for floating point programs and they have been documented in [36].

9.2.5 Reduction of information loss in subdivision search algorithms

A novel approach to information loss in subdivision search algorithms presented in Chapter 8. The use of polynomial interval arithmetic was intended to mitigate the so called *wrapping effects* and *dependency problem* of traditional interval arithmetic that limit the size and complexity of numerical theorems that standard techniques can tackle. We proposed in [37], for the first time, the use of *function interval arithmetic* to tackle such information loss effects in numerical theorem provers. We described the implementation of a prototype prover in Chapter 8. We showed in Section 8.3 that we managed to achieve full automation of the correctness theorems for our two example programs, proving very tight functional properties, as well as floating point exception freedom. The experiments showed that using polynomial intervals gives significantly improved performance of the bisections search algorithm implemented in the prototype prover. Most encouraging was the finding that both time and space complexity of the algorithm can be improved by several orders of magnitude, confirming the hypothesis that polynomial intervals can extend the set of floating point correctness theorems that are amenable to current automated techniques.

The main contribution in this part of the work has been the fully automated functional verification of two non-trivial floating-point programs. To our knowledge it is the first time that such a proof has been successfully conducted for a floating point program implementing a fixpoint algorithm.

9.3 Further work

9.3.1 Referential transparency for statement languages

The generalisation given in this thesis used the idea of generalised operators and expressions. this notion deserves a thoroughly formal treatment, in particular the use of the model language PROC, rather than a abstract expression language is unsatisfactory. Future work should address this problem, by defining an abstract syntax scheme which supports the definition of referential transparency and state transformer semantics, but without resorting to a concrete syntax.

A related direction is the investigation of the strength order on equivalences. There are

various data and information flow analyses that can support stronger notions of referential transparency, but it is not clear how to include them without referring to concrete constructs. If W-equivalence indeed yields the strongest notion possible to define for a fully abstract syntax, then this statement should be properly formalised and proven.

9.3.2 Generation of proof obligations

A possible direction for future research is to extend the model language and logical semantics to a more realistic language, possibly including more basic constructs, such as loops with multiple exits and a fully formal treatment of exceptional behaviour.

Another direction is the addition of arrays to the language. Anecdotal evidence suggests that while loops, with floating point expressions in the exit condition, are rare while for-loops, iterating over arrays of floating point values, seem to occur often in practical applications. The difficulty is that currently each element of an array would have to be treated as a separate variable, making our approach feasible only for very small arrays. As the arrays found in practice tend to be of the order of thousands of elements, it remains an open question how to extend the techniques described in this work to handle industrial code with arrays.

Yet another direction could investigate the potential advantages that referential transparency brings to whole program optimisation for imperative languages. Recent advances in automated parallelisation, such as the implementation of nested data parallelism in Haskell [51, 52], have been attributed to similar properties of the language.

9.3.3 Integration operator

While the current implementation only supports integration of a single variable, it can easily be modified to handle nested integrals. Another limitation of the current implementation is that the primitive function of the integrand needs to be nondecreasing over the problem domain. It is an interesting question how to extend the capabilities of the arithmetic and prover to enable arbitrary integrands while retaining the quality of the approximation.

9.3.4 Reduction of information loss

Future work should investigate how the additional information encoded in the computed functions can be harnessed to further improve such algorithms. One possibly fruitful direction is the subdivision variable selection algorithm, which in optimal cases can decrease the base of the exponential cost of subdivision.

Another direction of potential investigations is the optimal setting of the various parameters of the prover and arithmetic. The precision of the basic numeric type used, the enclosure degree and term number, the integration depth parameter and the various effort indicators of the elementary function approximations used by the arithmetic, all affect the performance of the prover. It is clear that no optimal setting can exist for a general theorem, therefore it is an important question how to set these parameters on a per-subproblem basis, making the bisection search an adaptive algorithm.

An important direction is the investigation of which of the numerous developments of interval evaluation in subdivision search algorithms, see *e.g.* [11, 12], could easily be carried over to the case of PFE evaluation based subdivision search. It seems that most implementations of interval-based subdivision search [45, 66] use some degree of symbolic pre-processing. In many cases serious over-estimations and following information loss can be eliminated by application of suitable rewriting rules. It is expected that significant improvements in the average performance of our prover could be achieved with the integration of an initial symbolic pre-processing phase.

A final direction of future research could investigate the possible benefits of employing function interval arithmetic in other decision procedures for the reals. One example is the Metitarski [4, 5, 6] system, where polynomial bounds are also derived for expressions involving elementary functions, but where a symbolic procedure called QEPCAD B [20] based on cylindrical algebraic decomposition is used to decide the inequalities. Also, the arithmetic used to derive polynomial intervals for expressions in our prover could potentially be used to improve the approximation of real predicates in SMT systems in general, where the prototype prover could be used as a T-solver for inequalities involving elementary transcendental functions and integrals. The integration of interval evaluation with a SAT solver has been described in [40]. The benefits that function intervals bring to automated reasoning systems should carry over to interactive provers as well. Recent work [29] in PVS [61, 62] on proving real number theorems involving elementary

functions also uses polynomial bounds to improve the approximation of ranges for expressions. The arithmetic used in the prototype prover could further improve such range estimations.

Bibliography

[1] *Ada Reference Manual, ISO/IEC 8652:2007(E) Ed. 3*. Ada Europe, 2007.

[2] Milton Abramowitz and Irene A. Stegun. *Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables*. Dover, New York, ninth dover printing, tenth gpo printing edition, 1964.

[3] Rev R. Acad, Cien Serie, A. Mat, Matt Kaufmann, and J Strother Moore. Ciencias de la computacin / computational sciences some key research problems in automated theorem proving for hardware and software verification, 2004.

[4] Behzad Akbarpour and Lawrence C. Paulson. Metitarski: An automatic prover for the elementary functions. In *AISC/MKM/Calculemus*, pages 217–231, 2008.

[5] Behzad Akbarpour and Lawrence C. Paulson. Applications of metitarski in the verification of control and hybrid systems. In *HSCC*, pages 1–15, 2009.

[6] Behzad Akbarpour and Lawrence C. Paulson. Metitarski: An automatic theorem prover for real-valued special functions. *J. Autom. Reasoning*, 44(3):175–205, 2010.

[7] John Anton, Eric Bush, Allen Goldberg, Klaus Havelund, Doug Smith, and Arnaud Venet. Towards the industrial scale development of custom static analyzers. In Elizabeth Fong, editor, *Proceedings of the Static Analysis Summit*, number 500-262, pages 36–40. National Institute of Standards and Technology, 2006.

[8] Thomas Ball, Byron Cook, Vladimir Levin, and Sriram K. Rajamani. Slam and static driver verifier: Technology transfer of formal methods inside microsoft. In *In: IFM. (2004*, pages 1–20. Springer, 2004.

[9] John Barnes. The spark way to correctness is via abstraction. *Ada Lett.*, XX(4):69–79, 2000.

[10] John Barnes. *High Integrity Software: The SPARK Approach to Safety and Security*. Addison-Wesley, 2 edition, April 2003.

[11] Heikel Batnini, Claude Michel, Michel Rueher, and Université De Nice Sophia-antipolis. Mind the gaps: A new splitting strategy for consistency techniques. In *In Proceedings of Principles and Practice of Constraint Programming (CP 2005*, pages 77–91. Springer, 2005.

[12] Frédéric Benhamou, Frédéric Goualard, Laurent Granvilliers, and Jean-François Puget. Revising hull and box consistency. In *Proceedings of the 1999 international conference on Logic programming*, pages 230–244, Cambridge, MA, USA, 1999. Massachusetts Institute of Technology.

[13] Christian Bessiere, editor. *Principles and Practice of Constraint Programming - CP 2007, 13th International Conference, CP 2007, Providence, RI, USA, September 23-27, 2007, Proceedings*, volume 4741 of *Lecture Notes in Computer Science*. Springer, 2007.

[14] Dirk Beyer, Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. The software model checker blast: Applications to software engineering. *INT. J. SOFTW. TOOLS TECHNOL. TRANSFER*, 2007.

[15] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. A static analyzer for large safety-critical software. In *PLDI*. ACM, 2003.

[16] Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. Design and implementation of a special-purpose static program analyzer for safety-critical real-time embedded software. In *The essence of computation: complexity, analysis, transformation*, pages 85–108. Springer-Verlag New York, Inc., New York, NY, USA, 2002.

[17] Sylvie Boldo. Floats & Ropes: a case study for formal numerical program verification. In *36th International Colloquium on Automata, Languages and Programming*, volume 5556 of *Lecture Notes in Computer Science - ARCoSS*, pages 91–102, Rhodos, Greece, July 2009. Springer.

[18] Sylvie Boldo and Jean-Christophe Filliâtre. Formal verification of floating-point programs. In *Proceedings of 18th IEEE International Symposium on Computer Arithmetic*, Montpellier, France, June 2007.

[19] Sylvie Boldo, Jean-Christophe Filliâtre, and Guillaume Melquiond. Combining coq and gappa for certifying floating-point programs. In *Calculemus '09/MKM '09: Proceedings of the 16th Symposium, 8th International Conference. Held as Part of CICM '09 on Intelligent Computer Mathematics*, pages 59–74, Berlin, Heidelberg, 2009. Springer-Verlag.

[20] Christopher W. Brown. Qepcad b: a program for computing with semi-algebraic sets using cads. *SIGSAM Bull.*, 37:97–108, December 2003.

[21] Bernard Carré and Jonathan Garnsworthy. Spark – an annotated ada subset for safety-critical programming. In *Proceedings of the conference on TRI-ADA '90*, TRI-Ada '90, pages 392–402, New York, NY, USA, 1990. ACM.

[22] Byron Cook, Daniel Kroening, and Natasha Sharygina. Cogent: Accurate theorem proving for program verification. In *Proceedings of CAV 2005, volume 3576 of Lecture Notes in Computer Science*, pages 296–300. Springer, 2005.

[23] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Conference Record of the Fifth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 84–97, Tucson, Arizona, 1978. ACM Press, New York, NY.

[24] Patrick Cousot. The verification grand challenge and abstract interpretation. In *Verified Software: Theories, Tools, Experiments: First IFIP TC 2/WG 2.3 Conference, VSTTE 2005, Zurich, Switzerland, October 10-13, 2005, Revised Selected Papers and Discussions*, pages 189–201, Berlin, Heidelberg, 2008. Springer-Verlag.

[25] Patrick Cousot and Radhia Cousot. Modular static program analysis. In *Proceedings of Compiler Construction*, pages 159–178. Springer-Verlag, 2002.

[26] Patrick Cousot and Radhia Cousot. On abstraction in software verification. In *CAV '02: Proceedings of the 14th International Conference on Computer Aided Verification*, pages 37–56, London, UK, 2002. Springer-Verlag.

[27] Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. The ASTREÉ analyzer. In Shmuel Sagiv, editor, *ESOP*, volume 3444 of *Lecture Notes in Computer Science*, pages 21–30. Springer, 2005.

[28] Anthony C. Daniels and Meanings Of Programs. Referential transparency in imperative languages.

[29] Marc Daumas, David Lester, and Cear Munoz. Verified real number calculations: A library for interval arithmetic. *IEEE Transactions on Computers*, 58:226–237, 2009.

[30] Marc Daumas and Guillaume Melquiond. Generating formally certified bounds on values and round-off errors. In *Proceedings of 6th Conference on Real Numbers and Computers*, pages 55–70, Schloss Dagstuhl, Germany, 2004.

[31] Marc Daumas and Guillaume Melquiond. Certification of bounds on expressions involving rounded operators. *ACM Trans. Math. Softw.*, 37(1):1–20, 2010.

[32] Alain Deutsch. Static verification of dynamic properties. Technical report, PolySpace Technologies, 2003. Accessed on 18th April 2010.

[33] Ellie D'hondt and Prakash Panangaden. Quantum weakest preconditions. *Mathematical Structures in Computer Science*, 16:429–451, 2006.

[34] Edsger W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM*, 18(8):453–457, 1975.

[35] Edsger W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.

[36] Jan Andrzej Duracz, Amin Farjudian, and Michal Konečný. Enclosure constraints for floating point software verification. In *Proceedings of CFV 2009 in Grenoble*, 2009.

[37] Jan Andrzej Duracz and Michal Konečný. Polynomial function enclosures and floating point software verication. In *Proceedings of CFV 2008 in Sydney*, pages 56–67, 2008.

[38] Pär Emanuelsson and Ulf Nilsson. A comparative study of industrial static analysis tools. *Electronic Notes in Theoretical Computer Science*, 217:5 – 21, 2008.

Proceedings of the 3rd International Workshop on Systems Software Veri[fi]cation (SSV 2008).

[39] Robert W. Floyd. Assigning meanings to programs. In J. T. Schwartz, editor, *Mathematical Aspects of Computer Science*, volume 19 of *Proceedings of Symposia in Applied Mathematics*, pages 19–32, Providence, Rhode Island, 1967. American Mathematical Society.

[40] Martin Fränzle, Christian Herde, Tino Teige, Stefan Ratschan, and Tobias Schubert. Efficient solving of large non-linear arithmetic constraint systems with complex boolean structure. *Journal on Satisfiability, Boolean Modeling and Computation*, 1:209–236, 2007.

[41] H. H. Goldstine and J. von Neumann. Planning and coding of problems for an electronic computing instrument. In *John von Neumann: Collected Works*, volume 5. Pergamon Press, 1963.

[42] Saul Gorn. Specification languages for mechanical languages and their processors a baker's dozen: a set of examples presented to asa x3.4 subcommittee. *Commun. ACM*, 4(12):532–542, 1961.

[43] E. Goubault, M. Martel, and S. Putot. Asserting the precision of floating-point computations: a simple abstract interpreter. In *European Symposium on Programming, LNCS 2305*, pages 209–212. Springer Verlag, 2002.

[44] Eric Goubault. Static analyses of floating-point operations. In *In SAS01, volume 2126 of LNCS*, pages 259–339. Springer, 2001.

[45] L. Granvilliers and F. Benhamou. Algorithm 852: RealPaver: An interval solver using constraint satisfaction techniques. *ACM Transactions on Mathematical Software*, 32(1), 2006.

[46] D. Guaspari, C. Marceau, and W. Polak. Formal verification of ada programs. *IEEE Trans. Softw. Eng.*, 16(9):1058–1075, 1990.

[47] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–585, October 1969.

[48] Michael Huth. Some current topics in model checking. *Int. J. Softw. Tools Technol. Transf.*, 9(1):25–36, 2007.

[49] Andrew Ireland, Bill J. Ellis, Andrew Cook, Roderick Chapman, and Janet Barnes. An integrated approach to high integrity software verification. *Journal of Automated Reasoning*, 36(4):379–410, 2006.

[50] C. B. Jones. The early search for tractable ways of reasoning about programs. *IEEE Annals of the History of Computing*, 25, 2003.

[51] Simon L. Peyton Jones, Roman Leshchinskiy, Gabriele Keller, and Manuel M. T. Chakravarty. Harnessing the multicores: Nested data parallelism in haskell. In Ramesh Hariharan, Madhavan Mukund, and V. Vinay, editors, *FSTTCS*, volume 2 of *LIPIcs*, pages 383–414. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2008.

[52] Gabriele Keller, Manuel M. T. Chakravarty, Roman Leshchinskiy, Simon L. Peyton Jones, and Ben Lippmeier. Regular, shape-polymorphic, parallel arrays in haskell. In Paul Hudak and Stephanie Weirich, editors, *ICFP*, pages 261–272. ACM, 2010.

[53] Leslie Lamport. win and sin: Predicate transformers for concurrency. *ACM Transactions on Programming Languages and Systems*, 12(3):396–428, 1990.

[54] J. Mccarthy. Towards a mathematical science of computation. In *Proc. Information Processing '62*, pages 21–28. North-Holland, 1963.

[55] A. Miné. Relational abstract domains for the detection of floating-point run-time errors. In *ESOP'04*, volume 2986 of *LNCS*, pages 3–17. Springer, 2004.

[56] David Monniaux. The pitfalls of verifying floating-point computations. *ACM Trans. Program. Lang. Syst.*, 30(3):1–41, 2008.

[57] R. E. Moore. *Interval Analysis*. Prentice-Hall, Englewood Cliffs N. J., 1966.

[58] Carroll Morgan, Annabelle McIver, and Karen Seidel. Probabilistic predicate transformers. *ACM Transactions on Programming Languages and Systems*, 18:325–353, 1995.

[59] P. Naur. Proof of algorithms by general snapshots. *BIT*, 6(4):310–316, 1966.

[60] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1999.

[61] S. Owre, J. M. Rushby, , and N. Shankar. PVS: A prototype verification system. In Deepak Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752, Saratoga, NY, jun 1992. Springer-Verlag.

[62] Sam Owre, John Rushby, N. Shankar, and David Stringer-Calvert. PVS: an experience report. In Dieter Hutter, Werner Stephan, Paolo Traverso, and Markus Ullman, editors, *Applied Formal Methods—FM-Trends 98*, volume 1641 of *Lecture Notes in Computer Science*, pages 338–345, Boppard, Germany, oct 1998. Springer-Verlag.

[63] G. D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, University of Aarhus, 1981.

[64] Sylvie Putot, Eric Goubault, and Matthieu Martel. Static analysis-based validation of floating-point computations. *LNCS*, 2991:306–313, 2004.

[65] B. Randell. *The Origins of Digital Computers: Selected Papers (Monographs in Computer Science)*. Springer-Verlag, 2nd edition, 1975.

[66] Stefan Ratschan et al. RSolver. `http://rsolver.sourceforge.net`, 2004. Software Package.

[67] Sriram Sankaranarayanan, Michael Colon, Henny Sipma, and Zohar Manna. Efficient strongly relational polyhedral analysis. In E. Allen Emerson and Kedar S. Namjoshi, editors, *Verification, Model Checking, and Abstract Interpretation: 7th International Conference, (VMCAI)*, volume 3855 of *lncs*, pages 111–125, Charleston, SC, January 2006. Springer Verlag.

[68] IEEE Computer Society. *IEEE Std 754-1985*. Institute of Electrical and Electronics Engineers, New York, 1985.

[69] IEEE Computer Society. *ANSI/IEEE Std 854-1987*. Institute of Electrical and Electronics Engineers, New York, 1987.

[70] Harald Sondergaard and Peter Sestoft. Referential transparency, definiteness and unfoldability. *Acta Inf.*, 27:505–517, January 1990.

[71] Alan M. Turing. Checking a large routine. In *Report of a Conference on High Speed Automatic Calculating Machines*, pages 67–69, 1949.

[72] A. van Wijngaarden. Numerical analysis as an independent science. *BIT*, 6(1):66–81, 1966.

[73] M. Warmus. Calculus of approximations. *Bull. Acad. Polon. Sci. CL. III*, IV(5):253–259, 1956.

[74] M. Warmus. Approximations and inequalities in the calculus of approximations. classification of approximate numbers. *Bull. Acad. Polon. Sci., Ser. math. astr. et phys.*, IX(4):241–245, 1961.