Automatic Numerical Solving for Auto-active Verification of Floating-Point Programs

Junaid Ali Rasheed

Doctor of Philosophy

Aston University

September 2022

Abstract

We present a new process for the verification of numerical programs with tight functional specifications that feature exact arithmetic including selected transcendental functions. The process, which simplifies, derives bounds, and safely eliminates floating-point operations from Verification Conditions (VCs) produced by Why3, is capable of automatically verifying such specifications and is implemented in our new open source tool named PropaFP. We evaluate PropaFP alongside the state-of-the-art in formal verification of floating-point programs where we find that the process is able to verify specifications that current tools are unable to verify.

We also present novel branch-and-prune contractions based on linearisations of conjunctions that consist of nonlinear real inequalities with differentiable expressions. These linearisations and contractions are implemented in our new open source numerical prover named LPPaver. The contractions we have discovered are used to significantly improve the 'pruning' step of our branch-and-prune algorithm. We evaluate LPPaver alongside state-of-the-art automated solvers for problems involving nonlinear real arithmetic. LPPaver performs comparably and, in some cases, better than these solvers.

Together, PropaFP and LPPaver yield the first fully automatically verified implementations of the sine and square root functions with tight functional specifications.

Acknowledgements

I would like to thank my supervisor, Michal Konečný, for many valuable discussions throughout my PhD, for valuable feedback on the methods I've discovered, for valuable feedback on early implementations of PropaFP and LPPaver, and for feedback on early drafts of this thesis.

I would like to thank Yannick Moy from AdaCore for providing valuable examples of SPARK programs that cannot be verified using state-of-the-art verification tools and for giving valuable feedback on a paper describing PropaFP. I would also like to thank my examiners, Claude Marché and Hai Wang, for their helpful suggestions on the thesis.

Finally, I would like to thank Michal Konečný and Eike Neumann for their work on AERN2 [41] which is an interval arithmetic library that both PropaFP and LPPaver make heavy use of.

Collaborator Acknowledgements

Some content, namely Chapter 4 and Sections 2.8, 5.1, and 6.2, has been reused from a paper describing PropaFP [56] that was co-written with my supervisor.

The 'bounds derivation' algorithm described in Section 4.2 was implemented mostly by my supervisor and is used in both PropaFP and LPPaver.

Funding

This PhD project has received funding from AdaCore Ltd and from the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement No 731143.

Contents

Abstract ii

Acknowledgements iii

1 Introduction 1
1.1 Context 2
1.1.1 Verification of FP Programs 2
1.1.2 Automated Solving 3
1.1.3 Overview of Contributions 4

2 Background 8
2.1 Preliminaries 8
2.1.1 Floating-point Arithmetic 8
2.1.2 Matrices 11
2.1.3 Intervals 16
2.1.4 Interval Arithmetic 17
2.1.5 Automatic Differentiation 19
2.1.6 S-expressions 19
2.2 Interval Methods 19
2.2.1 Branch-and-prune 19
2.2.2 Interval Constraint Checking 21
2.2.3 Newton's Method 21
2.3 Constraint Satisfaction Problems 24
2.3.1 SAT 24

2.3.2 SMT 27
2.3.3 Linear Programming 28
2.3.4 Interval Constraint Propagation 28
2.4 Optimisation Problem 32
2.4.1 Linear Programming 32
2.5 Solving Systems of Linear Equations 33
2.5.1 General Forms 34
2.5.2 Vector Equation 34
2.5.3 Matrix Equation 35
2.5.4 Gauss-Seidel Method 35
2.6 Haskell Basics 37
2.7 Solving Numerical CSPs 37
2.7.1 MetiTarski 38
2.7.2 dReal 38
2.7.3 ksmt 41
2.7.4 Colibri 41
2.8 Formal Verification of FP Programs 43
2.8.1 Why3 43
2.8.2 Writing and Specifying FP Programs in SPARK 44
2.8.3 Verifying a Sine Approximation in SPARK 44
2.8.4 Alternatives to SPARK 46

3 LPPaver 48

3.1 Input 48
3.2 Symbolic Simplifications 50
3.3 Variable Domains and Boxes 50
3.4 eDNF Local Simplifications and Bound Derivations 51
3.5 Contracting a Box Using Linearisations 52
3.5.1 System with Two Variables 52
3.5.2 System with an Arbitrary Number of Variables 54
3.5.3 Calling the Simplex Method 55
3.5.4 Soundness 56
3.6 Pruning via Interval Methods and Linearisations 59

3.6.1 Termination . . . 60
3.6.2 Soundness . . . 60
3.7 Showing Unsatisfiability via Depth-First Splitting & Pruning . 62
3.7.1 Termination . . . 64
3.7.2 Soundness . . . 65
3.8 Searching for a Model using Linearisations . . . 67
3.8.1 System with Two Variables . . . 68
3.8.2 System with an Arbitrary Number of Variables . . . 69
3.8.3 Calling the Simplex Method . . . 70
3.8.4 Soundness . . . 71
3.9 Pruning and Searching for Models via Interval Methods and Linearisations . . . 73
3.9.1 Termination . . . 73
3.9.2 Soundness . . . 75
3.10 Finding Models via Best-First Searching and Pruning . . . 76
3.10.1 Termination . . . 77
3.10.2 Soundness . . . 79

4 PropaFP 81
4.1 Our Proving Process Steps . . . 83
4.1.1 Generating and processing verification conditions . . 83
4.1.2 Simplifications . . . 86
4.2 Bounds Derivations . . . 93
4.2.1 Eliminating floating-point operations . . . 106
4.3 Deriving Provable Error Bounds . . . 112
4.4 Verifying Heron's Method for Approximating the Square Root Function . . . 117
4.5 Verifying AdaCore's Sine Implementation . . . 118
4.5.1 Multiply_Add . . . 119
4.5.2 My_Machine_Rounding . . . 119
4.5.3 Reduce_Half_Pi . . . 120
4.5.4 Approx_Sin and Approx_Cos . . . 122

4.5.5 Sin 122
4.5.6 Generated Why3 NVCs 125

5 Evaluation **130**
5.1 Evaluation of PropaFP 130
5.1.1 Benchmarking the PropaFP Proving Process 130
5.2 Evaluation of LPPaver 136
5.2.1 Performance of Provers on PropaFP Examples 137
5.2.2 Placing Spheres of an Equal Size in a Cube 139

6 Conclusion **148**
6.1 LPPaver 148
6.1.1 Future Work 149
6.2 PropaFP 151

Bibliography **162**

List of Tables

4.1 Error bounds computed by FPTaylor 109
4.2 Error bound components for `Taylor_Sin` 114
4.3 Why3 NVCs Generated for each Procedure from our Modified AdaCore Sine Implementation 125

5.1 Proving Process on Described Examples 131
5.2 Effect of Specification Bound on Proving Time 133
5.3 Proving Process on Described Counter-examples 135
5.4 Generated Instances of (5.1) 141
5.5 Results and timings of solvers on (5.1) instances 143
5.6 Models for Place23 145
5.7 Checking models given by provers for (5.1) instances 146

List of Figures

2.1 Overview of Automated Verification via GNATprove (adapted from [28]) 43
2.2 Overview of Automated Verification via Frama-C/SPARK/Krakatoa (adapted from [44]) 47

3.1 Linearisations that weaken a term whose function graph is f over the 1-dimensional box $\mathbf{b}$. The lines labelled $w_L(f)$ and $w_R(f)$ are the linearisations made from the left and right 'extreme' corners of $\mathbf{b}$, respectively. This linearisation allows one to safely contract $\mathbf{b}$ by a small amount from the left, giving us the new box $\mathbf{b}'$. 56
3.2 A linearisation that strengthens a term whose function graph is f over the 1-dimensional box $\mathbf{b}$. The lines labelled $s_L(f)$ and $s_R(f)$ are the linearisations made from the left and right 'extreme' corners of $\mathbf{b}$, respectively. Only the $s_R(f)$ linearisation would succeed in finding a model over the box and the set of models that can be found with this linearisation is represented by the dotted line. 70

4.1 Overview of Automated Verification via GNATprove with PropaFP 84

5.1 An arrangement of 3 equally sized circles in a square that satisfies (5.1). 140

5.2 Packing of 4 equally sized circles in a square. This does not satisfy (5.1) as the circles are touching; the distances between the centres of the touching circles is exactly $2 \cdot radius$. 142
5.3 Approximation of the δ-model shown in (5.2) given by dReal. This does not satisfy (5.1) as some of the circles are overlapping. 144

6.1 On the left, we have a contraction of a box using a system of inequalities. On the right, we rotate the box while contracting, reducing to a much smaller box. 150

List of Algorithms

1 Generic branch-and-prune algorithm [35] 20
2 DPLL Algorithm 26
3 Basic DPLL(T) Algorithm. A real world implementation would also return a model for satisfiable results. 29
4 Prune: contract a box using interval methods and linearisations 59
5 Proving with depth-first branching + pruning 63
6 PruneAndSearch: contract a box and search for a model using interval methods and linearisations 74
7 priority: calculate a priority number for some box and conjunction of `EConstraints`, a higher priority value should be prioritised over lower values. 77
8 Searching for a model with best-first branching + pruning . . 78
9 SimplifyE: simplify an expression with symbolic rules 87
10 SimplifyF: simplify a VC with symbolic rules 90
11 Simplify: simplify a VC with symbolic rules 91
12 EContainsVars : Check if an expression contains at least one variable from a list of variables 94
13 FilterVarsF : Attempts to filter out given variables from the given formula by weakening the formula 96
14 ScanAndDerive: Scan through a formula, deriving bounds where possible 99
15 DeriveBounds: Iteratively derive bounds for variables 102

16 DeriveBoundsAndFilter: Derive bounds for variables and attempt to filter out variables with at least one unbounded endpoint. 104
17 EliminateFloats: eliminate floating-point operations within a formula over some box 107
18 PropaFP: simplify, derive bounds for variables, and eliminate floats within a VC 111

Listings

2.1 Sine approximation in Ada 45
2.2 SPARK formal specification of `Taylor_Sin` 45
4.1 Approximation of the sine function in Ada 81
4.2 SPARK formal specification of `Taylor_Sin` 83
4.3 NVC corresponding to the post-condition from Listing 4.2 . . 85
4.4 `Taylor_Sin` NVC after simplification and bounds derivation . . 106
4.5 FPTaylor file to compute an error bound of the `Taylor_Sin` VC 109
4.6 `Taylor_Sin` NVC after removal of FP operations 110
4.7 `Taylor_Sin` simplified exact NVC, ready for provers 110
4.8 SinSin function definition in SPARK 115
4.9 SinSin function specification in SPARK 115
4.10 Heron's Method Specification 117
4.11 Heron's Method Implementation 117
4.12 Multiply_Add Implementation 119
4.13 Multiply_Add Specification 119
4.14 My_Machine_Rounding Implementation 120
4.15 My_Machine_Rounding Specification 120
4.16 Reduce_Half_Pi Implementation 121
4.17 Reduce_Half_Pi Specification 121
4.18 Approx_Sin and Approx_Cos Implementation 123
4.19 Approx_Sin and Approx_Cos Specification 123
4.20 Sin Implementation 124
4.21 Sin Specification 124
4.22 Selected Reduce_Half_Pi Simplified Exact NVCs 127

4.23 Selected Approx_Sin NVC 128
4.24 Selected Sin NVC 129

Chapter 1

Introduction

In this thesis, we describe the theory underling two new tools, PropaFP and LPPaver. PropaFP implements a novel process for automatic verification of floating-point (FP) programs. LPPaver is an automated numerical prover that implements novel contractors based on linearisations of nonlinear real functions and is useful for automatic verification of FP programs when used alongside PropaFP. In some cases, LPPaver is able to outperform other state-of-the-art numerical provers. The outline of the thesis is given below.

- Chapter 1 - Introduction - Give some context for what the problem is and outline contributions.
- Chapter 2 - Background - Describe preliminaries as well as state-of-the-art FP software verification tools and automated provers for nonlinear real arithmetic.
- Chapter 3 - LPPaver - Describe the algorithms in our new tool, LPPaver, including the novel contractors that it implements.
- Chapter 4 - PropaFP - Describe our novel proving process for verification of FP programs. We also present several new benchmarks for evaluating formal verification tools for FP programs.
- Chapter 5 - Evaluation - Evaluate PropaFP and LPPaver alongside state-of-the-art formal verification tools and solvers.

- Chapter 6 - Conclusion - An overview of what we have contributed and potential avenues for future work.

1.1 Context

1.1.1 Verification of FP Programs

When writing programs that require some sort of numerical computations, FP numbers are commonly used. Most CPUs include a dedicated FP unit, improving the speed of FP computations which makes the choice of using FP numbers more attractive. However, an issue with FP arithmetic is rounding errors: if some number cannot be represented in FP form, the number is rounded to the nearest FP number. This causes unintuitive behaviour, for example, in FP arithmetic, $0.1+0.1+0.1+0.1+0.1+0.1+0.1+0.1+0.1+0.1 \neq 0.1 * 10 = 1$.

Rounding errors may propagate in further FP operations and this can lead to catastrophic results, particularly with safety critical applications. For example, on 25 February 1991, during the Gulf War, propagation of rounding errors lead to a missile defence system incorrectly approximating the trajectory of a missile. The defence system failed to stop the missile, contributing to the death of 28 people [50]. Alternative numerical representations, however, tend to be slower than FP operations (e.g. rational and interval arithmetic). There is a need to 'prove' that a safety critical program written with FP arithmetic will behave as expected.

Formal verification is a technique used to *prove* or *disprove* that a program is correct with respect to some specification. This is done by deriving a mathematical model of the program and then using 'automated solvers' to attempt to prove/disprove said model. For example, consider a FP approximation of the sine function named $\sin_{fp}$. A *functional specification* can be given, specifying that $\sin_{fp}$ is a sufficiently close approximation of the exact sine function.

$$|\sin_{fp}(x) - \sin(x)| \leq 0.0001 \tag{1.1}$$

If $sin_{fp}(x)$ is a single precision Taylor series approximation of the sine function, current automated provers are not able to automatically verify the specification shown in (1.1). This is due to the use of FP operations: provers that support FP operations are typically not powerful enough to prove these sorts of specifications and provers that would be able to prove a specification like this tend to not support FP operations.

Proposal - New Proving Process for Specifications of FP Programs

We propose a new process for proving Verification Conditions (VCs) derived from specifications of FP programs. The core idea behind the process is to safely eliminate FP operations using overapproximations of rounding errors. The processed VC, that now contain only exact operations, can then be passed to more powerful automated solvers. The process we propose is implemented in a new tool that we have named PropaFP. PropaFP is available under the open source MPL licence. Both the process and the tool are described in Chapter 4. The process itself is evaluated alongside the state-of-the-art tool for automatic formal verification of FP programs in Section 5.1.

1.1.2 Automated Solving

As mentioned above, automated solving can be used to prove or disprove VCs. Automated approaches to numerical solving are more popular than manual approaches due to the ease of use of automated solvers; one just needs to understand how to call an automated solver on a VC whereas for manual solvers, one would need to have the knowledge to write formal proofs for complex mathematical propositions.

However, when evaluating PropaFP, we discovered that current automated solvers are either unable to, or take a significant amount of time to, decide some of the more 'difficult' VCs (e.g., VCs that are true but become false if some numbers within them change a little bit). This may be due to the difficult VCs containing conjunctions consisting of nonlinear real inequalities including uses of transcendental functions.

Proposal - Numerical Solving with Linearisations for Conjunctions of Inequalities

To deal with this, we have developed a numerical prover that uses novel ways to utilise linearisations of conjunctions of nonlinear inequalities. There are linearisations available for both attempting to prove that the conjunction holds or attempting to find a value for variables where the conjunction is violated. The prover we have developed is named LPPaver and, with these linearisations, LPPaver is able to verify the 'difficult' VCs mentioned earlier faster than the other provers in our tests. LPPaver and the linearisations are described in Chapter 3. LPPaver is evaluated alongside state-of-the-art solvers for problems involving nonlinear real arithmetic in Section 5.2.

1.1.3 Overview of Contributions

A new proving process for formal verification of FP programs

We have introduced a proving process for FP arithmetic. The proving process is able to simplify VCs, derive bounds for variables, and safely eliminate FP operations using overapproximations of rounding errors from a given VC. The process is described in Chapter 4. The process has been evaluated and found to improve upon the state-of-the-art in formal verification of FP programs. This evaluation is presented in Section 5.1.

A set of benchmarks for evaluating techniques for verification of FP programs

When attempting to evaluate PropaFP, we discovered that there exists no standard set of benchmarks that consist of VCs arising from functional specifications of real-world FP programs, so we designed our own set of benchmarks. This set consists of:

- A functional specification of a single and double precision FP implementation of a Taylor series approximation of the sine function.
- A functional specification where the single-precision approximation of the sine function is called twice.
- A functional specification of Heron's method for approximating the square-root function which includes loop invariants.
- A functionally specified FP implementation of the sine function written by AdaCore for a high-integrity mathematics library. This implementation was modified to make it conducive for formal verification by rewriting functions that make use of features unsupported by verification tools and by limiting the input domain to avoid loops.
- A set of incorrectly specified functions to evaluate the effectiveness of a process at producing counter-examples that would be useful for users.

The FP programs that produce these benchmarks are described in detail in Chapter 4.

Two-Phase Exact Simplex Method Library in Haskell

The tools we describe in this thesis are implemented in Haskell. One of the tools implements an algorithm that relies on the two-phase simplex method. We could not find an existing Haskell library that implemented the two-phase simplex method that fit our criteria: the library must be implemented using exact arithmetic, the library must be open source, and the library must be well tested or trusted.

Thus, we have written a new implementation of the two-phase simplex method in Haskell [52] in exact rational arithmetic. The implementation well documented, well tested, and is available under the open source and permissive BSD 3-Clause licence. The implementation is integrated with popular Haskell development tools, giving the Haskell community easy access to an exact implementation of the two-phase simplex method.

New Uses of Linearisations in Branch-and-prune Methods for Proving and Disproving Systems of Nonlinear Real Inequalities

We describe linearisations for deciding conjunctions of nonlinear real inequalities over some box. There are two linearisations.

A novel contractor has been implemented with a novel use of an optimisation algorithm. The box and a linearisation that *weakens* the conjunction is used to build a system of linear real inequalities. This system is used as a contractor to remove areas from the box where the conjunction is certain to be false by optimising each variable in the system using our implementation of the simplex method. If this contraction results in the new box being empty, then the linearisation of the conjunction is false over the whole box and since this linearisation is a *weakening* of the original conjunction, the original conjunction must also be false.

The other linearisation *strengthens* the conjunction, which is used to find a value for variables where the linearisation of the conjunction is certain to be true. If the resulting system is feasible (which is determined using our implementation of the simplex method), we have values for variables where the linearisation is true, i.e., a model. Since the linearisation is a *strengthening* of the conjunction, the model for the linearisation of the conjunction is also a model for the original conjunction.

This is all implemented in a new numerical prover named LPPaver [53]. LPPaver is written in Haskell and is available under the open source MPL licence. LPPaver, the novel contractor, and these linearisations are discussed in detail in Chapter 3 and evaluated in Section 5.2.

Formally Verified FP Implementations of the Sine and Square Root Functions

With the use of LPPaver and PropaFP, we fully verified our set of benchmarks, thus yielding the first *automatically* verified FP SPARK[1] implementations of the sine and square root functions with tight functional specifications, albeit

[1] A subset of the Ada programming language designed for Formal Verification.

over a reduced domain.

Papers

PropaFP Content from a published paper regarding PropaFP [55] (along with the extended preprint [57]) was used to write most of Chapter 4 as well as part of Chapters 1, 2, 5, 6.

LPPaver A paper regarding LPPaver based mainly on content from Chapter 3 as well as some content from Chapters 5 and 6 is planned.

Chapter 2

Background

We set notations and describe existing concepts which the rest of this thesis builds on, as well as discuss relevant work. In Section 2.1, we describe some preliminaries, particularly floating-point and interval arithmetic. In Section 2.2, we present some interval variations of a selection of numerical algorithms. In Section 2.3, we introduce constraint satisfaction problems and a selection of methods to solve them. In Section 2.4, we introduce Optimisation Problems and a method to solve them. In Section 2.5, we describe systems of linear equalities and a selection of methods to solve them. In Section 2.6, we introduce some Haskell syntax that is used in later chapters. In Section 2.7, we introduce *numerical* constraint satisfaction problems and a selection of methods to solve them. Finally, in Section 2.8 we discuss available techniques used for verifying floating-point programs.

2.1 Preliminaries

2.1.1 Floating-point Arithmetic

A floating-point (FP) number is a number represented with some fixed number of significant digits, called a *significand*, multiplied with some fixed *base* that has been scaled with some *exponent*. So, FP numbers have the form:

$$base^{exponent} \times significand \tag{2.1}$$

For example, the real number 1.2 can be represented in a base 10 FP form as 12×10^{-1}. The set of all floating-point numbers is denoted $\mathbb{F}$. For programs that need to perform some non-integer operations, base 2 FP numbers are commonly used as FP arithmetic is supported by hardware and thus much faster than exact (rational) arithmetic and other types of arithmetic used to approximate real arithmetic in computer programs such as high-accuracy interval arithmetic.

The IEEE-754 Standard

The IEEE-754 Standard [40] is the widely established standard for FP arithmetic. The standard defines multiple formats for representing FP numbers with a *base* of 2 and differing *precisions*, i.e. the number of bits used to represent a FP number. A higher precision results in more accurate FP operations but at a (slight) cost to memory and speed and vice versa. The two most commonly used formats defined in this standard are:

- Single precision - 32 bits are used to represent a FP number in a binary format; 23 bits for the *significand*, 8 bits for the *exponent*, and 1 bit for the *sign*, i.e. whether the number is positive or negative.
- Double precision - 64 bits are used to represent a FP number in a binary format; 52 bits for the *significand*, 11 bits for the *exponent*, and 1 bit for the *sign*.

So, an IEEE-754 FP number is represented as:

$$sign \times 2^{exponent} \times significand \tag{2.2}$$

For a single-precision IEEE-754 FP number, bits 1-23 represent the significand. An "invisible" bit, i.e. one that is not actually stored is placed in front of the significand with value 1.0. The most significant "visible" bit in the significand has a value of ½, the next bit has a value of ¼ and so on. Thus, the value of the significand is $1.0 \leq significand < 2.0$.

The standard defines some special values: positive infinity ($+\infty$), negative infinity ($-\infty$), negative zero (which is distinct from the 'normal' positive zero), and NaN (not a number). The value of the *exponent* is the standard integer value of the 8 bits used to represent the exponent subtracted by 127. If all 8 bits of the *exponent* are set to 1 and the *significand* is not 0, we get one of the special values $\pm\infty$ depending on the *sign* bit. If all 8 bits of the *exponent* are set to 1 and the *significand* is 0, we get NaN.

Comparisons are mostly intuitive with a few exceptions: negative zero is equal to positive zero, NaN is not equal to anything (including itself), and any finite FP number is strictly greater than $-\infty$ and strictly smaller than $+\infty$.

Floating-point overflow occurs when one tries to represent a number that requires more bits to represent than the format one is converting to. For example, let $maxFloat$ be equal to the largest single precision FP number. If the result of an operation is larger than $maxFloat$, the result turns into $+\infty$, and if the result is smaller than $-maxFloat$, the result turns into $-\infty$.

IEEE-754 Rounding

The IEEE-754 standard requires that basic FP operations are correctly rounded. This means that if the result of a FP operation cannot be represented in the format required, the result is rounded to one of the nearest FP numbers depending on the specified *rounding mode*. The IEEE-754 specifies the following *rounding modes*. The abbreviations below are not standard but are commonly used.

- RNE - Round to the nearest FP number, with ties rounding to the number that ends with even digit. This is most common.
- RNA - Round to the nearest FP number, with ties rounding away from zero.
- RTP - Round towards $+\infty$.
- RTN - Round towards $-\infty$.
- RTZ - Round towards 0.

Rounding Real Numbers

A real number can be rounded upwards or downwards to the nearest (arbitrarily precise) FP number (denoted $\mathbb{F}_p$).

$$\downarrow(\cdot) : r \in \mathbb{R} \rightarrow \max\{f \in \mathbb{F}_p \mid f \leq r\} \tag{2.3}$$

$$\uparrow(\cdot) : r \in \mathbb{R} \rightarrow \min\{f \in \mathbb{F}_p \mid f \geq r\} \tag{2.4}$$

Underflow and Subnormal Numbers

The smallest *normalized* IEEE-754 single precision FP number is 2^{-126}, which is the result of having a *significand* of 1 (by setting the visible bits of the *significand* to 0) and an *exponent* of -126 (which occurs when the binary representation of the *exponent* is 00000001). So, when converting a number smaller than 2^{-126} to float, it will convert to either 0 or 2^{-126} depending on the rounding-mode used. This situation is mirrored when the *sign* bit is set to 1. Such a distinct jump between values is undesirable.

To reduce this abrupt underflow we can use *subnormal* numbers (also known as *denormalized* numbers). If the bits used to represent the exponent are all set to -126, special rules for subnormal numbers apply; the *exponent* is set to 0 and the *significand* no longer has an invisible leading bit, meaning the possible values for the *significand* are now $0.0 \leq significand < 1.0$. The smallest non-zero value for the *significand* is 2^{-23} which is achieved by setting only the least significant bit to 1. Subnormal numbers support a 'gradual underflow' from 2^{-126} to the smallest non-zero subnormal number which is $2^{-126} \times 2^{-23} = 2^{-149}$ before underflowing to 0.

2.1.2 Matrices

A matrix is a rectangular array where each element has some value. In this thesis, we mainly discuss matrices where each element is a real number. These elements are also known as the entries of a matrix. Below is an example of a 3×2 real number matrix.

$$A = \begin{bmatrix} 1.2 & 50.2 & 2 \\ 1.2 & 50.2 & 2 \end{bmatrix} \tag{2.5}$$

The size of a matrix with m rows and n columns is mn, thus the size of A is $2 \cdot 3 = 6$.

Basic Operations

Addition. The sum of two matrices with the same number of row and columns is achieved by summing each entry in an entrywise order.

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} + \begin{bmatrix} 10 & 20 \\ 30 & 40 \end{bmatrix} = \begin{bmatrix} 1+10 & 2+20 \\ 3+30 & 4+40 \end{bmatrix} = \begin{bmatrix} 11 & 22 \\ 33 & 44 \end{bmatrix} \tag{2.6}$$

Scalar multiplication. To multiply some matrix A with some scalar value c, multiply each entry in A with c.

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \cdot 2 = \begin{bmatrix} 1 \cdot 2 & 2 \cdot 2 \\ 3 \cdot 2 & 4 \cdot 2 \end{bmatrix} = \begin{bmatrix} 2 & 4 \\ 6 & 8 \end{bmatrix} \tag{2.7}$$

Multiplication. Let A be a matrix with m rows and n columns. This matrix may be multiplied by another matrix, B, with n columns and o rows. The matrix AB is thus an $m \times o$ matrix, where each entry is the dot product (2.8) of the corresponding row in A and column in B.

$$\begin{aligned} 1 &\leq i \leq m \\ 1 &\leq j \leq o \\ [AB]_{i,j} &= a_{i,1}b_{1,j} + a_{i,2}b_{2,j} + \cdots + a_{i,n}b_{n,j} \end{aligned} \tag{2.8}$$

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \begin{bmatrix} 0 & 4 & 0 \\ 2 & 5 & 7 \end{bmatrix} = \begin{bmatrix} 8 & 33 \\ 10 & 75 \end{bmatrix} \tag{2.9}$$

Row operations. In some matrix algorithms, one may add one row to another, multiply a row with a non-zero constant, and swap two rows in a matrix.

Square matrix. A square matrix is an $n \times n$ matrix, that is a matrix with an equal number of rows and columns.

Identity matrix. An identity matrix is a square matrix where all entries in the main diagonal are 1 and all other entries are zero. For example:

$$I_4 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{2.10}$$

Matrix inversion. A square matrix A is invertible if there exists a matrix B such that $AB = BA = I_n$ where I_n is an $n \times n$ identity matrix. B is known as the inverse of A.

Transpose. The transpose of a matrix flips a matrix by reflecting the matrix over its main diagonal. Transposing the resulting matrix again will give you the original matrix. For example:

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}^T = \begin{bmatrix} 1 & 3 \\ 2 & 4 \end{bmatrix} \tag{2.11}$$

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix}^T = \begin{bmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{bmatrix} \tag{2.12}$$

Submatrix Say we have the following matrix:

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} \tag{2.13}$$

A submatrix of A is constructed by removing any number of rows or columns. For example, removing the 2nd column and the 2nd row of A gives us the following submatrix:

$$\begin{bmatrix} 1 & 3 \\ 7 & 9 \end{bmatrix} \tag{2.14}$$

Triangular matrices. Triangular matrices are special cases of square matrices. A lower triangular matrix is a square matrix where all entries *above* (not including) the main diagonal are zero. An upper triangular matrix is a square matrix where all entries *below* the main diagonal are zero. A strict upper/lower triangular matrix is an upper/lower triangular matrix where all entries above/below and *including* the main diagonal are zero.

LU decomposition. A matrix A may be factorised into the product of some lower triangular matrix L and an upper triangular matrix U. This is known as *LU Decomposition*.

Row and Column Vectors

Matrices with one row or one column are known as row or column vectors respectively. For example, in the following equations, $\mathbf{x}$ and $\mathbf{y}$ are row and column vectors respectively.

$$\mathbf{x} = \begin{bmatrix} x_1 & x_2 & \dots & x_n \end{bmatrix} \tag{2.15}$$

$$\mathbf{y} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix} \tag{2.16}$$

Transpose. The transpose of a column vector is an equivalent row vector and vice versa. For example, the transposes of $\mathbf{x}$ and $\mathbf{y}$ are:

$$\mathbf{x}^T = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} \tag{2.17}$$

$$\mathbf{y}^T = \begin{bmatrix} y_1 & y_2 & \dots & y_n \end{bmatrix} \tag{2.18}$$

Mapping A 'map' is an entrywise application of a function, for example:

$$\begin{aligned} f : a \rightarrow b \\ A = \begin{bmatrix} a_1 & a_2 & a_3 & a_4 \end{bmatrix} \\ map(f, A) := \begin{bmatrix} f(a_1) & f(a_2) & f(a_3) & f(a_4) \end{bmatrix} \end{aligned} \tag{2.19}$$

$$\begin{aligned} &f : a \rightarrow b \\ &A = \begin{bmatrix} a_1 \\ a_2 \\ a_3 \\ a_4 \end{bmatrix} \; map(f, A) := \begin{bmatrix} f(a_1) \\ f(a_2) \\ f(a_3) \\ f(a_4) \end{bmatrix} \end{aligned} \tag{2.20}$$

p-norm Let $p \geq 1$ be a natural number. The p-norm of some vector $\mathbf{x} = (x_1, \dots, x_n)$ is defined as follows:

$$\|\mathbf{x}\|_p := \left(\sum_{i=1}^{n} |x_i|^p \right)^{1/p} \tag{2.21}$$

The 1-norm and 2-norm are also called the taxicab norm and Euclidean norm, respectively. As p approaches ∞, the p-norm approaches the infinity norm, also known as the max norm.

2.1.3 Intervals

Definition

Consider the set $\mathbb{R}$ of real numbers and the set $\mathbb{IR}$ of closed intervals bounded by these numbers. Every closed interval $X \in \mathbb{IR}$ is denoted as $[a, b]$, where $a \leq b$ are real numbers. An interval $X = [a, b]$ is the set of real numbers $\{r \in \mathbb{R} \mid a \leq r \leq b\}$.

Consider the set $\mathbb{IR}^*$ of intervals with potentially unbounded endpoints. Every interval $X \in \mathbb{IR}^*$ has endpoints a and b. a can be either a real number or $-\infty$. b can be a real number such that $a \leq b$ or $+\infty$. If $a = -\infty$, the left endpoint is open. If $b = \infty$, the right endpoint is open. Otherwise, all endpoints are closed. Unless specified otherwise, all intervals in the following are bounded and closed.

Upper and Lower Bounds of Intervals

The upper and lower bounds of intervals are the largest and smallest numbers within the intervals respectively. These are also known as the right and left endpoints of an interval. When the endpoints do not have names, we can obtain them using the following notation:

$$\overline{\cdot} : X \in \mathbb{IR} \mapsto \max\{x \in X\} \tag{2.22}$$

$$\underline{\cdot} : X \in \mathbb{IR} \mapsto \min\{x \in X\} \tag{2.23}$$

Centre and Radius

The centre (midpoint) of some interval is average of the endpoints.

$$c(\cdot) : X \in \mathbb{IR} \mapsto \frac{\underline{X} + \overline{X}}{2} \tag{2.24}$$

The radius of an interval is the distance from the centre of an interval to its endpoints.

$$r(\cdot) : X \in \mathbb{IR} \mapsto c(X) - \underline{X} = \overline{X} - c(X) \tag{2.25}$$

Widths and Boxes

The $width(\cdot)$ of an interval is the number $\overline{X} - \underline{X}$. An n-dimensional box $\mathbf{b} : \mathbb{IR}^n$ where $n \in \mathbb{N}_{>0}$ is a vector consisting of the intervals $X_1 \times \cdots \times X_n$. The $maxWidth(\cdot)$ of some box $\mathbf{b}$ is the maximum of the width of each interval in $\mathbf{b}$. The $taxicabWidth(\cdot)$ of some box $\mathbf{b}$ is the sum of the width of each interval in $\mathbf{b}$.

Centre and Boxes

Let $\mathbf{b} : \mathbb{IR}^n$ be an n-dimensional box consisting of intervals. The centre of the box is a new n-dimensional box of real numbers where each entry is the centre of the respective interval, formally:

$$\begin{aligned} c(\cdot) &: \mathbb{IR}^n \to \mathbb{R}^n \\ c(\mathbf{b}) &:= map(c, \mathbf{b}) \end{aligned} \tag{2.26}$$

where $map(c, \mathbf{b})$ applies the function c on each interval component of the box $\mathbf{b}$.

Note that sometimes, we use a set of variables $vars$ instead of n and, in this case, denote by $\mathbf{b}_v$ the component of $\mathbf{b} \in \mathbb{IR}^{vars}$ corresponding to the variable $v \in vars$.

2.1.4 Interval Arithmetic

Interval arithmetic is an extension of real arithmetic made to work with intervals. Given a binary operation $\diamondsuit \in \{+, -, *, /\}$, some elementary function $\phi : \mathbb{R} \to \mathbb{R}$, and intervals $X, Y \in \mathbb{IR}$, the following definitions apply:

$$\begin{aligned} X \diamondsuit Y &:= hull\{x \diamondsuit y \mid (x, y) \in X \times Y\} \\ \phi(X) &:= hull\{\phi(x) \mid x \in X\} \end{aligned} \tag{2.27}$$

The hull of some set of real numbers $A \subset \mathbb{R}$ is the smallest interval enclosing A.

$$hull(A) := [min(A), max(A)] \tag{2.28}$$

Generalising to n-dimensions, the hull of a closed bounded set $A \subset \mathbb{R}^n$ is the smallest box enclosing A.

$$hull(A) := \Big[[l_1, r_1], \quad \dots \quad , [l_n, r_n] \Big] \tag{2.29}$$

where for each $i = 1, \dots, n$, we have $l_i := min_{x \in A} x_i$, $r_i := max_{x \in A} x_i$

Interval Extension of a Function and the Inclusion Property

Say we have a function $f : D_f \subseteq \mathbb{R}^n \to \mathbb{R}$. F is an interval extension of f as long as the inclusion property (2.30) holds.

$$\begin{aligned} &\forall \mathbf{b} \in \mathbb{IR}^n \text{ with } \mathbf{b} \subseteq D_f \\ &\mathbf{b} \in dom(F) \wedge (\forall x \in \mathbf{b})(f(x) \in F(\mathbf{b})) \end{aligned} \tag{2.30}$$

Interval extensions of functions may be used to approximate the range of a real function as the range of a real function with some domain is a subset of the output of the interval extension of said function with the same domain given as the (box) input. Therefore, $range(f) \subseteq F(D_f)$. Note that $F(D_f)$ is often a bad approximation of $range(f)$

Intervals with FP Endpoints

It is common for an implementation of interval arithmetic to use FP endpoints, for example: $[\downarrow (a), \uparrow (b)]$ where $a \in \mathbb{R} \leq b \in \mathbb{R}$. The results of basic operations are similarly rounded.

Operations on intervals are typically implemented using FP computations. For example, interval addition is defined as:

$$\begin{aligned} \forall x_1, x_2, y_1, y_2 \in& \mathbb{F}_p \\ [x_1, x_2] + [y_1, y_2] =& [\downarrow (x_1 + y_1), \uparrow (x_2 + y_2)] \end{aligned} \tag{2.31}$$

Note that the above is an interval extension of addition. For less basic operations, the implementations often introduce further errors in addition to the rounding errors, but still maintain the inclusion property.

Implementations

There are many implementations of interval arithmetic. An implementation of interval arithmetic is safe as long as the inclusion property (2.30) holds for all of the implemented interval extensions. For details on various implementations of interval arithmetic, see e.g. [48, 13, 38, 58]

2.1.5 Automatic Differentiation

Automatic differentiation (AD) is a set of techniques used to evaluate the derivative of a differentiable function. AD works by applying the chain rule to the operations performed by a differentiable function. AD avoids the inefficiency of both symbolic and numerical differentiation: AD can efficiently work with functions with many inputs and can evaluate higher derivatives. Refer to [36] for more information on AD.

2.1.6 S-expressions

S-expressions (or symbolic expressions) is an expression represented using a tree data structure. S-expressions were created for (and popularized) by the Lisp programming language. S-expressions are classically defined as one of the following (using standard lisp prefix notation):

1. an atom
2. $(\diamond\ x\ y)$ where $\diamond$ is a binary operator.

$1 + 2 \times 3$ is equivalent to the s-expression $(+\ 1\ (\times\ 2\ 3))$. Refer to [47] for more information on S-expressions.

2.2 Interval Methods

2.2.1 Branch-and-prune

Say we have a box $\mathbf{x} \in \mathbb{IR}^n$, some constraint C, and some termination condition $T : \mathbb{IR} \rightarrow \mathbb{B}$ which takes a box and returns a Boolean value. T, for example, may be a function that returns true when $\mathbf{x}$ has a very small width. A branch-and-prune algorithm is a standard algorithm that can be used to approximate the set $[\![C]\!] := \{x \in \mathbf{x} \mid C(x)\}$ with a tolerance depending on T [35]. A generic branch-and-prune algorithm is shown in Algorithm 1. The algorithm uses three variables holding sets of boxes: I, O, and L. I only contains boxes that are guaranteed to be solutions, i.e. entirely in $[\![C]\!]$. O contains boxes, which should not be split according to T, that may or may not be solutions. These are typically found around the boundary of $[\![C]\!]$. In other words, I and $I \cup O$ are inner and outer approximations respectively of a model that satisfies C. L stores boxes that need to be processed by the algorithm and initially contains only $\mathbf{x}$.

Algorithm 1 Generic branch-and-prune algorithm [35]

Input: $(\mathbf{x} : \mathbb{IR}^n, C : \mathbb{R}^n \to \mathbb{B}, T : \mathbb{IR}^n \to \mathbb{B})$
Output: Set of boxes

$I := \emptyset$ — # Set of boxes guaranteed to be solutions
$O := \emptyset$ — # Set of boxes guaranteed to **may** be solutions
initialise L with $\mathbf{x}$ — # Set of boxes that require processing
while $L \neq \emptyset$ **do**
  $\mathbf{y}$:= prune(pick(L), C) — # Here, we **pick** (and remove) a box from L and **prune** it, removing values that violate C
  if $\mathbf{y} \neq \emptyset$ **then**
    if $\mathbf{y}$ satisfies C **then**
      add $\mathbf{y}$ to I
    else if $T(\mathbf{y})$ **then** — # Check if we should stop splitting $\mathbf{y}$
      add $\mathbf{y}$ to O
    else
      split $\mathbf{y}$ and add to L — # Split $\mathbf{y}$ into a union of smaller boxes; add to L
    end if
  end if
end while
return $I \cup O$

As shown in Algorithm 1, we loop on L while it is not empty. We pick and remove a box from L and then contract/reduce it using some *pruning* algorithm, i.e. an algorithm that removes values in the box that do not satisfy the constraint C. The contracted box is assigned to $\mathbf{y}$. If $\mathbf{y}$ is empty (i.e. the chosen box has reduced to $\emptyset$), the chosen box had no solutions that satisfy C. If we are certain that $\mathbf{y}$ satisfies C, then we can add the box to I. If T returns true for $\mathbf{y}$, we cannot split any further and are have not determined whether $\mathbf{y}$ satisfies or violates C, so we add $\mathbf{y}$ to O. Finally, we have the case where T returns false for $\mathbf{y}$ and we are not certain that $\mathbf{y}$ satisfies C, so we *split* $\mathbf{y}$ into a union of smaller boxes and add this union to L. These steps are repeated until L is empty, and we return the union of the set of guaranteed solutions (I) and the set of possible solutions (O), i.e. an outer approximation of $[\![C]\!]$.

2.2.2 Interval Constraint Checking

Given some constraint C, some box $\mathbf{b}$, and some interval function F with $\mathbf{b} \in \mathrm{dom}(F)$, interval evaluation can be used to test whether $F(\mathbf{b})$ either *satisfies* or *contradicts* C. To demonstrate this, say $C \sim f \bowtie 0$ where $\bowtie \in \{\geq, =, \leq\}$. This is equivalent to $f(x) \in A$ where A = [0, 0] for $f(x) = 0$, A = [0, $+\infty$] for $f(x) \geq 0$, and A = [$-\infty$, 0] for $f(x) \leq 0$. Let F be an interval extension of f. C is *certainly satisfied* on the whole box $\mathbf{b}$ if $F(\mathbf{b}) \subseteq A$. C is certainly *contradicted* on the whole box $\mathbf{b}$ if $F(\mathbf{b}) \cap A = \emptyset$.

2.2.3 Newton's Method

Newton's method (2.32) is an iterative root-finding algorithm. Let $f : \mathbb{R} \to \mathbb{R}$ be a differentiable function with one root. Starting from some initial guess $x_0 \in \mathbb{R}$, Newton's method can iteratively produce better approximations of the root of f, eventually converging to the root. With a good initial guess, the rate of convergence of this method is at least quadratic.

$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)} \tag{2.32}$$

Intuitively, one would start at some guess x_0 and draw the tangent of f from the point $f(x_0)$. The root of this tangent line becomes x_1. Repeat these steps until one converges to an acceptable approximation of the root of f.

Limitations

Newton's method is proven to converge at (at least) a quadratic rate as long as some assumptions are met. Without these assumptions, the method may fail to converge.

Bad starting point. Newton's method will eventually converge to the roots as long as the initial guess is close enough to the root and the derivative of the function at the initial guess is not zero. It is important to have a heuristic that chooses a starting point for the Newton Method that increases the likelihood of convergence.

Bad point. The method may reach a point where the derivative is zero. We cannot continue from this point due to division by zero.

Infinite cycles. Some functions, combined with certain starting points, may lead to infinite cycles. For example, consider the function $f(x) = x^3 - 2x + 2$. With an initial guess of $x_0 := 0$, Newton's method gives $x_1 = 1$, $x_2 = 0$, $x_3 = 1$, and so on.

Discontinuous derivative. If the derivative of the function is discontinuous around the root, the method will fail to converge (unless the initial guess is the root).

These issues may be worked around in implementations of the method by, for example, placing limits on the number of iterations, detecting divergence and stopping further iterations, or reattempting the method with another initial guess.

Interval Newton's Method

Newton's method can be combined with interval arithmetic [37]. This gives us a more reliable stopping condition. Sometimes, the method may diverge due to having too small of a precision for the FP numbers used to represent the intervals, though this can easily be resolved by using a higher-precision FP type.

The interval newton method works on a square system of equations. We first discuss the one dimensional case. The (non-interval) Newton method is also often generalised to multiple dimensions.

One Dimension Let $f : \mathbb{R} \to \mathbb{R}$ be a differentiable function over the one-dimensional box $\mathbf{b}$ with at least one root. Let $F'(\cdot)$ be an interval extension of the derivative of f. Assuming $0 \notin F'(\mathbf{b})$, the interval Newton method for one dimensional cases is defined in (2.34).

$$\begin{aligned} \mathbf{b}_0 &= \mathbf{b} \\ \mathbf{b}_{k+1} &= c(\mathbf{b}_k) - \frac{f(c(\mathbf{b}_k))}{F'(\mathbf{b}_k)} \cap \mathbf{b}_k \end{aligned} \tag{2.33}$$

Arbitrary Dimension Say we have a function $f : \mathbb{R}^n \to \mathbb{R}^n$, an n-dimensional box $\mathbf{b}$ and $\mathbf{J}_F$ which is the interval version of the Jacobian[1] of F. Assuming $\mathbf{J}_F(\mathbf{b_k})$ is invertible, the interval Newton method is defined in (2.34).

$$\begin{aligned} \mathbf{b}_0 &= \mathbf{b} \\ \mathbf{b}_{k+1} &= c(\mathbf{b}_k) - \mathbf{J}_F(\mathbf{b}_k)^{-1} f(c(\mathbf{b}_k)) \cap \mathbf{b}_k \end{aligned} \tag{2.34}$$

Note that if b_k becomes empty, then the method has determined that there are no roots.

Alternatively, one may attempt to solve the following linear system[2] which avoids the need to invert $\mathbf{J}_F$:

$$\begin{aligned} \mathbf{b}_0 &= \mathbf{b} \\ \mathbf{J}_F(\mathbf{b}_k)(\mathbf{b}_{k+1} - \mathbf{b}_k) &= -F(\mathbf{b}_k) \cap \mathbf{b}_k \end{aligned} \tag{2.35}$$

The non-interval Newton method can also use a similar iterator.

[1] The Jacobian of a vector-valued function is a matrix of said function's first-order partial derivatives.

[2] Linear systems are discussed in Section 2.5

2.3 Constraint Satisfaction Problems

A constraint satisfaction problem (CSP) is a problem where one has some finite collection of constraints over some set of variables. These problems are solved using constraint satisfaction techniques.

A solution for a CSP is an assignment of values to variables that do not violate any constraint. A CSP may have more than one solution. A single solution is called a *feasible point*. The set of all solutions is called the *feasible region*. If a CSP has no solutions, it is *infeasible*.

2.3.1 SAT

A common form of a CSP is the Boolean satisfiability problem, commonly called SAT. A SAT problem is a collection of Boolean formulas with some Boolean variables. The SAT problem is satisfiable if there exists an assignment of Boolean values to variables that result in the formula evaluating to true. For example:

- $a \land \neg b$ is satisfiable, the solution is $a = true, b = false$.
- $a \land \neg a$ is unsatisfiable.

SAT Solvers

A SAT solver is a tool that attempts to decide SAT problems, telling us whether the formula is 'satisfiable' ('sat') or 'unsatisfiable' ('unsat'). SAT solvers are also able to produce 'models' for satisfiable SAT formulas, that is, assignments for variables that lead to a 'sat' result. SAT solvers often translate formulas to CNF (Conjunctive Normal Form) before calling their core solving algorithm.

Davis-Putnam-Logemann-Loveland (DPLL) Algorithm

The DPLL algorithm [22] is an algorithm commonly used to solve SAT formulas in CNF (commonly called CNF-SAT).

The DPLL algorithm is based on a 'backtracking' algorithm: an algorithm that incrementally assigns values to variables, abandoning assignments as soon as they are determined to violate the SAT problem by 'backtracking' to a set of assignments that do not violate the problem. After each assignment, the formula is simplified by removing disjunctions that become true and removing variables from disjunctions that must consequently be false. Thus, if the CNF becomes empty, the CNF is satisfiable. If a disjunction in the CNF becomes empty, every variable in each disjunction was false, thus the disjunction and the CNF are both false.

If the formula after the above simplifications is satisfiable, then the original formula is also satisfiable. If the simplified formula is not satisfiable (but not necessarily unsatisfiable), assign the opposite boolean value to the same variable and repeat the checks. This assignment of opposite boolean values is commonly referred to as the 'splitting' rule.

The DPLL algorithm builds on these two rules with the following:

Unit propagation. If a clause (disjunction) in the CNF contains only a single variable, then that variable must be true, there is no choice to make. Set the variable to true and propagate this assignment throughout the CNF, simplifying clauses where appropriate. For example, if a CNF contains a clause with only the variable a, a must be true. If the same CNF contains clauses such as $a \vee b$, they can be removed as they are trivially true. If the CNF contains clauses such as $\neg a \vee c$, these can be simplified to c (and thus the unit propagation rule may be applied on this simplified clause).

Pure literal elimination. A variable is pure if it has an assignment that causes all clauses containing the variable to become true. After this assignment, we can remove clauses that become true. For example, a is pure in (2.36); setting a to true causes both clauses containing a to become true. Thus, with the pure literal elimination rule, set a to true and remove the two clauses containing a (as they are now trivially true).

$$(a \vee c) \wedge (b \vee \neg c) \wedge (a \vee b \vee d) \tag{2.36}$$

Algorithm 2 DPLL Algorithm

Input: A set of clauses C in CNF and set of variable assignments A
Output: Satisfiability of C with a set of variable assignments.

In the initial call to this algorithm, A is normally empty.

while there exists some unit clause l in C **do**
    C := apply unit propagation rule on C with unit clause l
end while
while there exists a pure variable p in C **do**
    C := apply pure literal elimination rule on C with pure variable p
end while
if C is empty **then**
    return satisfiable with assignments A
else if C contains an empty clause **then**
    return unsatisfiable with assignments A
else
    v := choose some variable in C
    return (DPLL(C where v is set to true, $A \cup \{v = \text{true}\}$) $\vee$
        DPLL(C where v is set to false, $A \cup \{v = \text{false}\}$)) # This recursive call performs 'backtracking' and 'splitting'.
end if

Conflict Driven Clause Learning (CDCL) Algorithm

The CDCL algorithm [45] is an alternative algorithm inspired by the DPLL algorithm. The main benefit of the CDCL algorithm is its use of non-chronological backtracking which reduces the search space. Let S be a SAT formula in CNF. The algorithm can be summarised as follows:

1. Select a variable in S and give it an arbitrary boolean value. Remember this assignment.

2. Apply the unit propagation rule after this assignment and build an implication graph[3].

3. If the assignment and propagation has led to a conflict, use the implication graph to find a conflicting assignment of variables. Derive

[3] An implication graph keeps track of forced assignments due to the unit propagation rule

a clause which is the negation of the conflicting assignments, add the clause to S, then non-chronologically backtrack to the conflicting variable that was assigned first.

4. If there is no conflict, continue again from step 1 until all variables are assigned.

2.3.2 SMT

Satisfiability modulo theories [3] (SMT) are a generalization of the Boolean satisfiability problem (SAT). SMT extend SAT formulas and allows one to express more complex problems involving real numbers, data structures, and so on. Typically, SMT can be used to check the satisfiability of some quantifier-free formula which is defined with some decidable theory, e.g. linear arithmetic, the theory of real closed fields with quantifier eliminations, etc..

The SMT-LIB standard [2] is an international effort that provides a common input language and interfaces for SMT solvers. SMT-LIB also provides an extensive set of benchmarks.

SMT Solvers

SMT solvers are tools used to decide SMT problems. SMT solvers are typically used to aid program verification[4] and are often the main tool used to make decisions in verification frameworks. Typically, SMT solvers are integrated in a black-box manner with verification frameworks either via files or with some solver-specific API.

One method of solving SMT formulas is to translate them into SAT formulas. For example, an 8-bit integer variable could be translated to 8 variables in a SAT formula, with each variable representing a single bit. Basic operations such as plus and minus could be translated into lower level bit-wise operations. This gives us the benefit of using existing SAT solvers, however translating the formula to SAT causes a loss of semantics,

[4] Verification is discussed in detail in Section 2.8.

meaning that the SAT solver has to work 'harder', even for 'easy' problems. For example, the commutative property for integer addition may be lost when translating formula such as $1 + 2 = 2 + 1$ to SAT.

DPLL(T) DPLL(T) [30] is an extension of the DPLL algorithm (see Algorithm 2) that allows reasoning on some arbitrary theory T. The algorithm works by transforming some SMT formula in CNF that includes theory T to a SAT formula and running the DPLL algorithm on the SAT formula. If the DPLL algorithm says the SAT formula is unsatisfiable, then the original SMT formula is also unsatisfiable. If the algorithm returns a satisfiable formula, translate the assigned variables back to their original form and check if there is a contradiction when using theory T. If this assignment is also satisfiable with T (denoted T-satisfiable), the original SMT formula is also satisfiable. If there is a contradiction, add the (transformed) contradiction to the SAT CNF clauses and repeat the algorithm.

2.3.3 Linear Programming

Another type of CSPs involve constraints over variables with nonlinear real inequalities. For example:

$$\begin{aligned} 3x + 2y &\leq 15 \\ x + 2y &\leq 7 \\ y &\leq 4 \\ -x + 2y &\leq 6 \end{aligned} \tag{2.37}$$

(2.37) is a system of linear inequalities. These systems can be solved using techniques used in Linear Programming, also known as Linear Optimisation. For example, phase I of the two-phase simplex algorithm [18] can find a *feasible point* for systems of non-strict real linear inequalities.

2.3.4 Interval Constraint Propagation

For CSPs involving a set of real, potentially nonlinear, constraints over real variables, interval constraint propagation (ICP) [21] may be used to contract

Algorithm 3 Basic DPLL(T) Algorithm. A real world implementation would also return a model for satisfiable results.

Input: An SMT formula S in CNF which uses theory T and a set of clauses C
Output: Satisfiability of S

```
(F, M) := Translate S to a SAT formula F, keep a map M of variables
    created during this translation.
result R and assignments A := DPLL(F, C)
if R is unsatisfiable then
    return S is unsatisfiable
else
    A_o := Translate variables in A back to their original form using M
    if A_o is satisfiable using theory T then
        return M is T-satisfiable          # A model would also be returned here.
    else
        A_b := Minimum conjunction of assignments in A_o that leads to a
    contradiction.          # E.g. If A_o = {x > 0, y > 0, x < 0}, A_b = x > 0 ∧ x < 0
        A_m := Translate A_b into SAT form using M
        Add to C the clause/disjunction arising from the negation of A_m
        return DPLL(T)(S, C)
    end if
end if
```

the domains of the variables, removing values from the domain without removing any value that satisfies the set of constraints. Intuitively, ICP is used to contract the domains of variables so that they contain all values that satisfy the CSP. This helps interval CSP solving algorithms as the search space is (sometimes greatly) reduced.

Rules

Atomic contractors. An atomic contractor is able to reduce domains when they come across a supported constraint. The contractor will reduce the domain of each variable without removing any value that satisfies the constraint. Contractors can be written for many functions and types of constraints. For example, a very simple contractor can be written for constraints of the form $y = sin(x)$; it is clear here that the only values for y that satisfy this constraint must be in the interval $[-1, 1]$. Consider the following equation:

$$x = y + z \tag{2.38}$$

The domain of each variable may be contracted using the domains of the other variables as shown in (2.39). In essence, these are atomic contractors for addition and subtraction.

$$\begin{aligned} dom(x) &:= dom(x) \cap (dom(y) + dom(z)) \\ dom(y) &:= dom(y) \cap (dom(x) - dom(z)) \\ dom(z) &:= dom(z) \cap (dom(x) - dom(y)) \end{aligned} \tag{2.39}$$

Thus, if $x := [0, \infty]$, $y := [3, 5]$, and $z := [2, 10]$, we may use the contractors in (2.39) to contract the domains as shown in (2.40). The atomic contractor shown in (2.39) reduces the domain of x to $[5, 15]$.

$$
\begin{aligned}
x = y + z \implies x \in & [0, \infty] \cap ([3, 5] + [2, 10]) = \\
& [0, \infty] \cap [5, 15] = [5, 15] \\
y = x - z \implies y \in & [3, 5] \cap ([5, 15] - [2, 10]) = \\
& [3, 5] \cap [-5, 13] = [3, 5] \\
z = x - y \implies z \in & [2, 10] \cap ([5, 15] - [3, 5]) = \\
& [2, 10] \cap [0, 12] = [2, 10]
\end{aligned}
\tag{2.40}
$$

Decomposition. If an atomic contractor cannot be applied to a constraint, one can attempt to 'decompose' the constraint by replacing terms with variables until we have a constraint for which an atomic contractor exists. For example, $\sqrt{x} + \sin(xy) \geq 0$ can be decomposed as shown in (2.41).

$$
\begin{aligned}
a &= xy \\
b &= sin(a) \\
c &= sqrt(x) \\
d &= c + b
\end{aligned}
\tag{2.41}
$$

After this decomposition, we can propagate constraints over the new variables using atomic contractors.

$$
\begin{aligned}
a &\in [-\infty, \infty] \\
b &\in [-1, 1] \\
c &\in [0, \infty] \\
d &\in [-1, \infty]
\end{aligned}
\tag{2.42}
$$

Propagation. These rules may be repeatedly applied until no more contraction can occur. ICP will enclose all feasible values for each variable in an interval CSP.

Use in branch-and-prune

ICP may be used as part of a branch-and-prune algorithm as the 'pruning' function. ICP can be used to remove values in a box that are guaranteed to not be in the solution set. When no further 'pruning' can occur, branch on the 'pruned' interval and repeat these steps until either a solution is found or the termination condition is met. If the 'pruning' results in an empty box, then the CSP was unsatisfiable.

2.4 Optimisation Problem

An optimisation problem is the problem of finding the "best" or optimal solution with respect to some constraints. Say we have some function $f : A \to \mathbb{R}$ where A is an arbitrary set, an optimisation problem is finding some input for f that minimises or maximises the output. More formally:

- If the *objective* is to minimise f, find some value $x_0 \in A$ such that $\forall x \in A. f(x_0) \leq f(x)$
- If the *objective* is to maximise f, find some value $x_0 \in A$ such that $\forall x \in A. f(x_0) \geq f(x)$

Since $f(x_0) \geq f(x) \iff -f(x_0) \leq -f(x)$, it is sufficient for an algorithm to only be able to minimise (or maximise) optimisation problems.

2.4.1 Linear Programming

Linear Programming techniques may also be used to solve optimisation problems. The canonical form of a linear program is:

$$\begin{aligned} maximise(\mathbf{c}^T\mathbf{x}) \\ A\mathbf{x} \leq \mathbf{b} \\ \mathbf{x} \geq 0 \end{aligned} \tag{2.43}$$

where $\mathbf{x}$ is a vector whose components are variables whose values are to be determined, $\mathbf{c}$ and $\mathbf{b}$ are given vectors of real numbers[5], and A is a given matrix of real numbers. The function to be optimised (maximised or

[5] Recall that c^T is the transpose of c.

minimised) is called the *objective function* which, in this case, is $\mathbf{c}^T\mathbf{x}$. The two inequalities in (2.43) restrict possible values for the components of $\mathbf{x}$.

Therefore, if one can express an optimisation problem in the form shown in (2.43), one is able to use established linear programming techniques to optimise the problem.

Simplex Method

The simplex method [18], also known as the simplex algorithm, is a popular Linear Programming algorithm. The method has two phases. In Phase I, one finds a *feasible point* for a set of constraints in the form shown in (2.43). The *objective function* is not required for the first phase. In Phase II, one starts with a *feasible point*, and *optimises* it according to some *objective function*. Thus, Phase I is a CSP solver, and Phase II is an optimisation algorithm.

An Example Using Phase I of the simplex method, we can find a *feasible point* for (2.37) (Note that the simplex method assumes that all variables are non-negative). Phase I gives us a *feasible point* where $x = y = 0$. For Phase II, let the *objective function* be $maximise(3x + 5y)$. Phase II gives a value of 29 for the objective function, with $x = 3$ and $y = 4$. If we minimise instead of maximise here, Phase II gives a value of 0 for the objective with $x = y = 0$.

2.5 Solving Systems of Linear Equations

A system of linear equations, also called a linear system, is a collection of one or more linear equalities that involve the same variable. The following is an example of a linear system:

$$\begin{aligned} 3x + 2y + z &= 10 \\ -2x + z &= 5 \\ y/2 + z &= 14/5 \end{aligned} \tag{2.44}$$

A linear system may have a 'solution'; assigning values to variables that satisfy all equalities. For example, the assignments $x = {-23}/{5}$, $y = 14$, and $z = {-21}/{5}$ is valid for all equations in 2.44.

A linear system may have infinitely many solutions, a unique solution, or no solutions. The set of all possible solutions is called the solution set. If a solution exists for some linear system, the system is *feasible*. A linear system with no solutions is *infeasible*.

2.5.1 General Forms

Let $m, n \in \mathbb{N}_{>0}$, and say we have m equations with n variables. This is written generally as:

$$\begin{aligned} &c_{11}x_1 + c_{12}x_2 + \cdots + c_{1n}xn + b_1 &&= 0 \\ &c_{21}x_1 + c_{22}x_2 + \cdots + c_{2n}xn + b_2 &&= 0 \\ &\vdots \\ &c_{m1}x_1 + c_{m2}x_2 + \cdots + c_{mn}xn + b_m &&= 0 \end{aligned} \tag{2.45}$$

In (2.45), $c_{11}, c_{12}, \ldots, c_{mn}$ are coefficients of the system. $b_1, b_2, \ldots, b_m$ are constants. Both the coefficients and constants are real numbers. At least one of the coefficients must be non-zero.

2.5.2 Vector Equation

A linear system may also be written as a vector equation:

$$x1 \begin{bmatrix} c_{11} \\ c_{21} \\ \vdots \\ c_{m1} \end{bmatrix} + x2 \begin{bmatrix} c_{12} \\ c_{22} \\ \vdots \\ c_{m2} \end{bmatrix} + \cdots + x_n \begin{bmatrix} c_{1n} \\ c_{2n} \\ \vdots \\ c_{mn} \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_m \end{bmatrix} \tag{2.46}$$

2.5.3 Matrix Equation

(2.46) is equivalent to the equation $A\mathbf{x} = \mathbf{b}$ where A is an $m \times n$ matrix and both $\mathbf{x}$ and $\mathbf{b}$ are column vectors, as follows:

$$A = \begin{bmatrix} c_{11} & c_{12} & \dots & c_{1n} \\ c_{21} & c_{22} & \dots & c_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ c_{m1} & c_{m2} & \dots & c_{mn} \end{bmatrix} \mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} \mathbf{b} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_m \end{bmatrix} \tag{2.47}$$

2.5.4 Gauss-Seidel Method

Let $n \in \mathbb{N}_{>0}$. The Gauss-Seidel method is an iterative method that can be used to solve a square linear system[6] of n equations. The Gauss-Seidel method may be used to solve square systems of linear equations arising from a variant of the Newton method that avoids inverting the Jacobian matrix (i.e. the non-interval version of (2.35)).

Let $\mathbf{x}$ and $\mathbf{b}$ be column vectors with n entries. So A, $\mathbf{x}$, and $\mathbf{b}$ have the form shown in (2.47) where $n = m$.

We consider a square linear system: $A\mathbf{x} = \mathbf{b}$. The Gauss-Seidel method is defined recursively:

$$L^{k+1} = \mathbf{b} - U_* \mathbf{x}^k. \tag{2.48}$$

L and U_* are lower triangular and strictly upper triangular matrices

[6] A square linear system is one where the matrix, denoted as A in (2.47), is square.

derived from A, thus:

$$\begin{aligned} A &= L + U_* \\ L &= \begin{bmatrix} c_{11} & 0 & \dots & 0 \\ c_{21} & c_{22} & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ c_{n1} & c_{n2} & \dots & c_{nn} \end{bmatrix} \\ U_* &= \begin{bmatrix} 0 & c_{12} & \dots & c_{1n} \\ 0 & 0 & \dots & c_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & 0 \end{bmatrix} \end{aligned}$$

Using these definitions, we can rewrite the linear system:

$$\begin{aligned} A\mathbf{x} &= \mathbf{b} \\ L\mathbf{x} + U_*\mathbf{x} &= \mathbf{b} \\ L\mathbf{x} &= \mathbf{b} - U_*\mathbf{x}. \end{aligned}$$

The method now solves for $\mathbf{x}$ on the left-hand side using the previous value for $\mathbf{x}$ on the right-hand side:

$$\mathbf{x}^{k+1} = \frac{\mathbf{b} - U_*\mathbf{x}}{L}.$$

Matrix equations of the form $LU\mathbf{x} = \mathbf{b}$ where L and U are *lower* and *upper* triangular matrices, respectively, and $\mathbf{x}$ and $\mathbf{b}$ are column vectors can be solved using an iterative process called *forward substitution*. Thus, we can compute each element of $\mathbf{x}^{k+1}$ as follows:

$$x_i^{k+1} = \frac{b_i - \sum_{j=1}^{i-1} c_{ij} x_j^{k+1} - \sum_{j=1+1}^{n} c_{ij} x_j^k}{c_{ii}}. \tag{2.49}$$

2.6 Haskell Basics

In Chapter 3, we describe some concepts using Haskell syntax. The syntax used is quite intuitive, but to help readers unfamiliar with Haskell, we briefly introduce them here.

In listing 2.6, we use the `data` keyword to define a custom data type `Tree`. `Tree` has two constructors, `Leaf` which constructs a one-node tree holding the given integer value, and `Branch` which takes two Trees as parameters and returns another Tree. One may use this data type to construct a binary tree that stores integers.

```
data Tree = Leaf Integer | Branch Tree Tree
```

Listing 2.6 shows the syntax used to define lists or arrays. Here, `x` is a list that has three entries. `y` is a list of lists that contains two ‘inner’ lists.

```
x = [1, 2, 3]
y = [[1, 2], [3]]
```

2.7 Solving Numerical CSPs

When proving problems made up of numerical constraints, it is common for a prover to make use of symbolical or numerical techniques. For example, consider the following trivial equation:

$$1 + 2 + 3 = 3 + 2 + 1 \tag{2.50}$$

A prover using symbolical techniques would most likely have a rule regarding the commutative property of addition and easily deduce that 2.50 is true. A prover using numerical techniques would evaluate the program and understand that both 1 + 2 + 3 and 3 + 2 + 1 are equal to 6, trivially verifying the example with 6 = 6. Note that some provers employ both numerical and symbolical techniques.

Numerical CSPs may consist of (quantifier-free) nonlinear real arithmetic. These problems can be difficult to solve, though various automated solvers exist, including SMT solvers. We now discuss various commonly used provers for nonlinear real arithmetic.

2.7.1 MetiTarski

MetiTarski [1] is an automated theorem prover that supports the theory of *real closed fields*. It is designed to prove universally quantified inequalities involving nonlinear real functions. MetiTarski supports using the Z3 [49] SMT solver as a backend solver which implements the DPLL(T) algorithms alongside simplex-based linear arithmetic solving techniques.

2.7.2 dReal

dReal [32] is an automated SMT solver for nonlinear real formulas. dReal supports nonlinear arithmetic and transcendental functions. Floating-point numbers can be used as constants.

Formulas containing nonlinear real functions and trigonometric are typically difficult to solve and are, in general, undecidable. dReal implements a δ-complete decision procedure [31] which aims to ease the solving of nonlinear real formulas. For some positive $\delta \in \mathbb{Q}$, a decision procedure is δ-complete for some SMT formula $\varphi \in S$ where S is the set of SMT formulas if the procedure returns either:

- 'unsat' which means φ is unsatisfiable
- ''δ-sat' which means φ^δ is satisfiable.

Where φ^δ is essentially a weakening of φ by numerically relaxing all equalities and inequalities in the formula by δ. For example, if $\varphi \sim \sin(0) = 0$, then $\varphi^\delta \sim |\sin(0)| \leq \delta$.

dReal uses both numerical (RealPaver) and symbolical (OpenSMT) methods in its decision algorithm.

OpenSMT

OpenSMT [12] is an open-source incremental SMT Solver that implements the DPLL(T) algorithm. dReal uses OpenSMT to combine symbolical methods with numerical methods by using DPLL(T) with ICP, thus dReal implements DPLL(ICP).

RealPaver

RealPaver [35] is a tool used to model and solve nonlinear real systems using interval methods. Systems are expressed as sets of equations or inequalities with integer or real variables, i.e. a CSP with nonlinear real and integer arithmetic. dReal uses RealPaver as an ICP solver in it's DPLL(ICP) algorithm.

Solving techniques. RealPaver implements a branch-and-prune algorithm and combines several techniques in its pruning step. Constraint satisfaction techniques and ICP are used to reduce domains of variables by removing values in the domain that violate the constraints. If the CSP contains a square system of equations, a variant of the interval Newton method is used to attempt to solve the system or show that it has no solution on the current box.

Branch-and-prune with ICP. RealPaver combines a branch-and-prune algorithm with ICP techniques. ICP is used to reduce the domains of variables where possible by using constraints given by the system of interest. If ICP cannot be (further) applied on a constraint, RealPaver will branch by splitting on a single variable. RealPaver will then use interval tests to eliminate subdomains where every value violates the constraint. The result of this procedure is a union of the non-eliminated subdomains. This union contains possible solutions for the constraint.

Reduction with an interval Newton method. The pruning method above is combined with a reduction method using a variant of the interval Newton method to process square systems of equations such as $f(x) = 0$. RealPaver uses the interval Newton method to construct a linearisation of the square system. The resulting linear system is then solved sequentially using an interval extension of the Gauss-Seidel method. For more details, refer to [35].

Strategies. We now discuss the branch-and-prune algorithm used in RealPaver and how RealPaver applies the rules we've discussed.

- In it's branch-and-prune algorithm, RealPaver processes boxes in a last-in-first-out manner.
- A box is split by choosing and subdividing one domain. RealPaver has two strategies to pick the domain to subdivide; 'largest first', i.e. choosing the dimension with the largest domain and 'round robin', choosing dimensions in a fixed order. By default, 'round robin' is used.
- If a constraint contains one variable that occurs once, perform ICP.
- If a constraint contains variables which occur more than once, perform the branch-and-prune algorithm.
- If a square system of equation is encountered, perform the described variant of the interval Newton method.

Processing Systems of Linear Inequalities. The authors of RealPaver [35] suggest that systems of inequality constraints can be processed by the simplex method by replacing each nonlinear term with a variable lying in its interval evaluation and using the simplex method to derive an enclosure of the solution set. This is not implemented in RealPaver but has been implemented in LPPaver. Refer to Chapter 3 for more details.

2.7.3 ksmt

ksmt [10] is an automated SMT solver that is able to solve existentially quantified nonlinear constraints in CNF over real numbers, including constraints involving polynomials and certain transcendental functions. ksmt combines symbolical and numerical methods, more specifically combining reliable real computations and resolutions with a CDCL-style algorithm.

ksmt transforms a CNF C into equisatisfiable separated linear form $\mathcal{L} \wedge \mathcal{N}$ where $\mathcal{L}$ is a set of clauses containing only linear terms such as $c_1x_1 + \cdots + c_nx_n + c_0 \diamondsuit 0$ where $c_i \in \mathbb{Q}$ and $\mathcal{N}$ is a set of unit clauses containing only nonlinear literals of the form $x \diamondsuit f(\mathbf{t})$ where f is a nonlinear function, and $\diamondsuit \in \{\leq, <, >, \geq\}$.

The Algorithm

ksmt attempts to find assignments for a chosen variable that keeps $\mathcal{L}$ conflict-free. ksmt keeps a list of these assignments. If an assignment causes a conflict in $\mathcal{N}$, the assignments that caused the conflict is linearised and this linearisation is added to $\mathcal{L}$. ksmt then derives clauses to add to $\mathcal{L}$ to resolve the conflict and 'backjumps' to the maximal prefix of the list of assignments that avoids the conflict. ksmt will also do this 'resolution' and 'backjumping' if a variable cannot be assigned in a way that keeps $\mathcal{L}$ conflict-free. These rules repeat until ksmt can determine the satisfiability of $\mathcal{L} \wedge \mathcal{N}$. Refer to [10, 11], for more details on ksmt, particularly the algorithm and its linearisations.

2.7.4 Colibri

Colibri [46] is an automated SMT solver that uses constraint programming techniques and specialises in the verification of quantifier-free FP SMT-LIB problems. Colibri uses Constraint Programming (CP) and Propagation techniques including linearisations and the simplex method.

Novelties

Colibri uses CP to attribute several domain representations (i.e. integer intervals, FP intervals, known bits) to variables. This representation of the domain is an over approximation of the set of values that the variable can take.

Difference logic [26], a form of domain propagation for difference constraints[7], is used to associate relational attributes to variables by recording an FP 'tailor-made distance' between variables. With the monotonicity property of FP rounding, one can propagate information from one edge of the difference logic global constraints to another

Colibri transforms relations between FP operations into relations on linear rational formulas using relaxations and linearisations. Each operation with (FP) finite arguments is linearised by adding a constraint that bounds the relative error between the FP operation and an analogous operation on reals using the (current) domain of the arguments. The resulting system is solved using the simplex method.

Colibri understands the bit-vector (BV) domain[8], allowing type casts between FP, BV, and Real.

Colibri does not perform bit-blasting[9] on FP variables, allowing the high-level structure of the problem to be preserved. Thus, Colibri can intertwine constraint propagation with simplifications and factorisations that rely on the preserved high-level properties of FP arithmetic. Most SMT solvers cannot do this after a preprocessing step as they typically perform bit-blasting and thus lose this high-level information.

For more information on Colibri, refer to [46]. For more information on linearisations of FP operations, refer to [5].

[7] Difference constraints are constraints of the form $x - y \leq d$.

[8] A data structure that holds only bits. Can be used to store the bit value of some FP number, for example.

[9] A popular approach to deal with FP problems; bit-blasting transforms an FP problem to the bit-vector domain.

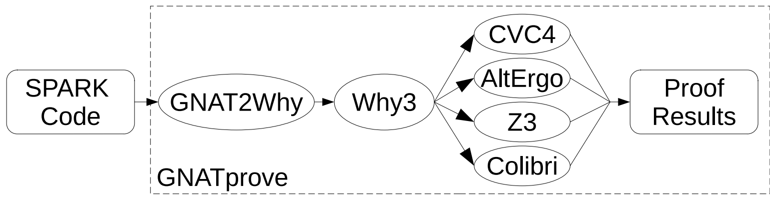


Figure 2.1: Overview of Automated Verification via GNATprove (adapted from [28])

2.8 Formal Verification of FP Programs

As stated in the introduction, formal verification of safety-critical FP programs is important to ensure the program behaves in a precisely specified way. This is important as unnoticed errors in safety-critical programs, particularly those arising from propagation of rounding errors, can lead to catastrophic results. SPARK technology [39] represents the state-of-the-art in industry-standard formal FP software verification[28].

As a language, SPARK is a subset of Ada with a focus on program verification. SPARK technology includes GNATprove, a tool that manages interactions between SPARK, Why3 [7] (described in Section 2.8.1), and a selection of bundled SMT solvers (Alt-Ergo [14], Colibri [46], CVC4 [4], Z3 [49]) as shown in Figure 2.1. If desired, one may use more powerful interactive provers such as Coq [6] and Isabelle [51].

2.8.1 Why3

Why3 is a program verification tool that provides a rich language, WhyML, for writing and specifying programs.

Why3 derives Verification Conditions (VCs) from these programs using the standard weakest-precondition calculus [23] and uses external provers to discharge VCs. WhyML can both be used as a primary programming language but is more commonly used as an intermediary for verification of C, Java, or Ada programs: Why3 can translate derived VCs into a supported input for external provers. This translation may include transformation that eliminate features unsupported in the chosen prover. Users may also apply transformations using Why3 to simplify VCs.

GNATprove integrates with Why3 by translating SPARK programs into WhyML programs. Why3 then derives VCs and then uses the provers bundled with SPARK to discharge them. Why3 plays a key role in SPARK as well as other toolchains, effectively harnessing available solvers and provers for software verification.

2.8.2 Writing and Specifying FP Programs in SPARK

Consider the functional specification of a sine approximation given in the introduction and restated here:

$$|\sin_{fp}(x) - \sin(x)| \leq 0.0001 \tag{2.51}$$

Let $\sin_{fp}$ be a Taylor series approximation of sine to the 3rd degree. With some bound on the input, one could verify the following:

$$x \in [-0.5, 0.5] \implies |\sin_{fp}(x) - \sin(x)| \leq 0.00025889 \tag{2.52}$$

Verifying this is an example of auto-active verification [43], i.e. automated proving of inline specifications such as post-conditions and loop invariants. A SPARK implementation of $\sin_{fp}$ is shown in Listing 2.1 and a SPARK specification equivalent to 2.52 is shown in Listing 2.2

2.8.3 Verifying a Sine Approximation in SPARK

With the specification shown in Listing 2.1, the SPARK toolchain automatically verifies absence of overflow in the `Taylor_Sin` function. This is not difficult since the input `X` is restricted to the small domain $[-0.5, 0.5]$. However, the current SPARK toolchain and other frameworks we know of are unable to automatically verify that the result of `Taylor_Sin(X)` is close to the exact $\sin(\texttt{X})$. Part of the problem is that the VCs feature a mixture of exact real and FP operations. For example, in the VCs derived from Listing 2.1, the

Listing 2.1: Sine approximation in Ada

```
function Taylor_Sin (X : Float) return Float is
  (X - ((X * X * X) / 6.0));
```

Listing 2.2: SPARK formal specification of `Taylor_Sin`

```
function Taylor_Sin (X : Float) return Float with
  Pre => X >= -0.5 and X <= 0.5,
  Post =>
    -- Real_Sin is a non-implemented function with an axiomatic specification
    abs(Real_Sin(Rf(X)) - Rf(Taylor_Sin'Result))
                                <= Ri(25889) / Ri(100000000);
                                -- 0.00025889
```

result of the `Taylor_Sin` function is encoded as

$$X \ominus ((X \otimes X \otimes X) \oslash 6.0);$$

where $\ominus$, $\otimes$, and $\oslash$ are FP subtraction, multiplication, and division, respectively. Although SPARK has some support for FP verification [28, 24], automatically verifying (2.52) requires further work.

We briefly discuss various approaches to automatically verify specification of FP programs and why they are not able to verify our `Taylor_Sin` function.

Why3 Axiomatization

Why3 includes a formalization of the FP IEEE-754 standard [40]. For SMT solvers that natively support FP operations, this formalization is mapped to the SMT-LIB FP theory, and for SMT solvers that do not support FP operations, an axiomatization of the formalization is given [28]. This approach is currently unable to verify our `Taylor_Sin` function, as SMT solvers and their FP theories are not yet sufficiently powerful to decide problems with mixed nonlinear real expressions and FP operations.

Thorough Auto-active approach

One possible method to verify specifications of problems that include FP operations is a more thorough auto-active approach through ghost code[10]. This method has previously been used to verify absence of overflow errors and functional specifications on basic FP functions such as computing a weighted average [24], though it requires more manual work; 59 lines of code required 'a bit less than 400 lines of ghost code' [24] to verify.

Colibri

We ran Colibri on all VCs produced by the `Taylor_Sin` example and it tends to outperform the SMT solvers included in SPARK in both verification speed and the number of problems it can verify.

Colibri was not able to verify the final post-condition in Listing 2.2.

2.8.4 Alternatives to SPARK

While SPARK is a state-of-the-art tool for FP software verification tried and tested in industry. There exist other, similar, frameworks for other languages, namely Frama-C [16] for C programs and Krakatoa [27] for Java programs.

Both Frama-C and Java allow for writing specifications of C and Java programs, respectively. Both frameworks support Jessie [44], a tool which is able to integrate with Why3 (in a similar manner to GNATprove) by translating Java or C programs into WhyML programs. Why3 will then derive verification conditions and use various solvers to discharge them as shown in Figure 2.2.

[10] Ghost code is code that does not affect any implementation, i.e. code that is only used in specifications.

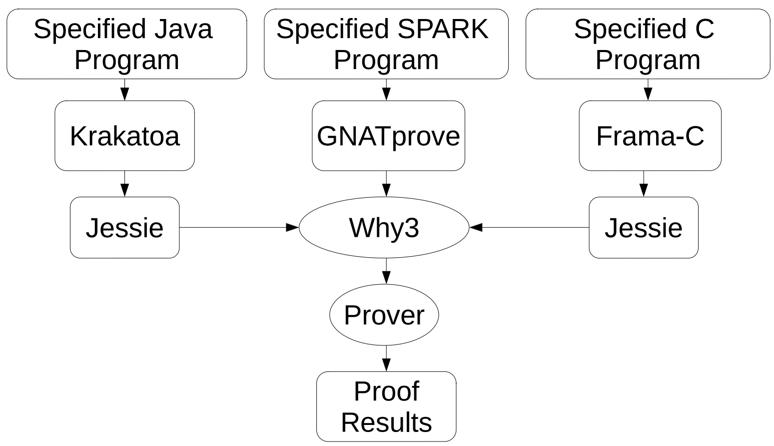


Figure 2.2: Overview of Automated Verification via Frama-C/SPARK/Krakatoa (adapted from [44])

Chapter 3

LPPaver

LPPaver is an automatic numerical prover for nonlinear mixed real/integer formulas. LPPaver uses interval methods along with modified branch-and-prune algorithms. Currently, there is one branch-and-prune algorithm that focuses on proving that some constraint is unsatisfiable over some box and another branch-and-prune algorithm that focuses on finding a model for some constraint over some box.

3.1 Input

LPPaver reads a subset of the standard SMT-LIB language. Here, we describe the abstract syntax of the supported expressions used internally in LPPaver. We use a small subset of Haskell syntax in these definitions.

```haskell
data BinOp = Add | Sub | Mul | Div | Pow | Mod | Min | Max
data UnOp  = Negate | Sqrt | Sin | Cos | Abs
```

LPPaver can encode the following rounding modes:

```haskell
data RoundingMode = RNE | RTP | RTN | RTZ | RNA
-- RNE - Round to nearest, with ties rounding to the nearest even digit
-- RTP - Round up towards +∞
-- RTN - Round down towards −∞
-- RTZ - Round towards zero
-- RNA - Round to nearest, with ties rounding away from zero
```

With these data types, we can describe the symbolic expressions used within LPPaver:

```
data E =
  EBinOp          BinOp        E        E |
  EUnOp           UnOp         E          |
  PowI            E            Integer    | -- E^Integer
  Float32         RoundingMode E          | -- rnd32(RoundingMode, E)
  Float64         RoundingMode E          | -- rnd64(RoundingMode, E)
  RoundToInteger  RoundingMode E          | -- to_int(RoundingMode, E)
  Lit             Rational                |
  Var             String
```

Now, we can encode certain expressions for LPPaver using Haskell expressions with data type `E`. For example, $2\sin(x)$ can be encoded as `EBinOp Mul (Lit 2.0) (EUnOp Sin (Var "x"))`.

We now define data type `F` which is used to encode formulas featuring comparisons of `E` expressions.

```
data Comp = Gt | Ge | Lt | Le | Eq
data Conn = And | Or | Impl

data F =
  FComp Comp E E |
  FConn Conn F F |
  FNot       F   |
  FTrue          |
  FFalse
```

For example, with data type `F`, we can encode $\text{false} \vee 2\sin(x) \geq \sin(x)$ as:

```
FConn Or
  FFalse
  (FComp Ge
    (EBinOp Mul (Lit 2.0) (EUnOp Sin (Var "x")))
    (EUnOp Sin (Var "x")))
```

When LPPaver reads some SMT-LIB input, it parses the input into data type `F`. LPPaver then transforms the input into disjunctive normal form (DNF)[1], a standard tactic used when attempting to find a model for a formula. In Haskell, we represent a DNF as a list of lists. For example, `[[F]]`, which we call fDNF, is a DNF type where each term is constructed using data type `F`. The inner lists have an implicit `And` between the terms, and the outer list has an implicit `Or` between the inner lists.

3.2 Symbolic Simplifications

More specifically, when given a formula with data type `F`, LPPaver first transforms this formula into an fDNF in a standard manner. To simplify reasoning within the algorithm, the fDNF is then transformed into an eDNF. An eDNF is a DNF where all terms are the inequalities > 0 or ≥ 0 with an expression of type `E` on the left-hand side. These inequalities are represented using the following data type:

```
data EConstraint =
  EStrict     E -- E > 0
  ENonStrict  E -- E >= 0
```

Thus, an eDNF is represented as an element of type `[[EConstraint]]` in Haskell.

3.3 Variable Domains and Boxes

Variable domains are encoded in LPPaver as boxes with rational endpoints for each variable. There is also a variation of boxes that allows one to specify each variable as a real or integer variable, though endpoints are still rational.

For example, we encode the box $x \in [0, 5], y \in [-5, 2.4]$ as follows:

$$[x \in [0.0, 5.0], y \in [-5.0, 2.4]]. \tag{3.1}$$

[1]Note that in some cases, conversion to DNF can cause an exponential growth of the formula. For example, converting $(x_1 \vee y_1) \wedge \cdots \wedge (x_n \vee y_n)$ to DNF will lead to 2^n conjunctions.

If x is an integer variable and y is a real variable, we can encode it as follows:

$$[x \in \mathbb{Z} \cap [0.0, 5.0], y \in [-5.0, 2.4]] \tag{3.2}$$

Typically, when splitting a box, LPPaver splits the domain of a chosen in the middle. For example, if splitting (3.1), we get the following two boxes:

$$\{[x \in [0.0, 2.5], y \in [-5.0, 2.4]], [x \in [2.5, 5.0], y \in [-5.0, 2.4]]\} \tag{3.3}$$

When splitting a box using an integer variable, LPPaver safely rounds the endpoints of the new boxes. For example, if we split (3.2), we get the following boxes:

$$\begin{aligned} \{&[x \in \mathbb{Z} \cap [0.0, 2.0], y \in [-5.0, 2.4]], \\ &[x \in \mathbb{Z} \cap [3.0, 5.0], y \in [-5.0, 2.4]]\} \end{aligned} \tag{3.4}$$

3.4 eDNF Local Simplifications and Bound Derivations

LPPaver's proving algorithms work on each conjunction within the eDNF separately. The outer disjunction is checked in a standard manner, stopping as soon as a conjunction is determined to be true.

When checking each conjunction, in some cases, it may be worth analysing the conjunction to see if the bounds on the variables in the box can be improved. This is done using a bounds derivation algorithm interleaved with some simplification rules, both of which are also implemented in PropaFP and described in Sections 4.1.2 and 4.2. The bounds derivation algorithm may reduce the domains for variables and even remove a variable from the box if, for example, the variable only appears in other conjunctions in the DNF. The bounds derivation algorithm may have led to some tautologies so the conjunction is then simplified. This bounds derivation and simplification is interleaved until there are no more changes in the box or the conjunction.

3.5 Contracting a Box Using Linearisations

We now describe how we create a system of linear inequalities from a conjunction of nonlinear differentiable terms to contract a box. The contractor will remove areas from the box whose values are guaranteed to be false over the given conjunction. First, we describe the creation of a nonlinear system using a box with two variables. Then, we generalise to an arbitrary number of variables. At the end of this subsection, we describe how the two-phase simplex method is used with a system of linear inequalities to contract a box.

3.5.1 System with Two Variables

Let $\mathbf{b}$ be a box with two variables:

$$\mathbf{b} := [\mathtt{x} \in [x_L, x_R], \mathtt{y} \in [y_L, y_R]] \tag{3.5}$$

Since the simplex method assumes each variable is ≥ 0, we transform each variable to account for this, giving us the new box:

$$\mathbf{b}' := [\mathtt{x}' \in [0, x_R - x_L], \mathtt{y}' \in [0, y_R - y_L]] \tag{3.6}$$

Let $x'_R := x_R - x_L$ and $y'_R := y_R - y_L$. We can now define a system that encloses these constraints (note that the teletype font variables are the formal variables of the system):

$$\begin{aligned} \mathtt{x}', \mathtt{y}' &\geq 0 \\ \mathtt{x}' &\leq x'_R \\ \mathtt{y}' &\leq y'_R \end{aligned} \tag{3.7}$$

Let C be a conjunction of constraints with differentiable terms. Let t be a term from C. When linearising t, we need to compute the range of t at both the 'extreme left corner' and 'extreme right corner' of $\mathbf{b}$. We define these corners as:

$$\begin{aligned}\mathbf{b_L} &:= [\mathrm{x} \in [x_L, x_L], \mathrm{y} \in [y_L, y_L]] \\ \mathbf{b_R} &:= [\mathrm{x} \in [x_R, x_R], \mathrm{y} \in [y_R, y_R]]\end{aligned} \tag{3.8}$$

Now, we compute the interval approximations of the values of t at both corners using interval arithmetic. For the linearisations, we also require partial derivatives of t for each variable which we compute using automatic differentiation and an interval version of the Jacobian matrix. Note that since t has one output, the resulting matrix has one row.

$$\begin{aligned}\mathbf{l_t} &:= [\![t]\!](\mathbf{b_L}) \\ \mathbf{r_t} &:= [\![t]\!](\mathbf{b_R}) \\ \mathbf{J_t} &:= \mathbf{J}(t, \mathbf{b})\end{aligned} \tag{3.9}$$

When creating a linear system of inequalities using t, we linearise from both $\mathbf{b_L}$ and $\mathbf{b_R}$. The resulting system, when combined with (3.7), represents a weakening of $t \geq 0$ over the domain $\mathbf{b}$. The following inequalities are derived from linearisations of t.

$$\begin{aligned}0 &\leq \overline{\mathbf{l_t}} + \overline{\mathbf{J_t}} \cdot \begin{bmatrix}\mathrm{x}' - 0 \\ \mathrm{y}' - 0\end{bmatrix} \\ 0 &\leq \overline{\mathbf{r_t}} - \underline{\mathbf{J_t}} \cdot \begin{bmatrix}x'_R - \mathrm{x}' \\ y'_R - \mathrm{y}'\end{bmatrix}\end{aligned} \tag{3.10}$$

The first inequality binds t from the left corner of $\mathbf{b}'$ where $\mathrm{x}' = \mathrm{y}' = 0$. The right-hand side of the inequality is a linearisation of t. To justify this linearisation, let us examine the left corner of $\mathbf{b}'$. Here, the actual value of $[\![t]\!](\mathbf{b_L})$ is $\in \mathbf{l_t}$, and we weaken $0 \leq t$ by specifying $0 \leq \overline{l_t}$. As we move away from the left corner, we multiply the point where we are at with the *upper* bound of the partial derivative for each variable and add the result to $\overline{\mathbf{l_t}}$. This is achieved by multiplying $\overline{\mathbf{J_t}}$ with a vector of all variables being subtracted by the left corner of $\mathbf{b}'$, which is always 0. Visually, one may imagine this linearisation as binding t from 'above'.

Similarly, the second inequality binds t from the right and from 'above'. We start from the right corner of $\mathbf{b}'$, safely weakening the formula $t \geq 0$ at this point with $\overline{r_t}$. Now, as we move away from the *right* corner, we multiply the point we are at with the *lower* bound of the *negated* partial derivatives and add the result to $\overline{r_t}$. As the lower bounds of the derivatives are negated, we are still bounding t from 'above'.

The linearisation (3.10) is repeated for every term in C. All of these linearisations are compiled in one system along with the box reformulation (3.7). The system, which is a weakening of the original nonlinear conjunction, is solved and optimised by the two phase simplex method as described at the end of Section 3.5.

3.5.2 System with an Arbitrary Number of Variables

It is simple to extend (or shrink) the system described in the previous section to work with an arbitrary number of variables. To simplify the presentation of this system, we use variables x_1, x_2, etc. instead of x, y, etc. Similarly, x_L and y_L from (3.5) become x_{1L} and x_{2L}.

Let C be a conjunction of differentiable `EConstraint` terms and $\mathbf{b}$ be a box with an arbitrary number of variables:

$$\mathbf{b} := [\mathrm{x_1} \in [x_{1L}, x_{1R}], \mathrm{x_2} \in [x_{2L}, x_{2R}], \ldots, \mathrm{x_n} \in [x_{nL}, x_{nR}]] \tag{3.11}$$

As in (3.6), we transform the box so that all variable domains are ≥ 0:

$$\mathbf{b}' := [\mathrm{x}'_1 \in [0, x_{1R} - x_{1L}], \mathrm{x}'_2 \in [0, x_{2R} - x_{2L}], \ldots, \mathrm{x}'_\mathrm{n} \in [0, x_{nR} - x_{nL}]] \tag{3.12}$$

We define a system to enclose these constraints as in (3.7). In this system, we have $x'_{1R} := x_{1R} - x_{1L}$, and similarly for x'_{2R}, x'_{nR}, etc.

$$\begin{aligned} \mathrm{x}'_1, \mathrm{x}'_2, \ldots, \mathrm{x}'_\mathrm{n} &\geq 0 \\ \mathrm{x}'_1 &\leq x'_{1R} \\ \mathrm{x}'_2 &\leq x'_{2R} \\ &\vdots \\ \mathrm{x}'_\mathrm{n} &\leq x'_{nR} \end{aligned} \tag{3.13}$$

We now linearise each term in C. First, we require the 'extreme' left and right corners of b:

$$\begin{aligned}\mathbf{b_L} &:= [\mathrm{x_1} \in [x_{1L}, x_{1L}], \mathrm{x_2} \in [x_{2L}, x_{2L}], \ldots, \mathrm{x_n} \in [x_{nL}, x_{nL}]] \\ \mathbf{b_R} &:= [\mathrm{x_1} \in [x_{1R}, x_{1R}], \mathrm{x_2} \in [x_{2R}, x_{2R}], \ldots, \mathrm{x_n} \in [x_{nR}, x_{nR}]]\end{aligned} \tag{3.14}$$

Now we compute the interval value of each term at $\mathbf{b_L}$, the interval value of each term at $\mathbf{b_R}$, and partial derivatives for each term over b as shown in (3.9). As in (3.10), we linearise each term, weakening it with the goal of contracting b. This linearisation is shown in (3.15).

$$\begin{aligned}0 &\leq \overline{\mathbf{l_t}} + \overline{\mathbf{J_t}} \cdot \begin{bmatrix} \mathrm{x_1}' - 0 \\ \mathrm{x_2}' - 0 \\ \vdots \\ \mathrm{x_n}' - 0 \end{bmatrix} \\ 0 &\leq \overline{\mathbf{r_t}} - \underline{\mathbf{J_t}} \cdot \begin{bmatrix} x'_{1R} - \mathrm{x_1}' \\ x'_{2R} - \mathrm{x_2}' \\ \vdots \\ x'_{nR} - \mathrm{x_n}' \end{bmatrix}\end{aligned} \tag{3.15}$$

We combine the linearisations from (3.15) done for each term with the reformulation of the box shown in (3.12) to create a system of linear inequalities which is a weakening of C with respect to b. A one-dimensional example of this linearisation for a single term is shown in Figure 3.1 The system is solved using the two-phase simplex method as described below.

3.5.3 Calling the Simplex Method

Let s be a system of linear inequalities as described above. We optimise s using the two-phase simplex method. We first perform phase 1, finding a feasible point for the system. If phase 1 determines that the system is infeasible, we return the empty box. Otherwise, we optimise each variable. This is done by calling phase 2 of the simplex method twice for each variable, using the objective function to minimise and maximise each variable. The

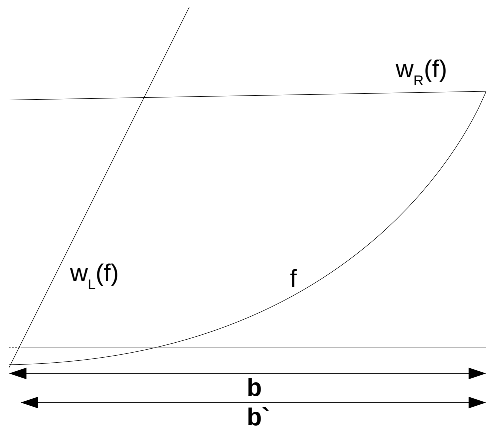


Figure 3.1: Linearisations that weaken a term whose function graph is f over the 1-dimensional box $\mathbf{b}$. The lines labelled $\mathrm{w_L}(\mathrm{f})$ and $\mathrm{w_R}(\mathrm{f})$ are the linearisations made from the left and right 'extreme' corners of $\mathbf{b}$, respectively. This linearisation allows one to safely contract $\mathbf{b}$ by a small amount from the left, giving us the new box $\mathbf{b}'$.

results from phase 2 are used to create an optimised box, cutting off areas which are definitely unsatisfiable for the given box and conjunction. The resulting box is called $\mathbf{r}'$.

Since a transformed box where all variables have a lower bound of 0 was used to create the constraints on variable domains, we need to transform $\mathbf{r}'$ appropriately. For example, in (3.12), $\mathbf{b}'$ was created by subtracting the lower endpoint of each variable in $\mathbf{b}$ in (3.11) from both endpoints of each variable. We 'undo' this subtraction in $\mathbf{r}'$ by adding $x_{1L}, x_{2L}, \ldots, x_{nL}$ to each variable $x_1, x_2, \ldots, x_n$ in $\mathbf{r}'$. The resulting box, named $\mathbf{r}$, is a contraction of the *original* box using the conjunction used to create s.

3.5.4 Soundness

We now proceed by proving that the linearisations described in this section soundly weakens a conjunction over some box. We must first prove that the linearisations soundly weaken a differentiable term over some box.

Lemma 3.5.1 (Soundness of using linearisations)**.** *For every differentiable* *term within an* `EConstraint` t *and for every box* $\mathbf{b}$*, let* s *be the system of linear inequalities produced by linearisation of* t *over* $\mathbf{b}$ *using the (3.15) linearisations. The system consists of two inequalities. Let* $e_{(W,1)}$ *and* $e_{(W,2)}$ *be the first and second inequalities in* s*, respectively. The following statements hold:*

1. $\forall x \in \mathbf{b}.t(x) \geq 0 \implies (e_{(W,1)}(x) \wedge e_{(W,2)})$.
2. $\forall x \in \mathbf{b}.t(x) > 0 \implies (e_{(W,1)}(x) \wedge e_{(W,2)})$.

Proof outline. Assume that the `EConstraint` we have is $t \geq 0$. From Section 3.5.2, we know that both $e_{(W,1)}$ and $e_{(W,2)}$ is a weakening of $t \geq 0$ over $\mathbf{b}$, so $\forall x \in \mathbf{b}.t(x) \geq 0 \implies (e_{(W,1)}(x) \wedge e_{(W,2)})$. Thus, using the (3.15) linearisations as described in Section 3.5.2 soundly weakens some $t \geq 0$ over some box where t is differentiable.

For the case where $t > 0$, first weaken this to $t \geq 0$. The rest of the proof is the same as above. $\square$

Now that we know that the linearisations in Section 3.5.2 soundly weaken a differentiable term over some box, we discuss how the same linearisations can be used to create a system of linear inequalities which represents a weakening of a conjunction of terms over some box.

Corollary 3.5.1.1 (Soundness of using linearisations to weaken a conjunction of `EConstraint`s)**.** *For every conjunction* C *:* [`EConstraint`] *and for every box* $\mathbf{b}$*, let* s *be the system of linear inequalities produced by the linearisation of every term in* C *over* $\mathbf{b}$ *using the (3.15) linearisations as described in Section 3.5.2. Let* C_W *be the* [`EConstraint`]*s equivalent to* s*. The following statement holds:*

$$\forall x \in \mathbf{b}.C(x) \implies C_W(x) \tag{3.16}$$

Proof outline. Linearise each differentiable term in C as done in (3.15). These linearisations soundly weaken each differentiable term as proven in Lemma 3.5.1. Discard the non-differentiable terms. Since C is a conjunction, we combine each system of linear inequalities into one system. Let C_W be an [`EConstraint`] version of the system. Let x be an arbitrary value from

$\mathbf{b}$. Since each term in C was either discarded or soundly weakened with respect to $\mathbf{b}$, $C(x) \implies C_W(x)$. $\square$

Now that we know that the Section 3.5.2 produces a system of linear inequalities which soundly weakens a conjunction of terms over some box, we discuss how the optimisations we perform over this system is a sound pruning of said box where only values which violate the conjunction are removed.

Lemma 3.5.2 (Soundness of pruning a box for some conjunction using linearisations)**.** *For every conjunction C : $[\texttt{EConstraint}]$, and for every box $\mathbf{b}$, let $\mathbf{b}_\mathrm{P}$ be the box resulting from optimising over the linearisation of C as described in Section 3.5.3. The following statements hold:*

$$\begin{aligned} \mathbf{b}_\mathrm{P} &\subseteq \mathbf{b} \\ \forall x \in \mathbf{b} \setminus \mathbf{b}_\mathrm{P}&.\neg C(x) \end{aligned} \tag{3.17}$$

Proof outline. Let C_W be the weakening of C over $\mathbf{b}$ from Corollary 3.8.1.1, The system of linear inequalities which represents C_W is combined with a system which represents $\mathbf{b}$, (3.12). The system is then solved and optimised as described in Section 3.5.3. Let $\mathbf{b}_\mathrm{P}$ be the box resulting from this optimisation.

Since the system was created using $\mathbf{b}$, the bounds for the optimised variables must be within $\mathbf{b}$, so $\mathbf{b}_\mathrm{P} \subseteq \mathbf{b}$.

If the system is infeasible, C_W is false for all values in $\mathbf{b}$, and $\mathbf{b}_\mathrm{P} := \emptyset$. Since $\forall x \in \mathbf{b}.C(x) \implies C_W(x)$, C must be false for all values in $\mathbf{b}$ and $\forall x \in \mathbf{b} \setminus \mathbf{b}_\mathrm{P}.\neg C(x)$ is trivial.

If the system is feasible and optimised, $\mathbf{b}_\mathrm{P} \subseteq \mathbf{b}$ in such a way that $\forall x \in \mathbf{b}_\mathrm{P}.C_W(x)$ and $\forall x \in \mathbf{b} \setminus \mathbf{b}_\mathrm{P}.\neg C_W(x)$. Let x be an arbitrary value in $\mathbf{b} \setminus \mathbf{b}_\mathrm{P}$. Since C_W is a weakening of C over $\mathbf{b}$, and $C_W(x)$ is false, $C(x)$ must also be false.

Thus, the linearisations and optimisations described in Sections 3.5.2 and 3.5.3, respectively, soundly prune away areas in a box which violate some conjunction. $\square$

3.6 Pruning via Interval Methods and Linearisations

We now discuss Algorithm 4 which is our pruning algorithm which uses interval methods and the novel contractors described in Section 3.5. Let $\mathbf{b}_\mathrm{I}$ be a box and C_I be a conjunction of inequalities represented using data type `EConstraint`. The algorithm aims to contract $\mathbf{b}_\mathrm{I}$ by removing 'unsatisfiable' areas, i.e. removing values in $\mathbf{b}_\mathrm{I}$ where C_I does not hold.

Algorithm 4 Prune: contract a box using interval methods and linearisations

Input: $(\mathbf{b}_\mathrm{I}$: box, C_I : [EConstraint])
Output: a box $\mathbf{b}_\mathrm{P} \subseteq \mathbf{b}_\mathrm{I}$ and a conjunction C_F : [EConstraint]

```
C_F := C_I without terms that interval evaluate to true over b_I
C_W := weaken C_F by transforming f > 0 into f ≥ 0
if C_F is empty then
    return (b_I, true)                 # An empty conjunction implies C_I holds over b_I
else if any term in C_F is false for all values in b_I then
    return (∅, C_F)    # An empty box implies at least one term in C_I was false for all values in b_I
end if
C_W^Δ := filter out non-differentiable terms from C_W
b_P := contract b_I using a linearisation of C_W^Δ described in Section 3.5.
if b_P = ∅ then                        # This means that C_W is false over b_P
    return (∅, C_F)
else if |b_I|/|b_P| ≥ 1 + ε_R ∧ |b_I| − |b_P| ≥ ε_A then    # Has b_P reduced significantly?
    return Prune(b_P, C_F)             # Recursive step
else
    return (b_P, C_F)
end if
```

As we are removing unsatisfiable areas, we safely weaken each term in the disjunction by transforming > 0 into ≥ 0 and name this conjunction C_F. We evaluate each term in C_F using interval arithmetic and remove any term in C_F that evaluates to true over the whole box $\mathbf{b}_\mathrm{I}$, i.e. remove any $t \in C_F$ where $[\![t]\!](c) = true$. We name the filtered conjunction C_F. If C_F is empty, then all terms in the conjunction evaluate to true over $\mathbf{b}_\mathrm{I}$. We return $\mathbf{b}_\mathrm{I}$ along with the trivial 'conjunction' $true$. This implies that C_I is true over $\mathbf{b}_\mathrm{I}$. If any term in C_F is false over the whole box $\mathbf{b}_\mathrm{I}$, we return the empty box.

If the algorithm has not yet returned anything, we contract $\mathbf{b}_\mathrm{I}$ by linearising differentiable terms in C_W and solving the resulting system of linear inequalities using the simplex method as explained in Section 3.5. The contracted box is called $\mathbf{b}_\mathrm{P}$. If $\mathbf{b}_\mathrm{P}$ is empty, the contractor has determined that C_W is false for all values in $\mathbf{b}_\mathrm{I}$ so we return the empty box. If $\mathbf{b}_\mathrm{P}$ is significantly smaller than $\mathbf{b}_\mathrm{I}$, i.e., if $|\mathbf{b}_\mathrm{I}|/|\mathbf{b}_\mathrm{P}|$ is greater than or equal to $\varepsilon_\mathrm{R}+1$ for some global $\varepsilon_\mathrm{R} > 0$ *and* if $|\mathbf{b}_\mathrm{I}| - |\mathbf{b}_\mathrm{P}|$ is larger than some global $\varepsilon_\mathrm{A} > 0$, we recursively call the pruning algorithm with parameters $\mathbf{b}_\mathrm{P}$ and C_F. By default, LPPaver uses $\varepsilon_\mathrm{R} = 0.2$ and $\varepsilon_\mathrm{A} = 2^{-100}$. If $\mathbf{b}_\mathrm{P}$ is *not* significantly smaller than $\mathbf{b}_\mathrm{I}$, we stop pruning, returning the box $\mathbf{b}_\mathrm{P}$ and C_F which is the conjunction of terms which are neither completely true nor completely false for all values in $\mathbf{b}_\mathrm{P}$.

3.6.1 Termination

Lemma 3.6.1 (Termination of Prune)**.** *For any box $\mathbf{b}_\mathrm{I}$, for any conjunction C_I, for any $\varepsilon_\mathrm{R} \in \mathbb{Q}^{>0}$, for any $\varepsilon_\mathrm{A} \in \mathbb{Q}^{>0}$, Algorithm 4 terminates.*

Proof outline. Assume that the algorithm recurses. This means that box we are recursing with has shrunk by at least ε_A against the input box. With each recursive call, the box we are recursing with must shrink by at least ε_A. Since the boxes have finite endpoints, eventually, the width of the box will be less than ε_A When this occurs, it is impossible for the box to shrink by, at least, ε_A, as the width of the box cannot become negative, so the algorithm terminates. □

3.6.2 Soundness

Building on the soundness of the linearisations and optimisations discussed in Section 3.5, we now discuss soundness of Prune (Algorithm 4).

Lemma 3.6.2 (Soundness of Prune)**.** *For any box $\mathbf{b}_\mathrm{I}$ and for any conjunction of* `EConstraint`*s C_I, The following holds for the outputs $\mathbf{b}_\mathrm{P}$ and C_F of Algorithm 4*

1. $\mathbf{b}_\mathrm{P} \subseteq \mathbf{b}_\mathrm{I}$

2. $\forall x \in \mathbf{b}_\mathrm{P}.C_I(x) \iff C_F(x)$

3. $\forall x \in \mathbf{b}_\mathrm{I}.C_I(x) \implies x \in \mathbf{b}_\mathrm{P}$

4. $\forall x \in \mathbf{b}_\mathrm{I} \setminus \mathbf{b}_\mathrm{P}.\neg C_F(y)$

Proof outline. The conjunction, C_F, is equivalent to C_I where terms which interval evaluate to true over $\mathbf{b}_\mathrm{I}$ are removed (note that empty conjunctions are trivially true). Let $C_{F,1}$ be the value of C_F after execution of line 1. Clearly, $\forall x \in \mathbf{b}_\mathrm{I}.C_I(x) \iff C_{F,1}(x)$.

If $C_{F,1}$ is empty, then C_I was true for all values in $\mathbf{b}_\mathrm{I}$, so let $\mathbf{b}_\mathrm{P} := \mathbf{b}_\mathrm{I}$ and $C_F := true$. Clearly, $\mathbf{b}_\mathrm{P} \subseteq \mathbf{b}_\mathrm{I}$. Since $\mathbf{b}_\mathrm{I} = \mathbf{b}_\mathrm{P}$, $\forall x \in \mathbf{b}_\mathrm{I}.C_I(x) \implies x \in \mathbf{b}_\mathrm{P}$ is trivial. $\forall x \in \mathbf{b}_\mathrm{P}.C_I(x) \iff C_{F,1}(x)$ is trivial. Since $\mathbf{b}_\mathrm{I} \setminus \mathbf{b}_\mathrm{P} := \emptyset$, $\forall x \in \mathbf{b}_\mathrm{I} \setminus \mathbf{b}_\mathrm{P} \neg C_{F,1}(x)$ is vacuously true. Thus, this branch is sound.

If $C_{F,1}$ is not empty and any term in $C_{F,1}$ is false for all values in $\mathbf{b}_\mathrm{I}$, let $\mathbf{b}_\mathrm{P} := \emptyset$. Clearly, $\mathbf{b}_\mathrm{P} \subseteq \mathbf{b}_\mathrm{I}$. $\forall x \in \mathbf{b}_\mathrm{P}.C_I(x) \iff C_F(x)$ is vacuously true. Since there exists a term in $C_{F,1}$ which is false for all values in $\mathbf{b}_\mathrm{I}$, and $C_{F,1}$ is C_I without terms which are *true* for all values in $\mathbf{b}_\mathrm{I}$, the falsifying term in $C_{F,1}$ must also be in C_I. Since C_I is false for all values in $\mathbf{b}_\mathrm{I}$, $\forall x \in \mathbf{b}_\mathrm{I}.C_I(x) \implies x \in \mathbf{b}_\mathrm{P}$ is vacuously true. $\forall x \in \mathbf{b}_\mathrm{I} \setminus \mathbf{b}_\mathrm{P} \neg C_{F,1}(x)$ is trivial. Thus, this branch is sound.

If neither of these cases occur, let C_W^Δ be a conjunction consisting of all differentiable terms in C_W as shown in line 8 of the algorithm. C_W^Δ is a clear weakening of C_W. If we cannot determine that a term in $C_{F,1}$ is false for all values in $\mathbf{b}_\mathrm{I}$, we contract $\mathbf{b}_\mathrm{I}$ using linearisations of C_W^Δ and optimisations of the resulting system as described in Section 3.5. On line 9, we use this contraction step to produce the box, $\mathbf{b}_{\mathrm{P},\mathbf{9}}$. From Lemma 3.5.2, $\mathbf{b}_{\mathrm{P},\mathbf{9}} \subseteq \mathbf{b}_\mathrm{I}$ and $\forall x \in \mathbf{b}_\mathrm{I} \setminus \mathbf{b}_{\mathrm{P},\mathbf{9}}.\neg C_W^\Delta(x)$.

If $\mathbf{b}_{\mathrm{P},\mathbf{9}}$ is empty, then all values in $\mathbf{b}_\mathrm{I}$ violate C_W^Δ. Let $\mathbf{b}_\mathrm{P} := \mathbf{b}_{\mathrm{P},\mathbf{9}}$. $\mathbf{b}_\mathrm{P} \subseteq \mathbf{b}_\mathrm{I}$ is trivial. $\forall x \in \mathbf{b}_\mathrm{P}.C_I(x) \iff C_F(x)$ is vacuously true. Let x be an arbitrary value from $\mathbf{b}_\mathrm{I}$. Since C_W^Δ is a weakening of C_F over $\mathbf{b}_\mathrm{I}$ (i.e. $C_F(x) \implies C_W^\Delta(x)$), all values in $\mathbf{b}_\mathrm{I}$ must also violate $C_{F,1}$. As $C_{F,1}(x) \iff C_I(x)$, and $C_{F,1}(x)$ is false, it must be true that $C_I(x)$ is false. Thus, $C_I(x) \implies x \in \mathbf{b}_\mathrm{P}$ is vacuously true. Since $\mathbf{b}_\mathrm{P}$ is empty, $\forall x \in \mathbf{b}_\mathrm{I} \setminus \mathbf{b}_\mathrm{P}.\neg C_I(x)$ is trivial. Thus, this branch is sound.

If, after the contraction step, $\mathbf{b}_{\mathrm{P},\mathbf{9}}$ is not significantly smaller than $\mathbf{b}_{\mathrm{I}}$, the algorithm gives $\mathbf{b}_{\mathrm{P},\mathbf{9}}$ as the contracted box. Let $\mathbf{b}_{\mathrm{P}} := \mathbf{b}_{\mathrm{P},\mathbf{9}}$. Since $\forall x \in \mathbf{b}_{\mathrm{I}} \setminus \mathbf{b}_{\mathrm{P}}.\neg C_W^{\Delta}(x)$, $\forall x \in \mathbf{b}_{\mathrm{I}}.C_I(x) \implies C_W^{\Delta}(x)$, and $\mathbf{b}_{\mathrm{P}} \subseteq \mathbf{b}_{\mathrm{I}}$, it is clear that $\forall x \in \mathbf{b}_{\mathrm{I}}.C_I(x) \implies x \in \mathbf{b}_{\mathrm{P}}$. Let x be an arbitrary value from the box $\mathbf{b}_{\mathrm{I}} \setminus \mathbf{b}_{\mathrm{P},\mathbf{9}}$. Since C_W^{Δ} is a weakening of C_F over $\mathbf{b}_{\mathrm{I}} \supseteq \mathbf{b}_{\mathrm{P},\mathbf{9}}$, and $C_W^{\Delta}(x)$ is false, $C_{F,1}(x)$ must be false. Since $C_{F,1}(x) \iff C_I(x)$, $C_I(x)$ is false. This branch is sound.

If $\mathbf{b}_{\mathrm{P},\mathbf{9}}$ is significantly smaller than $\mathbf{b}_{\mathrm{I}}$, we recurse with inputs $\mathbf{b}_{\mathrm{P},\mathbf{9}}$ and $C_{F,1}$. Since $\mathbf{b}_{\mathrm{P},\mathbf{9}}$ is a contraction of $\mathbf{b}_{\mathrm{I}}$ where only values which violate $C_{F,1}$ are removed, and all other branches in the algorithm soundly removes values from a given box which violate a given conjunction, it is sound to recurse with $\mathbf{b}_{\mathrm{P},\mathbf{9}}$. Since $C_{F,1} \iff C_I$ for all values in $\mathbf{b}_{\mathrm{P},\mathbf{9}}$, and $\mathbf{b}_{\mathrm{P},\mathbf{9}} \subseteq \mathbf{b}_{\mathrm{I}}$, both conjunctions have the same truth value over $\mathbf{b}_{\mathrm{P},\mathbf{9}}$ so it is sound to recurse with $C_{F,1}$ (it is also sound to recurse with C_I but this is inefficient). From Lemma 3.6.1, we know that this recursion must eventually terminate. Let $\mathbf{b}_{\mathrm{P},\mathbf{13}}$ and $C_{F,13}$ be the values of the box and conjunction returned from this recursive call, respectively. Since the soundness statements hold in all other branches in prune, it must be true $\mathbf{b}_{\mathrm{P},\mathbf{13}} \subseteq \mathbf{b}_{\mathrm{P},\mathbf{9}} \subseteq \mathbf{b}_{\mathrm{I}}$, $\forall x \in \mathbf{b}_{\mathrm{P},\mathbf{13}}.C_{F,13}(x) \iff C_{F,1}(x) \iff C_I(x)$,
$\forall x \in \mathbf{b}_{\mathrm{I}}.C_I(x) \implies x \in \mathbf{b}_{\mathrm{P},\mathbf{13}}$, and $\forall x \in \mathbf{b}_{\mathrm{I}} \setminus \mathbf{b}_{\mathrm{P},\mathbf{13}}.\neg C_{F,13}(x)$. This branch is sound.

Since all branches are sound, Algorithm 4 is sound. □

3.7 Showing Unsatisfiability via Depth-First Splitting & Pruning

We now describe our branch-and-prune algorithm which focuses on showing unsatisfiability of a conjunction of terms over some box using depth-first paving and pruning using Algorithm 4. Depth-first algorithms are well suited for this task due to their simplicity, low memory usage, and the fact that showing unsatisfiability requires an exhaustive search over the box we are examining. The algorithm is a variation of the branch-and-prune method described in Algorithm 1. The pseudocode for this algorithm can be found in Algorithm 5.

Algorithm 5 Proving with depth-first branching + pruning

Input: ($\mathbf{b}_\mathrm{I}$: box, C_I : `[EConstraint]`, $dMax$: $\mathbb{N}$)
Output: satisfiability of C_I over $\mathbf{b}_\mathrm{I}$, model $\mathbf{m} \subseteq \mathbf{b}_\mathrm{I}$ if C_I is satisfiable

initialise stack L with ($\mathbf{b}_\mathrm{I}$, C_I, 0)
while L is not empty **do**
  ($\mathbf{b}$, C, d) := pick(L) # retrieve a box with a constraint and depth from L
  ($\mathbf{b}_\mathrm{P}$, C_F) := Prune($\mathbf{b}$, C) # $[\![C]\!] \cap \mathbf{b} \subseteq [\![C]\!] \cap \mathbf{b}_\mathrm{P}$
  if $\mathbf{b}_\mathrm{P} \neq \emptyset$ **then**
    if C_F is trivially true **then** # If C_F is true, $\mathbf{b}_\mathrm{P}$ satisfies C_I
      $\mathbf{m}$:= $\mathbf{b}_\mathrm{P}$
      return C_I is satisfied over $\mathbf{m} \subseteq \mathbf{b}_\mathrm{I}$
    else if $d > dMax$ **then** # the termination condition depends on the *depth*
      return satisfiability of C_I undecided, gave up at box $\mathbf{b}_\mathrm{P} \subseteq \mathbf{b}_\mathrm{I}$
    else
      ($\mathbf{b}_\mathrm{P}^\mathrm{L}$, $\mathbf{b}_\mathrm{P}^\mathrm{R}$) := split($\mathbf{b}_\mathrm{P}$) # Bisect the variable with the largest width
      add ($\mathbf{b}_\mathrm{P}^\mathrm{L}$, C_F, $d+1$) and ($\mathbf{b}_\mathrm{P}^\mathrm{R}$, C_F, $d+1$) to L
    end if
  end if
end while
return C_I is unsatisfiable over $\mathbf{b}_\mathrm{I}$

In Algorithm 5, we first take a box $\mathbf{b}_\mathrm{I}$. Then, a conjunction of inequalities C_I represented using data type `[EConstraint]`, so all inequalities are either > 0 or ≥ 0. The algorithm attempts to check whether or not the conjunction holds over $\mathbf{b}_\mathrm{I}$. Finally, we have a number, $dMax$, which specifies the maximum number of times a box will be split.

Now we describe the main body of the algorithm. We have the stack L which stores triples. Each triple contains a box, a constraint, and a depth. Initially, L is set to a triple which stores $\mathbf{b}_\mathrm{I}$, the initial conjunction C_I, and depth 0. We then loop until L is empty.

In the loop, we take a triple (a box named $\mathbf{b}$, a conjunction to consider named C, and the depth of $\mathbf{b}$ which we call d) from L. We pass $\mathbf{b}$ and C to our pruning function, which is described in Algorithm 4. The pruning functions returns a new box, $\mathbf{b}_\mathrm{P}$ and a new, 'filtered' conjunction, C_F, where we have removed terms that have been determined to be true for all values in $\mathbf{b}_{\mathrm{P}}$.

After the pruning if $\mathbf{b}_{\mathrm{P}}$ becomes empty, our pruning has determined that the terms in the conjunction are unsatisfiable over $\mathbf{b}$, so we stop considering $\mathbf{b}$ and start the next iteration of the loop. Otherwise, if C_F is empty, the pruning method has decided that all terms in C are satisfied by any value in $\mathbf{b}_{\mathrm{P}}$, and we can return a satisfiable result with $\mathbf{b}_{\mathrm{P}}$ as a set of models for C. If C_F is not empty and the current depth of the box, d, is greater than $dMax$, we return an 'unknown' result, meaning LPPaver could not decide C_I over $\mathbf{b}_{\mathrm{I}}$ with the given parameters (i.e. the value of $dMax$, precision of interval arithmetic, splitting methods, etc.). $\mathbf{b}_{\mathrm{P}}$, the box where the algorithm gave up, is also returned, which is useful for users as it shows an area where the algorithm found it difficult to decide satisfiability of C. If d is less than or equal to $dMax$, we split $\mathbf{b}_{\mathrm{P}}$ into two smaller boxes by bisecting the domain of a variable with the largest width, rounding endpoints for interval variables as appropriate. The two new boxes, along with C_F and an incremented depth, are added to L.

If the while loop reaches its termination condition, all boxes in L have been processed and none returned a satisfiable/unknown result. Thus, C_I is unsatisfiable on all boxes in L, so C_I is unsatisfiable on $\mathbf{b}_{\mathrm{I}}$.

3.7.1 Termination

Lemma 3.7.1 (Termination of the Depth-First Proving Algorithm). *For any box $\mathbf{b}_{\mathrm{I}}$, for any conjunction C_I, for any $dMax \in \mathbb{N}$, Algorithm 5 terminates.*

Proof outline. The algorithm will loop while L, a stack which contains triples consisting of a box, a conjunction, and a natural number, is not empty. Within the loop, the algorithm picks a triple ($\mathbf{b}$, C, d) from L, where $\mathbf{b}$ is a box, C is a conjunction, and d is a natural number. The algorithm calls Prune which terminates (Lemma 3.6.1). All branches other than the one beginning on line 11 clearly terminate. For the remaining branch, the box $\mathbf{b}$ is split into two and added to L (with a conjunction equivalent to C over $\mathbf{b}$ and $d + 1$). Note that $dMax$ is a depth bound, i.e., $dMax$ specifies the maximum number of times a box can be split, as seen in line 9 of the algorithm. Since we split boxes using a depth bound of $dMax$, and $dMax$ is a finite natural number, the maximum number of boxes that can be added to L is 2^{dMax}. Since every other branch terminates, Algorithm 5 must terminate. $\square$

3.7.2 Soundness

Having established that Algorithm 4 is sound and terminates, and Algorithm 5 terminates we now show soundness of Algorithm 5.

Lemma 3.7.2 (Soundness of the Depth-First Proving Algorithm)**.** *For any box $\mathbf{b}_{\mathrm{I}}$, for any conjunction of* `EConstraint`*s C_I, for any $dMax \in \mathbb{N}$, the following statements regarding the output of Algorithm 5 hold:*

1. *If the output is "C_I is satisfied over $\mathbf{m} \subseteq \mathbf{b}_{\mathrm{I}}$" then $\mathbf{m} \subseteq \mathbf{b}_{\mathrm{I}}$ and $\forall x \in \mathbf{m}.C_I(x)$*
2. *If the output is "satisfiability of C_I undecided, gave up at box $\mathbf{b}_{\mathrm{P}} \subseteq \mathbf{b}_{\mathrm{I}}$" then $\mathbf{b}_{\mathrm{P}} \subseteq \mathbf{b}_{\mathrm{I}}$*
3. *If the output is "C_I is unsatisfiable over $\mathbf{b}_{\mathrm{I}}$" then $\forall x \in \mathbf{b}_{\mathrm{I}}.\neg C_I(x)$*

Proof outline. The algorithm starts by creating a stack of triples, named L, which initially contains only the triple ($\mathbf{b}_{\mathrm{I}}$, C_I, and 0). The algorithm loops while the stack is not empty. Let $\mathbf{b}_{\mathrm{U}}$ be the union of all boxes in L. The loop has the following invariants:

1. $\mathbf{b}_{\mathrm{U}} \subseteq \mathbf{b}_{\mathrm{I}}$.
2. $\forall x \in \mathbf{b}_{\mathrm{I}} \setminus \mathbf{b}_{\mathrm{U}}.\neg C_I(x)$
3. For every $(\mathbf{b}, C, d)$ in L, $\forall x \in \mathbf{b}.C(x) \iff C_I(x)$.

We first prove that these loop invariants hold in the first iteration of the loop. In the first iteration, $L = [(\mathbf{b}_{\mathrm{I}}, C_I, 0)]$. Clearly, $\mathbf{b}_{\mathrm{U}} = \mathbf{b}_{\mathrm{I}}$, $\mathbf{b}_{\mathrm{U}} \subseteq \mathbf{b}_{\mathrm{I}}$ is trivial. $\forall x \in \mathbf{b}_{\mathrm{I}} \setminus \mathbf{b}_{\mathrm{U}}.\neg C_I(x)$ is vacuously true as $\mathbf{b}_{\mathrm{I}} \setminus \mathbf{b}_{\mathrm{U}} = \emptyset$. Since there is only one entry in L, for invariant 3, $C = C_I$, so $\forall x \in \mathbf{b}.C(x) \iff C_I(x)$ is trivial.

We now prove that these invariants hold for any iteration of the loop. Let L_N be the stack in an arbitrary iteration of the loop. Assume that in the iteration leading to L_N, all invariants hold. Let $\mathbf{b}_{\mathrm{U}}$ be the union of all boxes in L_N. Pick ($\mathbf{b}$, C, d) from the top of L_N. Since we have assumed that all loop invariants hold, the following statements must be true:

1. $\mathbf{b} \subseteq \mathbf{b}_{\mathrm{I}}$
2. $\forall x \in \mathbf{b}.C(x) \iff C_I(x)$.

Within the loop, we call the prune algorithm with arguments $\mathbf{b}$ and C. This gives us a new box, $\mathbf{b}_{\mathrm{P}}$, and a conjunction, C_F. From Lemma 3.6.2, $\mathbf{b}_{\mathrm{P}} \subseteq \mathbf{b}$, $\forall x \in \mathbf{b} \setminus \mathbf{b}_{\mathrm{P}}.\neg C(x)$ and $\forall x \in \mathbf{b}.C(x) \iff C_F(x)$.

If $\mathbf{b}_{\mathrm{P}}$ is empty, we continue with the next iteration of the loop. Let L_M be the name for the stack in said iteration. Note that $L_M = L_N$ without the picked triple ($\mathbf{b}$, C, d). Let $\mathbf{b}_{\mathrm{U}}^{\mathrm{M}}$ be the union of all boxes in L_M. Clearly, $\mathbf{b}_{\mathrm{U}}^{\mathrm{M}} \subseteq \mathbf{b}_{\mathrm{U}}$, so $\mathbf{b}_{\mathrm{U}}^{\mathrm{M}} \subseteq \mathbf{b}_{\mathrm{I}}$. Since $\forall x \in \mathbf{b}_{\mathrm{I}} \setminus \mathbf{b}_{\mathrm{U}}.\neg C_I(x)$, $\forall x \in \mathbf{b}_{\mathrm{I}} \setminus \mathbf{b}_{\mathrm{U}}^{\mathrm{M}}.\neg C_I(x)$ is trivial. Since L_M is L_N with one entry removed, the final loop invariant is trivial. Thus, in this branch, all invariants in the next iteration of the loop hold.

If $\mathbf{b}_{\mathrm{P}}$ is not empty, we do the following. If C_F is trivially true, prune has decided that C is true for all values in $\mathbf{b}$, so we stop the algorithm. Similarly, if $d > dMax$, we stop the algorithm. In both cases, there is no further loop.

In the final branch, we split $\mathbf{b}_{\mathrm{P}}$ into two smaller boxes, $\mathbf{b}_{\mathrm{P}}^{\mathrm{L}}$ and $\mathbf{b}_{\mathrm{P}}^{\mathrm{R}}$ such that the union of $\mathbf{b}_{\mathrm{P}}^{\mathrm{L}}$ and $\mathbf{b}_{\mathrm{P}}^{\mathrm{R}}$ is equal to $\mathbf{b}_{\mathrm{P}}$. Clearly, $\mathbf{b}_{\mathrm{P}}^{\mathrm{L}} \subseteq \mathbf{b}_{\mathrm{P}} \subseteq \mathbf{b} \subseteq \mathbf{b}_{\mathrm{I}}$ and similar for $\mathbf{b}_{\mathrm{P}}^{\mathrm{R}}$. Since both $\mathbf{b}_{\mathrm{P}}^{\mathrm{L}}$ and $\mathbf{b}_{\mathrm{P}}^{\mathrm{R}}$ are subboxes of $\mathbf{b}_{\mathrm{P}}$, and $\forall x \in \mathbf{b} \setminus \mathbf{b}_{\mathrm{P}}.\neg C(x)$, it must be true that $\forall x \in \mathbf{b} \setminus \mathbf{b}_{\mathrm{P}}^{\mathrm{L}}.\neg C(x)$ and similar for $\mathbf{b}_{\mathrm{P}}^{\mathrm{R}}$. These subboxes are added to L_N along with C_F and an incremented depth counter. Recall that C_F has the same truth value as C over $\mathbf{b}$ (Lemma 3.6.2).

Let $\mathbf{b}_{\mathrm{U}}^{\mathrm{M}} := \mathbf{b}_{\mathrm{U}} \cup \mathbf{b}_{\mathrm{P}}^{\mathrm{L}} \cup \mathbf{b}_{\mathrm{P}}^{\mathrm{R}}$ Since we know that $\mathbf{b}_{\mathrm{U}} \subseteq \mathbf{b}_{\mathrm{I}}$, and $\mathbf{b}_{\mathrm{P}}^{\mathrm{L}} \cup \mathbf{b}_{\mathrm{P}}^{\mathrm{R}} \subseteq \mathbf{b}_{\mathrm{I}}$, it must be true that $\mathbf{b}_{\mathrm{U}}^{\mathrm{M}} \subseteq \mathbf{b}_{\mathrm{I}}$, thus loop invariant 1 holds. Since we know that $\forall x \in \mathbf{b}_{\mathrm{I}} \setminus \mathbf{b}_{\mathrm{U}}.\neg C_I(x)$ from our assumption that the loop invariants from the previous iteration hold, $\forall x \in \mathbf{b} \setminus \mathbf{b}_{\mathrm{P}}.\neg C(x)$ from Lemma 3.6.2, and from our assumption that C has the same truth value as C_I over $\mathbf{b} \supseteq \mathbf{b}_{\mathrm{P}}$, and $\mathbf{b} \subseteq \mathbf{b}_{\mathrm{U}}$, it must be true that $\forall x \in \mathbf{b}_{\mathrm{I}} \setminus \mathbf{b}_{\mathrm{U}}^{\mathrm{M}}. \neg C_I(x)$. With the fact that C_F has the same truth value as C and C_I with respect to $\mathbf{b}_{\mathrm{P}}$, $\mathbf{b}_{\mathrm{P}}^{\mathrm{L}} \subseteq \mathbf{b}_{\mathrm{P}} \wedge \mathbf{b}_{\mathrm{P}}^{\mathrm{R}} \subseteq \mathbf{b}_{\mathrm{P}}$, and the fact that all loop invariants were true for the previous iteration of the loop, the final loop invariant is trivial.

Now that the loop invariants have been proven to hold, we continue with proving soundness of Algorithm 5. If the output is "C_I satisfiability undecided, gave up at box $\mathbf{b}_{\mathrm{P}} \subseteq \mathbf{b}_{\mathrm{I}}$", then we are in an iteration of the loop where the d picked from L is bigger than $dMax$ and prune could not decide the satisfiability of C over $\mathbf{b}$. With loop invariant 1, we know that $\mathbf{b} \subseteq \mathbf{b}_{\mathrm{I}}$. Prune guarantees that $\mathbf{b}_{\mathrm{P}} \subseteq \mathbf{b}$ (Lemma 3.6.2), so it must be true that $\mathbf{b}_{\mathrm{P}} \subseteq \mathbf{b}_{\mathrm{I}}$.

If the output is "C_I is satisfied over $\mathbf{m} \subseteq \mathbf{b}_{\mathrm{I}}$", then we are in an iteration of the loop where prune has decided that C is satisfiable for all values in $\mathbf{m}$ and $\mathbf{m} \subseteq \mathbf{b}$ (Lemma 3.6.2). Since loop invariant 1 holds, we know that $\mathbf{b} \subseteq \mathbf{b}_{\mathrm{I}}$. Clearly, $\mathbf{m} \subseteq \mathbf{b}_{\mathrm{I}}$. Let x be an arbitrary value in $\mathbf{m}$. From Lemma 3.6.2, $C_I(x)$ must be true.

Finally, If the output is "C_I is unsatisfiable over $\mathbf{b}_{\mathrm{I}}$", then we have exited the loop. Since we have exited the loop, L must be empty. The union of all boxes in L is clearly empty. Let x be an arbitrary value in $\mathbf{b}_{\mathrm{I}}$. Since the union of all boxes in L is empty, from loop invariant 2, we have the following fact: $\forall x \in \mathbf{b}_{\mathrm{I}} \setminus \emptyset. \neg C_I(x)$. $C_I(x)$ must be false.

Thus, Algorithm 5 is sound.

$\square$

3.8 Searching for a Model using Linearisations

We now describe how a conjunction can be strengthened via linearisations in order to create a system to find a model which satisfies said conjunction. We first show a system with two variables, then show a system with an arbitrary amount of variables, and finally describe how we call the simplex method.

3.8.1 System with Two Variables

Let C be a conjunction of differentiable `EConstraint` terms and $\mathbf{b}$ be a box as shown in (3.5). We use this box to create $\mathbf{b}'$, a box that has been transformed such that the lower bound of each variable is 0, as shown in (3.6). Let $x'_R := x_R - x_L$ and $y'_R := y_R - y_L$. The constraints on the variables are exactly the same as those shown in (3.7).

(3.18) and (3.19) show how we linearise each term in C from the 'extreme' left and 'extreme' right corners of $\mathbf{b}'$ respectively. In these equations, $\mathbf{l_t}$, $\mathbf{r_t}$, and $\mathbf{J_t}$ are defined in (3.9). We have two versions because it is not uncommon for these linearisations to not be able to find a model from one 'extreme' corner but be able to find a model from the opposite 'extreme' corner.

$$0 \leq \underline{\mathbf{l_t}} + \underline{\mathbf{J_t}} \cdot \begin{bmatrix} \mathrm{x}' - 0 \\ \mathrm{y}' - 0 \end{bmatrix} \tag{3.18}$$

$$0 \leq \underline{\mathbf{r_t}} - \overline{\mathbf{J_t}} \cdot \begin{bmatrix} x'_R - \mathrm{x}' \\ y'_R - \mathrm{y}' \end{bmatrix} \tag{3.19}$$

In (3.18), the first inequality binds t from the 'extreme' left corner of $\mathbf{b}'$ where $x = y = 0$. The right-hand side of the first inequality is a linearisation of t. Since we are looking for a model, this linearisation must be a *strengthening* of $t \geq 0$. Starting from the left corner of $\mathbf{b}'$, the actual value of $[\![t]\!](\mathbf{b_L})$ is $\in \mathbf{l_t}$. We strengthen this by starting the linearisation at $\underline{\mathbf{l_t}}$. As we move away from the left corner, we multiply the point where we are at with the *lower* bound of the partial derivative for each variable and add the result to $\underline{\mathbf{l_t}}$. The resulting linearisation *strengthens* $t \geq 0$. One may visualise this as guaranteeing that t is 'above' or equal to the linearisation.

Similarly, (3.19) strengthens $t \geq 0$ from the extreme right corner of $\mathbf{b}'$. Here, $[\![t]\!](\mathbf{b_R})$ is $\in \mathbf{r_t}$ and we strengthen this by starting the linearisation at $\underline{\mathbf{r_t}}$. As we move away from the right corner, we multiply the point where we are at with the *upper* bound of the *negated* partial derivatives for each variable. As the upper bound is negated, we are still *strengthening* $t \geq 0$, guaranteeing that t is 'above' or equal to the linearisation. Note that the two-phase simplex method only supports non-strict inequalities, linearising $t > 0$ with the above systems will cause a weakening due to the loss of strictness of the inequality.

A complete system of linear inequalities is made by combing the reformulation of the box shown in (3.7) with either one of the linearisations (3.18) and (3.19) for each term in the conjunction. In LPPaver, the corner chosen for the linearisation is alternated each time the linearisation is performed over a given box and conjunction.

3.8.2 System with an Arbitrary Number of Variables

As explained previously, it is easy to extend (or shrink) the system described above. As before, we simplify the presentation of this system by using $\mathtt{x}_1$, $\mathtt{x}_2$, etc. instead of $\mathtt{x}$, $\mathtt{y}$, etc. and x_{1L}, x_{2L}, etc. instead of x_L, y_L, etc.

Let C be a conjunction of differentiable `EConstraint` terms and $\mathbf{b}$ be a box with an arbitrary amount of variables as shown in (3.11). We transform $\mathbf{b}$ into $\mathbf{b}'$ where the lower bound of the domain of each variable is 0 as shown in (3.12).

$$0 \leq \underline{\mathbf{l_t}} + \underline{\mathbf{J_t}} \cdot \begin{bmatrix} \mathtt{x}'_1 - 0 \\ \mathtt{x}'_2 - 0 \\ \vdots \\ \mathtt{x}'_n - 0 \end{bmatrix} \tag{3.20}$$

$$0 \leq \underline{\mathbf{r_t}} - \overline{\mathbf{J_t}} \cdot \begin{bmatrix} x'_{1R} - \mathtt{x}'_1 \\ x'_{2R} - \mathtt{x}'_2 \\ \vdots \\ x'_{nR} - \mathtt{x}'_n \end{bmatrix} \tag{3.21}$$

We now linearise both extreme corners of the box in a similar manner as shown in (3.18) and (3.19). Let $x'_{1R} := x_R - x_L$ and similarly for x'_{2R}, x'_{nR}, etc. The linearisation of a conjunction and box with an arbitrary number of variables from the left and right 'extreme' corners are shown in (3.20) and (3.21) respectively. These linearisations *strengthen* $t \geq 0$.

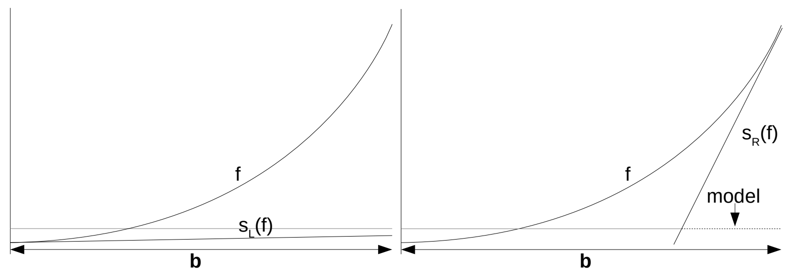


Figure 3.2: A linearisation that strengthens a term whose function graph is f over the 1-dimensional box $\mathbf{b}$. The lines labelled $\mathsf{s}_{\mathsf{L}}(\mathrm{f})$ and $\mathsf{s}_{\mathsf{R}}(\mathrm{f})$ are the linearisations made from the left and right 'extreme' corners of $\mathbf{b}$, respectively. Only the $\mathsf{s}_{\mathsf{R}}(\mathrm{f})$ linearisation would succeed in finding a model over the box and the set of models that can be found with this linearisation is represented by the dotted line.

The complete system of linear inequalities is constructed by combining the reformulation of the box shown in (3.7) with either one of (3.20) and (3.21). A 1-dimensional example of these linearisations for both the left and right 'extreme' corners is given in Figure 3.8.2

3.8.3 Calling the Simplex Method

Let s be a system of linear inequalities as described above. Since the goal of this system is to find a model of some $C : [\texttt{EConstraint}]$ with respect to some box $\mathbf{b}$, and the system is (mostly) a strengthening of C, we only need to perform the first phase of the two-phase simplex method.

If the first phase determines that the system is infeasible, we return the fact that we could not find a model. If the first phase gives a feasible result, we have a *feasible point* which is a *potential* model for C. As explained previously, it is a potential model due to the weakening of the strictness of the inequalities in C. The potential model is stored as a box named $\mathbf{m}'$

Since we transformed $\mathbf{b}$ when creating the system, we must 'undo' this by transforming $\mathbf{m}'$ appropriately. We transform $\mathbf{m}'$ by adding $x_{1L}, x_{2L}, \ldots, x_{nL}$ from (3.11) to each variable $x_1, x_2, \ldots, x_n$ in $\mathbf{m}$. The resulting box is called $\mathbf{m}$ and is a *potential* model for C within $\mathbf{b}$.

3.8.4 Soundness

To prove that the linearisations described in this section soundly strengthens a conjunction of non-strict differentiable terms over some box, we must first prove that the linearisations soundly strengthen a non-strict differentiable term over some box.

Lemma 3.8.1 (Soundness of using linearisations)**.** *For every differentiable term within a* non-strict `EConstraint` t*, and for every box* $\mathbf{b}$*, let* $e_{(S,1)}$ *and* $e_{(S,2)}$ *be the* `EConstraint` *equivalent of the linearisation of* t *over* $\mathbf{b}$ *using the (3.20) and (3.21) linearisations as described in Section 3.8.2, respectively. The following statements hold:*

1. $\forall x \in \mathbf{b}.(e_{(S,1)}(x) \implies t(x) \geq 0)$
2. $\forall x \in \mathbf{b}.(e_{(S,2)}(x) \implies t(x) \geq 0)$

Proof outline. From Section 3.8.2, we know that both $e_{(S,1)}$ and $e_{(S,2)}$ is a strengthening of $t \geq 0$ over $\mathbf{b}$. Note that we do *not* consider $t > 0$ since the linear system only supports nonlinear inequalities, $t > 0$ cannot be represented in the system without at least weakening the statement to $t \geq 0$. Let x be an arbitrary value from $\mathbf{b}$. $(e_{(S,1)}(x) \implies t(x) \geq 0)$ and $(e_{(S,2)}(x) \implies t(x) \geq 0)$ $\square$

Now that we know that the system of inequalities from Section 3.8.2 soundly strengthens a *non-strict* differentiable term over some box, we discuss how the same linearisations can be used to create a system of linear inequalities which represents a strengthening of a conjunction of non-strict differentiable terms over some box.

Corollary 3.8.1.1 (Soundness of using linearisations which strengthen a conjunction of `EConstraint`s)**.** *For every conjunction* C *:* [`EConstraint`] *consisting of* non-strict *differentiable terms, and for every box* $\mathbf{b}$*, let* s_1 *and* s_2 *be the system of linear inequalities produced by linearising every term in* C *using the (3.20) and (3.21) linearisations as described in Section 3.8.2, respectively. Let* $C_{(S,1)}$ *and* $C_{(S,2)}$ *be the* [`EConstraint`] *equivalent to* s_1 *and* s_2*, respectively. The following statements hold:*

1. $\forall x \in \mathbf{b}.C_{(S,1)}(x) \implies C(x)$
2. $\forall x \in \mathbf{b}.C_{(S,2)}(x) \implies C(x)$

Proof outline. We consider the first statement. Since each term in C is differentiable and non-strict, $C_{(S,1)}$ is a version of C where every term has been soundly strengthened over $\mathbf{b}$ as proven in Lemma 3.8.1. Let x be an arbitrary value from $\mathbf{b}$. Since $C_{(S,1)}$ is a strengthening of C over $\mathbf{b}$, $C_{(S,1)}(x) \implies C(x)$. The case for the second statement is similar. □

Now that we know that the Section 3.8.2 produces a system of linear inequalities which soundly strengthens a conjunction of non-strict differentiable terms over some box, we discuss how the optimisations we perform over this system can soundly find a model within the box which satisfies the conjunction if the system is feasible.

Lemma 3.8.2 (Soundness of searching for a model for some conjunction within a box using linearisations)**.** *For every conjunction C :* $[\texttt{EConstraint}]$ *consisting of* non-strict *differentiable terms, and for every box* $\mathbf{b}$*, let* $\mathbf{m}$ *be the box resulting from optimising over the linearisation of* C *as described in Sections 3.8.3. The following statement holds:*

$$\begin{gathered} \mathbf{m} \subseteq \mathbf{b} \\ \forall x \in \mathbf{m}.C(x) \end{gathered} \tag{3.22}$$

Proof outline. Let C_S be either strengthening of C over $\mathbf{b}$ from Lemma 3.8.1.1. The exact strengthening is not relevant for this proof outline, the justification is the same for both. Combine the system with the linearisation of $\mathbf{b}$ shown in (3.12). The resulting system is optimised as described in Section 3.8.3, producing the box $\mathbf{m}$.

Since the system was created using $\mathbf{b}$, the bounds for the optimised variables must be within $\mathbf{b}$, so $\mathbf{m} \subseteq \mathbf{b}$.

If the system is infeasible, $\mathbf{m} :=$ and $\forall x \in \mathbf{m}.C(x)$ is vacuously true. If a feasible system is optimised, we will have a new box, $\mathbf{m}$, such that $\forall x \in \mathbf{m}.C_S(x)$. Let x be an arbitrary value in $\mathbf{m}$. Since $\forall x \in \mathbf{b}.C_S(x) \implies$

$C(x)$ and $C_S(x)$ is true, $C(x)$ must also be true. Thus, the linearisations and optimisations described in Sections 3.8 and 3.8.3, respectively, will soundly find a model in a box which satisfies some conjunction consisting of *non-strict* differentiable terms. □

3.9 Pruning and Searching for Models via Interval Methods and Linearisations

In this section, we present the 'PruneAndSearch' algorithm which is identical to the 'prune' algorithm up to the contraction step of the given box. If the contracted box is not empty, and all non-true terms in the conjunction we are checking are differentiable, we look for a model for the given conjunction in the contracted box using the linearisations and optimisations described in Section 3.8.

If a model is found, it is first 'verified' by testing if the given conjunction interval evaluates to true over the given model. If the model is verified, we return the contracted box, the filtered conjunction, *and* the model. This 'model verification' step is necessary as the linearisations used to find a model do not distinguish between strict and non-strict inequalities. The verification of the model using interval methods often avoids 'touching' cases as the model is typically a point where the conjunction is very clearly true. If the 'model verification' step returns false, then the given model was either incorrect (due to the weakening of the strictness of the inequalities in the conjunction) or was indeterminate due to 'touching'. In either case, the rest of the algorithm is (more-or-less) identical to Algorithm 4.

3.9.1 Termination

Lemma 3.9.1 (Termination of PruneAndSearch)**.** *For any box $\mathrm{b_I}$, for any conjunction of* `EConstraint`*s C_I, Algorithm 6 terminates.*

Proof outline. The proof outline for this is the same as the proof outline for Lemma 3.6.1. □

Algorithm 6 PruneAndSearch: contract a box and search for a model using interval methods and linearisations

Input: ($\mathbf{b}_\mathrm{I}$: box, C_I : [EConstraint])
Output: a pruned box $\mathbf{b}_\mathrm{P}$, a filtered conjunction C_F, and maybe a model $\mathbf{m}$

```
C_F := C_I without terms that interval evaluate to true over b_I
C_W := weaken C_F by transforming f > 0 into f ≥ 0
if C_F is empty then
   m := b_I
   return (b_I, true, m)                # An empty conjunction implies C_I holds over b_I
else if any term in C_F is false for all values in b_I then
   return (∅, C_F, ∅) # An empty box implies at least one term in C_I was false for all values in b_I
end if
C_W^Δ := filter out non-differentiable terms from C_W
b_P := contract b_I using a linearisation of C_W^Δ described in Section 3.5
if b_P = ∅ then                         # This means that C_W is false over b_P
   return (∅, C_F, ∅)
else if all terms in C_W are differentiable then
   m := find a model in b_P using a linearisation of C_W described in Section 3.8
   if m ≠ ∅ and C_F interval evaluates to true over m then
      return (b_P, C_F, m)
   end if
else if |b_I|/|b_P| ≥ ε_R + 1 ∧ |b_I| − |b_P| ≥ ε_A then    # Has b_P reduced significantly?
   PruneAndSearch(b_P, C_F)             # Recursive step
else
   return (b_P, C_F, ∅)
end if
```

3.9.2 Soundness

Building on the soundness of the linearisations and optimisations discussed in Section 3.5 we now discuss soundness of PruneAndSearch (Algorithm 4).

Lemma 3.9.2 (Soundness of PruneAndSearch)**.** *For any box* $\mathbf{b}_{\mathrm{I}}$ *and for any conjunction of* `EConstraint`*s* C_I *the following statements about the output box* $\mathbf{b}_{\mathrm{P}}$*, the conjunction* C_F*, and model* $\mathbf{m}$ *from Algorithm 6, hold:*

1. $\mathbf{b}_{\mathrm{P}} \subseteq \mathbf{b}_{\mathrm{I}}$
2. $\mathbf{m} \subseteq \mathbf{b}_{\mathrm{P}}$
3. $\forall x \in \mathbf{b}_{\mathrm{P}} C_I(x) \iff C_F(x)$
4. $\forall x \in \mathbf{b}_{\mathrm{I}}.C_I(x) \implies x \in \mathbf{b}_{\mathrm{P}}$
5. $\forall x \in \mathbf{b}_{\mathrm{I}} \setminus \mathbf{b}_{\mathrm{P}}.\neg C_F(x)$
6. $\forall x \in \mathbf{m}.C_F(x)$

Proof outline. Let $\mathbf{b}_{\mathrm{I}}$ be an arbitrary box. Let C_I be an arbitrary conjunction of `EConstraint`s.

The proof for statements 1, 3, 4, and 5 is similar to Lemma 3.6.2 as the 'prune' part of the Algorithm 6 is more-or-less the same as Algorithm 4. Where Algorithm 6 differs is when it tries to find a model.

Recall that $C_{F,1}$ is C_I without terms that interval evaluate to true over $\mathbf{b}_{\mathrm{I}}$, C_W is a weakening of $C_{F,1}$ where all inequalities are non-strict, C_W^{Δ} is a weakening of C_W where all non-differentiable terms have been removed, and $\mathbf{b}_{\mathrm{P}}$ is a box resulting from an attempted contraction of $\mathbf{b}_{\mathrm{I}}$ using a linearisation of C_W^{Δ} as described in Section 3.5. From Lemma 3.6.2, we know that $\mathbf{b}_{\mathrm{P}} \subseteq \mathbf{b}_{\mathrm{I}}$, $\forall x \in \mathbf{b}_{\mathrm{I}}.C_I(x) \iff C_F(x)$, $\forall x \in \mathbf{b}_{\mathrm{I}}.C_I(x) \implies x \in \mathbf{b}_{\mathrm{P}}$, and $\forall x \in \mathbf{b}_{\mathrm{I}} \setminus \mathbf{b}_{\mathrm{P}}.\neg C_F(x)$.

Now, if all terms in C_W are differentiable, let $\mathbf{m}$ be the box representing a model for C_W from Lemma 3.8.2. If $\mathbf{m}$ is empty, statement 2 is trivial and statement 6 is vacuously true.

If $\mathbf{m}$ is not empty, from Lemma 3.8.2, we know that $\mathbf{m} \subseteq \mathbf{b}_{\mathrm{P}}$ and $\mathbf{m}$ satisfies C_W. Let x be an arbitrary value in $\mathbf{m}$. We know that $C_W(x)$ is true. Since C_W is a weakening of $C_{F,1}$ where all strict inequalities become non-strict, it may be true that $C_{F,1}(x)$ is false due to this weakening, so we use interval evaluations to verify if $C_{F,1}$ is satisfied by $\mathbf{m}$. If so, $C_{F,1}(x)$ is clearly true.

If interval evaluations show that $C_{F,1}$ is not satisfied by $\mathbf{m}$, we ignore the model given by the optimisations, i.e., let $\mathbf{m} := \emptyset$. Now statement 2 is trivial and statement 6 is vacuously true.

Since Lemma 3.6.2 shows that the first statements 1, 3, 4, and 5 hold, and we have shown that statements 2 and 6 regarding $\mathbf{m}$ hold, Algorithm 3.9.2 is sound. $\square$

3.10 Finding Models via Best-First Searching and Pruning

The algorithm described in Section 3.6 works well when given a constraint that is unsatisfiable over the given box. Due to the nature of a depth-first search, the algorithm may struggle to find a solution if the model is quite close to 0. This problem is exacerbated due to 'touching' which interval methods are known to struggle with. When branching in a depth-first algorithm, if the algorithm is examining a box that is an actual model but very close to the 'boundary' (which is always 0 in LPPaver), interval methods would not be able to verify this as a model due to 'touching' because of overapproximations made when computing intervals. The algorithm would bisect the box. The bisected boxes would be processed and the model may still not be verifiable with interval methods. The bisecting and checking repeats until we get a box that is small enough to verify with interval methods or we reach the termination condition.

To remedy this, we introduce another algorithm better suited to finding models. Algorithm 8 shows a 'best-first' branch-and-prune algorithm. The algorithm is very similar to Algorithm 5, but L is a priority queue instead of a stack, hence the name 'best-first'. We sort the priority queue using a heuristic with the goal of placing boxes and conjunctions that are more likely to produce models at the front of the queue. This heuristic is defined in Algorithm 7. The algorithm also depends on Algorithm 6 rather than Algorithm 4. In the best-first algorithm, the termination predicate depends on the *number of boxes processed* rather than the *depth* of the box being examined.

Algorithm 7 priority: calculate a priority number for some box and conjunction of `EConstraint`s, a higher priority value should be prioritised over lower values.

Input: ($\mathbf{b}_\mathrm{I}$: box, C_I : [EConstraint])
Output: a number representing the priority for C_I with $\mathbf{b}_\mathrm{I}$

```
ranges := compute interval ranges of each term in C_I over b_I
average := compute interval average of ranges
return centre of average
```

3.10.1 Termination

Remark 1 (Termination of the Priority Algorithm)**.** *For any box* $\mathbf{b}_\mathrm{I}$*, for any conjunction of* `EConstraint`*s* C_I*, Algorithm 7 terminates.*

Lemma 3.10.1 (Termination of the Best-First Proving Algorithm)**.** *For any box* $\mathbf{b}_\mathrm{I}$*, for any conjunction of* `EConstraint`*s* C_I*, for any* $bMax \in \mathbb{N}$*, Algorithm 8 terminates.*

Proof outline. Let $\mathbf{b}_\mathrm{I}$ be an arbitrary box. Let C_I be an arbitrary conjunction of `EConstraint`s. Let $bMax$ be an arbitrary natural number.

The algorithm initialises a variable i with the natural number 0. The algorithm then loops. Within the loop, the algorithm calls Algorithms 6 and 7 which both terminate according to Lemma 3.6.1 and Remark 1, respectively. The loop terminates when either something is returned or $i \geq bMax$. At the end of the loop, i is incremented by 1. Since $bMax$ is finite, after $bMax$ iterations, $i = bMax$. Clearly, $i \geq bMax$, so the number of iterations of the loop before the algorithm *must* terminate is $bMax$. $\square$

Algorithm 8 Searching for a model with best-first branching + pruning

Input: ($\mathbf{b}_\mathrm{I}$: box, C_I : [EConstraint], $bMax$: $\mathbb{N}$)

Output: satisfiability of C_I over $\mathbf{b}_\mathrm{I}$, model $\mathbf{m} \subseteq \mathbf{b}_\mathrm{I}$ if C_I is satisfiable

initialise priority queue L with (priority($\mathbf{b}_\mathrm{I}$,C_I), $\mathbf{b}_\mathrm{I}$, C_I)
i = 0 # This variable tracks the number of boxes that have been processed
while $L \neq \emptyset$ **do**
    ($\mathbf{b}$, C) := pick(L) # retrieve the box and conjunction with the highest priority from L
    ($\mathbf{b}_\mathrm{P}$, C_F, $\mathbf{m}$) := PruneAndSearch($\mathbf{b}$, C) # $\mathbf{m}$ is a box that stores a model for C_I (if found)
    if $\mathbf{b}_\mathrm{P} \neq \emptyset$ **then**
        if C_F is trivially true **then** # If C_F is true, $\mathbf{b}_\mathrm{P}$ satisfies C_I
            return C_I is satisfied over $\mathbf{m} \subseteq \mathbf{b}_\mathrm{I}$ # Note that here, $\mathbf{m} = \mathbf{b}_\mathrm{P}$
        else if $\mathbf{m} \neq \emptyset$ **then**
            return C_I is satisfied over $\mathbf{m} \subseteq \mathbf{b}_\mathrm{I}$
        else if $i \geq t$ **then** # The termination condition depends on the *number of boxes processed*
            return satisfiability of C_I undecided, gave up at box $\mathbf{b}_\mathrm{P}$
        else
            ($\mathbf{b}_\mathrm{P}^\mathrm{L}$, $\mathbf{b}_\mathrm{P}^\mathrm{R}$) := split($\mathbf{b}_\mathrm{P}$) # Bisect the variable with the largest width
            add (priority($\mathbf{b}_\mathrm{P}^\mathrm{L}$,$C_I$), $\mathbf{b}_\mathrm{P}^\mathrm{L}$, C_F) & (priority($\mathbf{b}_\mathrm{P}^\mathrm{R}$,$C_I$), $\mathbf{b}_\mathrm{P}^\mathrm{R}$, C_F) to L
        end if
    end if
    i = i + 1
end while
return C_I is unsatisfiable over $\mathbf{b}_\mathrm{I}$

3.10.2 Soundness

Having established that Algorithm 6 is sound and Algorithm 8 terminates, we now show the soundness of Algorithm 8.

Lemma 3.10.2 (Soundness of the Best-First Proving)**.** *For any box $\mathbf{b}_\mathrm{I}$, for any conjunction of* `EConstraint`*s C_I, for any $bMax \in \mathbb{N}$, the following statements regarding the output of Algorithm 8 hold:*

- *If the output is "C_I is satisfied over $\mathbf{m} \subseteq \mathbf{b}_\mathrm{I}$" then $\forall x \in \mathbf{m}.C_I(x)$ and $\mathbf{m} \subseteq \mathbf{b}_\mathrm{I}$*
- *If the output is "C_I satisfiability undecided, gave up at box $\mathbf{b}_\mathrm{P} \subseteq \mathbf{b}_\mathrm{I}$" then $\mathbf{b}_\mathrm{P} \subseteq \mathbf{b}_\mathrm{I}$*
- *If the output is "C_I is unsatisfiable over $\mathbf{b}_\mathrm{I}$" then $\forall x \in \mathbf{b}_\mathrm{I}.\neg C_I(x)$*

Proof outline. The proof for every statement apart from 'If the output is "C_I is satisfied over $\mathbf{m} \subseteq \mathbf{b}_\mathrm{I}$" then $\forall x \in \mathbf{b}_\mathrm{P}.C_I(x)$ and $\mathbf{b}_\mathrm{P} \subseteq \mathbf{b}_\mathrm{I}$' is similar to Lemma 3.7.2, with the only major difference being a priority queue being used rather than a stack.

For the remaining statement, let $\mathbf{b}$ and C be the box and `[EConstraint]` picked from the priority queue, respectively. Since the priority queue in this algorithm is populated in a very similar way to the stack in Algorithm 5, we know from Lemma 3.7.2 that $\mathbf{b} \subseteq \mathbf{b}_\mathrm{I}$ and $\forall x \in \mathbf{b}.C(x) \iff C_I(x)$. PruneAndSearch is called with arguments $\mathbf{b}$ and C, producing the triple ($\mathbf{b}_\mathrm{P}$, C_F, $\mathbf{m}$). From Lemma 3.9.2, we have the following:

1. $\mathbf{b}_\mathrm{P} \subseteq \mathbf{b}$
2. $\mathbf{m} \subseteq \mathbf{b}_\mathrm{P}$
3. $\forall x \in \mathbf{b}.C(x) \implies x \in \mathbf{b}_\mathrm{P}$
4. $\forall x \in \mathbf{b}_\mathrm{P}.C(x) \iff C_F(x)$
5. $\forall y \in \mathbf{b} \setminus \mathbf{b}_\mathrm{P}.\neg C_F(y)$
6. $\forall x \in \mathbf{m}.C_F(x)$

Clearly, $\mathbf{m} \subseteq \mathbf{b}_{\mathrm{I}}$. Let x be an arbitrary value from $\mathbf{m}$.

If C_F is trivially true, the algorithm returns "C_I is satisfied over $\mathbf{m} \subseteq \mathbf{b}_{\mathrm{I}}$". Since $\forall x \in \mathbf{b}_{\mathrm{P}}.C(x) \iff C_F(x)$ and $\mathbf{m} \subseteq \mathbf{b}_{\mathrm{P}}$, $C(x)$ is true. Since $\forall x \in \mathbf{b}.C(x) \iff C_I(x)$, and $\mathbf{b}_{\mathrm{P}} \subseteq \mathbf{b}$, $C_I(x)$ is true. Thus, this branch is sound

If C_F is not trivially true and $\mathbf{m}$ is non-empty, the algorithm returns "C_I is satisfied over $\mathbf{m} \subseteq \mathbf{b}_{\mathrm{I}}$". From $\forall x \in \mathbf{m}.C_F(x)$, we know that $C_F(x)$ is true. The remaining proof outline for this branch is the same as above.

All other branches are similar to the branches in Algorithm 5 and have been shown to be sound in Lemma 3.7.2. Thus, Algorithm 8 is sound. □

Chapter 4

PropaFP

Consider the approximation of the sine function shown in Listing 4.1 which we previously discussed in Chapter 2. The current state-of-the-art in automated formal verification is unable to verify functional specifications like those shown in (4.1).

$$X \in [-0.5, 0.5] \implies |\texttt{Taylor\_Sin'Result} - \sin(X)| \leq 0.00025889 \quad (4.1)$$

We would like a tool to automatically verify this specification or obtain a counter-example if it is not valid.

Problem. As mentioned in Chapter 2, with a SPARK version of the specification (4.1), the SPARK toolchain automatically verifies absence of overflow in the `Taylor_Sin` function which is not difficult since the input `X` is restricted to the small domain $[-0.5, 0.5]$.

We would like to be able to verify that the result of `Taylor_Sin(X)` is close to the exact $\sin(\texttt{X})$. We would also like this verification step to be done automatically, that is, with a specification for `Taylor_Sin` which specifies

Listing 4.1: Approximation of the sine function in Ada

```
function Taylor_Sin (X : Float) return Float is
  (X - ((X * X * X) / 6.0));
```

restrictions on the input and behaviour regarding the output, we would like a user to be able to call a process which will, without any interaction required from the user, attempt to decide whether or not the specification is correct. If the specification is incorrect, we would like the user to receive a counter-example.

There are some other processes in literature that aim to achieve a similar goal. For example, in [8], the authors make use of Gappa [31] and various SMT solvers in order to (attempt to) decide specifications such as the one we describe. In some cases, particularly when trigonometric operations are present, their process requires some manual steps such as adding lemmas to the program in order to aid solvers.

The current SPARK toolchain and other frameworks we know of are unable to verify that the result of `Taylor_Sin(X)` is close to the exact $\sin(\mathtt{X})$ with the only interaction from the user required being to call some prover and with only the specification present in (4.1). We suspect that this is because it is typically difficult to reason about FP operations, particularly when combined with non-linear real functions, due to the difficulty of soundly considering the possible rounding errors as well as their consequences.

Solution. To automatically verify functional specifications analogous to the one in equation (4.1), we have designed and implemented PropaFP, an extension of the SPARK proving process. The following steps are applied to quantifier-free VCs that contain real inequalities:

1. Derive bounds for variables and simplify the VC.
2. Safely replace FP operations with the corresponding exact operations.
3. Again simplify the VC.
4. Attempt to decide the resulting VCs with provers for nonlinear real theorems.

PolyPaver [25] is a nonlinear real theorem prover that integrates with an earlier version of SPARK in a similar way, but lacks the simplification steps and has a much less powerful method of replacing FP operations.

Listing 4.2: SPARK formal specification of `Taylor_Sin`

```
function Taylor_Sin (X : Float) return Float with
  Pre => X >= -0.5 and X <= 0.5,
  Post =>
    abs(Real_Sin(Rf(X)) - Rf(Taylor_Sin'Result))
                          <= Ri(25889) / Ri(100000000);
                          -- 0.00025889
```

4.1 Our Proving Process Steps

We will illustrate the steps using the program `Taylor_Sin` from Listing 4.1. Let us first consider its SPARK formal specification shown in Listing 4.2.

To write more intuitive specifications, we use the Ada `Big_Real` and `Big_Integer` libraries to get exact rational arithmetic in specifications. Although in Ada the type `Big_Real` contains only rationals, Why3 treats `Big_Real` as the type of reals. We added non-rational functions such as `Real_Sin` as ghost functions: functions with no implementation, only a specification. Their specifications give a collection of basic axioms for solvers that do not understand the function natively. For example, the specification of `Real_Sin` declares the range of sine and its values at selected points.

The listings in this thesis use shortened versions of some functions to aid readability. Functions `FC.To_Big_Real`, `FLC.To_Big_Real`, and `To_Real` respectively embed `Floats`, `Long_Floats` (doubles), and `Integers` to the `Big_Reals` type. We have shortened these to `Rf`, `Rlf`, and `Ri`, respectively. The post-condition specifies a bound on the total error, i.e., the difference between this Taylor series approximation of sine and the exact sine of `X`.

4.1.1 Generating and processing verification conditions

We use GNATprove/Why3 to generate VCs. In principle, we could use other programming and specification languages, as long as we can obtain VCs of a similar nature.

If a VC is not decided by the included SMT solvers, we use the Manual Proof feature in GNAT Studio to invoke PropaFP via a custom Why3 driver

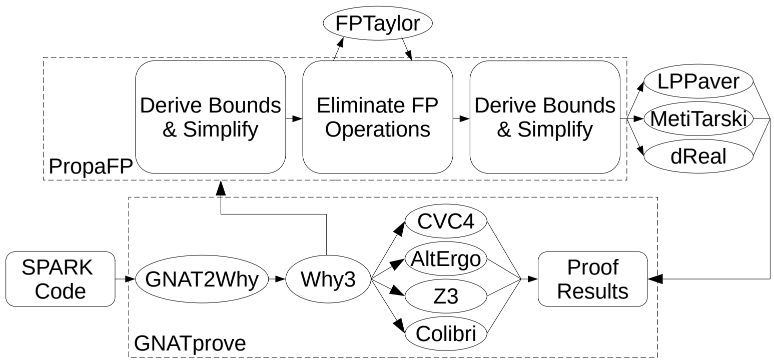


Figure 4.1: Overview of Automated Verification via GNATprove with PropaFP

based on the driver for CVC4. This driver applies selected Why3 transformations and saves the VC in SMT format. Since in this format the VC is a negation of the specification from which it was produced, we shall refer to it as ‘the negated VC’ (NVC). The VC $\texttt{contextAsConjunction} \implies \texttt{goal}$ becomes the NVC $\texttt{contextAsConjunction} \wedge \neg\texttt{goal}$. During further processing, we may weaken this conjunction of assertions by, for example, dropping assertions. A model that satisfies the weakened NVC will not necessarily be a counter-example to the original VC or the original specification. However, if the weakened NVC has no model, then both the original VC and the original specification are correct.

When parsing the SMT files, we ignore the definitions of basic arithmetic operations and transcendental functions. Instead of using these definitions, we use each prover’s built-in interpretations of such operations and functions. In more detail, the parsing stage comprises the following steps:

- Parse the SMT file as a list of Lisp S-expressions. Drop everything except assertions and variable and function type declarations.
- Scan the assertions and drop any that contain unsupported functions.
- Functions are interpreted using their names.
  - We rely on each prover’s built-in understanding of the supported functions, currently $-, +, \times, \div, \sin, \cos, \sqrt{\cdot}, \mathrm{mod}, \mathrm{abs}, \min, \max$.

Listing 4.3: NVC corresponding to the post-condition from Listing 4.2

```
-- assertions regarding axioms for sin and pi omitted
assert to_float(RNA, 1) = 1.0
assert isFiniteFloat(x)
assert (-0.5) ≤ x ∧ x ≤ 0.5
assert isFiniteFloat(x⊙x)
assert isFiniteFloat((x⊙x)⊙x)
assert isFiniteFloat(x ⊖ (((x⊙x)⊙x)⊘6.0))
assert
 ¬((
    sin(x) + (-1·(x ⊖ (((x⊙x)⊙x)⊘6.0))) ≥ 0.0
    ⟹
    sin(x) + (-1·(x ⊖ (((x⊙x)⊙x)⊘6.0))) ≤
                                             25889/100000000
  )∧(
    ¬(sin(x) + (-1·(x ⊖ (((x⊙x)⊙x)⊘6.0))) ≥ 0.0)
    ⟹
    -1⊙(sin(x) + (-1·(x ⊖ (((x⊙x)⊙x)⊘6.0)))) ≤
                                             25889/100000000
  ))
```

  - To increase the safety of this interpretation, we check the return type of some 'ambiguous' functions.
    - For example, the output of a function named `of_int` depends on the return type, i.e. If the return type of `of_int(x)` is a single-precision float, interpret this as `Float32(x)`.
    - For functions such as `fp.add`, the return type is clear from the name of the function. Also, for these functions a type declaration is normally not included in the SMT file.

- Determine the precision of FP operations by a bottom-up type derivation. The precision of literals is clear since they are given as bit vectors and the precision of variables is given in their declarations.

Dealing with π. Similar to `Real_Sin`, we have added a ghost parameterless function, `Real_Pi`, whose specification contains selected axioms for the exact

π. Why3 turns this into the function `real_pi` with no input. To help provers understand that this is the exact π, all calls to `real_pi` are substituted with π. For `Taylor_Sin`, the only VC that the SMT solvers included with GNAT Studio cannot solve is the post-condition VC. The NVC for this post-condition is in Listing 4.3. It has been reformatted for better readability by removing redundant brackets, using circled symbols for FP operations, and omitting some irrelevant statements regarding axioms for trigonometric functions like sine as well as π, e.g. $\sin \pi = 0$, $\cos \pi = 1$, etc. The predicate `isFiniteFloat(X)` is short for the inequalities `MinFloat <= X, X <= MaxFloat`.

4.1.2 Simplifications

As some of the tools used by PropaFP require bounds on all variables, we attempt to derive bounds from the assertions in the NVC. But first, we make the following symbolic simplifications to help derive better bounds:

- Reduce vacuous propositions and obvious tautologies, such as:
  - `(NOT` φ `OR true) AND (`φ `OR false)` $\longrightarrow \varphi$
  - φ `=` $\varphi \longrightarrow$ `true`
- Eliminate variables by substitution as follows:
  - Find variable-defining equations in the NVC, except circular definitions.
  - Pick a variable definition and make substitutions accordingly.
    - E.g., pick `i=i1+1`, and replace all occurrences of `i` with `i1+1`.
  - If the variable has multiple definitions, pick the shortest one.
    - E.g., if we have both `x=1` and `x=0+1`, all occurrences of `x` will be replaced with `1`, including `x=0+1` $\longrightarrow$ `1=0+1`.
- Perform simple arithmetic simplifications, such as:
  - φ `/ 1` $\longrightarrow \varphi$
  - `0 + 1` $\longrightarrow$ `1`

  - `MIN (e, e)` $\longrightarrow$ `e`.

- Repeat the above steps until no further simplification can be made.

These steps are performed automatically and are defined more rigorously in Algorithm 11, which depends on Algorithm 10, which depends on Algorithm 9. We first turn to Algorithm 9, which applies a series of symbolic rules to simplify some input of type `E`.

Algorithm 9 SimplifyE: simplify an expression with symbolic rules

---

Input: (t_I : `E`)
Output: (t_S : `E`)

E := t_I
E_S := E
E_S := in E_S, transform each $t/1$ into t # t can be any expression
E_S := in E_S, transform each $0/t$ into 0
E_S := in E_S, transform each $-1 \times t$ into $-t$
E_S := in E_S, transform each $0 \times t$ into 0 # also its commutative version
E_S := in E_S, transform each $1 \times t$ into t # also its commutative version
E_S := in E_S, transform each $0 + t$ into t # also its commutative version
E_S := in E_S, transform each $0 - t$ into t # also its commutative version
E_S := in E_S, transform each $t - t$ into 0
E_S := in E_S, transform each t^0 into 1
E_S := in E_S, transform each t^1 into t
E_S := in E_S, transform each $-1 \times -1 \times t$ into t # also its commutative version
E_S := in E_S, transform each -0 into 0
E_S := in E_S, transform each $-(-t)$ into t
E_S := in E_S, transform each $\sqrt{0}$ into 0
E_S := in E_S, transform each $\sqrt{1}$ into 1
E_S := in E_S, if we have $|t|$ and t is a literal, replace $|t|$ with the absolute value of t
E_S := in E_S, transform each $min(t, t)$ into t
E_S := in E_S, transform each $max(t, t)$ into x
if E = E_S **then** # If none of the simplification rules were applied, stop simplifying
  return E_S
else
  return SimplifyE(E_S) # Keep simplifying until simplification rules do not apply
end if

Let size_o be a function which returns the number of operations within any input of either type `E` or type `F`. Let $\mathrm{size_l}$ be a function which returns the number of logical operators (`And`, `Or`, `Impl`, and `Not`) within any input of type `F`. We use $\mathrm{size_o}$ and $\mathrm{size_l}$ in our proof outline for various lemmas that appear in this chapter.

Lemma 4.1.1 (Termination of SimplifyE)**.** *For any input t_I : `E`, Algorithm 9 terminates.*

Proof outline. The algorithm executes a number of simplification rules on E_S and terminates when none of these rules apply.

Let $t_{S,k}$ be the value of E_S after k successful applications of the rules in the algorithm. Successful application of any rule in the algorithm reduces the size of the expression being simplified by at least 1. Thus, if k rules were applied, we have $\mathrm{size_o}(t) > \mathrm{size_o}(t_{S,1}) > \cdots > \mathrm{size_o}(t_{S,k})$. This sequence cannot be infinite. Thus, the algorithm will, at some point, stop changing E_S and terminate. □

Remark 2. *For any input t_I : `E` and corresponding output t_S : `E` of Algorithm 9, if $t_I \neq t_S$, then $\mathrm{size_o}(t_I) > \mathrm{size_o}(t_S)$.*

We now discuss soundness of Algorithm 9. For this and future algorithms and lemmas, we define $vars$, a function which takes some formula or term as input and returns the set of variables within the formula, e.g., $vars(x-1) = x$, $vars(x > y \wedge x < 0) = x, y$.

Lemma 4.1.2 (Soundness of SimplifyE)**.** *For any input t_I : `E` and corresponding output t_S : `E` of Algorithm 9, the following statement holds:*

$$\forall x \in vars(t_I).t_I(x) = t_S(x) \tag{4.2}$$

Proof outline. The algorithm starts by defining $E_S := t_I$. The algorithm then applies a number of simplification rules on E_S. All rules in SimplifyE clearly preserve the value of the term being simplified. No matter how many times it recurses, when the algorithm stops, we still have: $\forall x \in vars(t_I).t_I(x) \iff E_S(x)$. Therefore, this holds for the output value t_S □

We now define Algorithm 10 which simplifies any input of type `F`. Algorithm 10 relies on Algorithm 9 to simplify any t : `E` within the formula.

Lemma 4.1.3 (Termination of SimplifyF)**.** *For any input φ :* `F`*, Algorithm 10 terminates.*

Proof outline. The algorithm executes a number of simplification rules on φ and terminates when none of these rules apply. Note that $\mathrm{size_o}$ treats implications as equivalent disjunctions, i.e., $\mathrm{size_o}(\varphi_1 \implies \varphi_2) = \mathrm{size_o}(\neg\varphi_1 \vee \varphi_2)$.

Let $\varphi_{S,k}$ be the value of F_S after k successful applications of the rules in the algorithm. Successful application of any rule in the algorithm reduces the size of the expression being simplified by at least 1. Thus, we have $\mathrm{size_o}(\varphi_I) > \mathrm{size_o}(\varphi_{S,1}) > \cdots > \mathrm{size_o}(\varphi_{S,k})$. This sequence cannot be infinite. Thus, the algorithm will, at some point, stop changing F_S and terminate. □

Remark 3. *For any input φ :* `F` *and corresponding output φ_S :* `F` *of Algorithm 10, if $\varphi \neq \varphi_S$, then $\mathrm{size_o}(\varphi) > \mathrm{size_o}(\varphi_S)$.*

Building on the discussion of termination and soundness of Algorithm 9, and the termination of Algorithm 10, we now discuss soundness of Algorithm 10.

Lemma 4.1.4 (Soundness of SimplifyF)**.** *For any input φ :* `F` *and corresponding output φ_S :* `F` *of Algorithm 10, the following statement holds:*

$$\forall x \in vars(\varphi).\varphi(x) \iff \varphi_S(x) \tag{4.3}$$

Proof outline. The algorithm starts by defining $F_S := \varphi$. The algorithm then applies a number of simplification rules on F_S. All rules in SimplifyF clearly preserve the truth value of the formula being simplified, including the call to SimplifyE for expressions within it (Lemma 4.1.2).

The algorithm now has two branches. Let x be an arbitrary point in $vars(\varphi)$. If $\varphi = F_S$, the algorithm returns F_S. Here, $\varphi(x) \iff \varphi_S(x)$ is trivial. If $\varphi \neq F_S$, the algorithm then recurses with F_S. Since all other

Algorithm 10 SimplifyF: simplify a VC with symbolic rules

Input: (φ : `F`)
Output: (φ_S : `F`)

F_S := φ
F_S := in F_S, transform each $(t_1 < t_2) \vee (t_1 = t_2)$ into $t_1 \leq t_2$ # t_1, t_2 can be any expression, commutative version also applies
F_S := in F_S, transform each $(t_1 > t_2) \vee (t_1 = t_2)$ into $t_1 \geq t_2$ # also its commutative version
F_S := in F_S, transform each $(t_1 \geq t_2) \wedge (t_1 \leq t_2)$ into $t_1 = t_2$ # also its commutative version
F_S := in F_S, transform each $(\varphi_1 \implies \varphi_2) \wedge (\neg\varphi_1 \implies \varphi_2)$ into φ_2 # also its commutative version
F_S := in F_S, transform each $(\varphi_1 \implies \varphi_2) \wedge (\neg\varphi_1 \implies \varphi_3)$ into $\varphi_2 \wedge \varphi_3$ # also its commutative version
F_S := in F_S, transform each $\varphi_1 \wedge (\varphi_1 \implies \varphi_2)$ into $\varphi_1 \wedge \varphi_2$ # also its commutative version
F_S := in F_S, transform each $\varphi_1 \wedge (\neg\varphi_1 \vee \varphi_2)$ into $\varphi_1 \wedge \varphi_2$ # also its commutative version
F_S := in F_S, transform each $\neg\varphi_1 \wedge (\varphi_1 \vee \varphi_2)$ into $\neg\varphi_1 \wedge \varphi_2$ # also its commutative version
F_S := in F_S, replace each $\varphi_1 \wedge \neg\varphi_1$ with false # also its commutative version
F_S := in F_S, transform each $\varphi_1 \vee \mathit{true}$ into true # also its commutative version
F_S := in F_S, transform each $\varphi_1 \vee \mathit{false}$ into φ_1 # also its commutative version
F_S := in F_S, replace each $(\varphi_1 \vee \neg\varphi_1)$ with true # also its commutative version
F_S := in F_S, transform each $\mathit{false} \implies \varphi_1$ into true
F_S := in F_S, transform each $\varphi_1 \implies \mathit{true}$ into true
F_S := in F_S, transform each $\varphi_1 \implies \mathit{false}$ into $\neg\varphi_1$
F_S := in F_S, transform each $\mathit{true} \implies \varphi_1$ into φ_1
F_S := in F_S, transform each $\varphi_1 \implies \neg\varphi_1$ into $\neg\varphi_1$
F_S := in F_S, transform each $\neg\varphi_1 \implies \varphi_1$ into φ_1
F_S := in F_S, replace each $\varphi_1 \implies \varphi_1$ with true
F_S := in F_S, evaluate all comparisons of literals and replace the comparison with the evaluated truth value
F_S := in F_S, transform each $\neg(\neg\varphi_1)$ into φ_1
F_S := in F_S, transform each $\neg\mathit{false}$ into true
F_S := in F_S, transform each $\neg\mathit{true}$ into false
F_S := apply SimplifyE on each expression in F_S
if φ = F_S **then** # If none of the simplification rules were applied, stop simplifying
    return F_S
else # Keep simplifying until simplification rules do not apply
    return SimplifyF(F_S)
end if

simplification rules preserve the truth value of the formula being simplified, this recursion is safe. Thus, $\varphi(x) \iff \varphi_S(x)$ □

We can now define Algorithm 11 which performs further simplifications on any input of type `F` in addition to the simplification rules discussed in Algorithm 10.

Algorithm 11 Simplify: simplify a VC with symbolic rules

Input: (φ : `F`)
Output: (φ_S : `F`)

1: F_S := SimplifyF(φ)
2: **repeat**
3:     F_I := F_S
4:     **if** F_S contains an equality of the form $v = t$ where v is a variable, t : `E`, and t does not contain v **then**
5:         F_S := in F_S, replace each occurrence of v with t.
6:     **end if**
7:     F_S := SimplifyF(F_S)
8: **until** $F_I = F_S$
9: **repeat**
10:     F_I := F_S
11:     **if** a variable x represents π as described in Section 4.1.1 **then**
12:         F_S := in F_S, replace each occurrence of x with π.
13:     **end if**
14: **until** $F_I = F_S$
15: **return** F_S

Lemma 4.1.5 (Termination of Simplify)**.** *For any input φ :* `F`*, Algorithm 11 terminates.*

Proof outline. The algorithm calls Algorithm 10 on lines 1 and 7 which, by Lemma 4.1.3, terminates. From Remark 3, we know that successful application of the rules on lines 2 and 8 reduces the number of operations within the formula being simplified. Successful application of the rule on lines 6 and 13 clearly reduces the number of variables within the formula. It is clear that successful application of any of the rules do not increase the number of variables within the formula.

Now, we discuss the first loop. Successful application of the rule on line 6 reduces the number of variables within the formula. Since there are a finite number of variables, eventually, this rule cannot be applied. Similarly, successful application of the rule on line 8 reduces the number of operations within the formula. Since there are a finite number of operations, eventually, this rule cannot be applied. So, this loop must terminate.

Now we discuss the second loop. Successful application of the rule on line 13 reduces the number of variables within the formula. Since there are a finite number of variables, eventually, this rule cannot be applied. So, this loop must terminate.

Since both loops terminate, Algorithm 11 terminates. □

Building on the discussion of termination and soundness of Algorithm 10, and the termination of Algorithm 11, we now discuss soundness of Algorithm 11.

Lemma 4.1.6 (Soundness of Simplify)**.** *The PropaFP simplification steps as defined in Algorithm 11 soundly simplify any constraint φ of type* $\mathtt{F}$*. The algorithm outputs φ_S of type* $\mathtt{F}$*. The following statement holds:*

$$\forall x \in vars(\varphi).\varphi(x) \iff \varphi_S(x) \tag{4.4}$$

Proof outline. The algorithm assigns $F := \varphi$ and then calls SimplifyF with input F as defined in 10, producing F_S. From Lemma 4.1.4, we know that F and F_S have the same truth value for any point.

Now, the algorithm loops. Within the loop, the algorithm substitutes variables defined as an equality $v = t$, where t does not contain v, in F_S with the value of the variable. This step preserves the truth value of $F)S$. SimplifyF is then called on F_S which preserves the truth value. These two steps are repeated until no further variable substitutions can occur.

Finally, if a variable can be assumed to semantically mean π as defined in Section 4.1.2, replace the variable with π. This step preserves the truth value, semantically, of F_S.

The algorithm outputs F_S. Since all steps in the algorithm preserve the truth value of F, and $F = \varphi, \forall x \in vars(\varphi).\varphi(x) \iff \varphi_S(x)$. Algorithm 11 is sound. $\square$

4.2 Bounds Derivations

Deriving bounds for variables proceeds as follows:

- Identify inequalities which contain only a single variable on either side.
- Iteratively improve bounds by interval-evaluating the expressions given by these inequalities.
  - For our interval evaluations, we use intervals with floating-point endpoints with a precision of 60
  - Initially the bounds for each variable are $-\infty$ and ∞.
  - For FP rounding $rnd(x)$, we overestimate the rounding error by the interval expression $x \cdot (1 \pm \epsilon) \pm \zeta$ where ϵ is the machine epsilon, and ζ is the machine epsilon for denormalized numbers for the precision of the rounded operation.
- Variables are assumed to be real unless they are declared integer.
- For integer variables, trim their bounds by rounding the lower bounds upwards towards the nearest integer and by rounding the upper bounds downwards towards the nearest the integer.

Next, use the derived bounds to potentially further simplify the NVC:

- Evaluate all formulas in the NVC using interval arithmetic.
- If an inequality is decided by this evaluation, replace it with $true$ or $false$.

Finally, repeat the symbolic simplification steps, e.g., to remove any tautologies that have arisen from the interval evaluation. These steps are shown more formally in Algorithm 16. We first define several auxiliary algorithms.

Algorithm 12 EContainsVars : Check if an expression contains at least one variable from a list of variables

Input: (t : `E`, vars : `[String]`)
Output: (b : $\mathbb{B}$)

switch t **do**
  case `Var` v **return** true if $v \in$ vars
  case `Lit` l **return** $false$
  case `EBinOp op` t_1 t_2 **return** EContainsVars(t_1, vars) $\vee$ EContainsVars(t_2, vars)
  case `EUnOp op` t_1 **return** EContainsVars(t_1, vars)
  case `PowI` t_1 i **return** EContainsVars(t_1, vars)
  case `Float32 roundingMode` t_1 **return** EContainsVars(t_1, vars)
  case `Float64 roundingMode` t_1 **return** EContainsVars(t_1, vars)
  case `Float roundingMode` t_1 **return** EContainsVars(t_1, vars)
  case `RoundToInteger roundingMode` t_1 **return** EContainsVars(t_1, vars)

Lemma 4.2.1 (Termination of EContainsVars)**.** *For any inputs t of type* `E`*, vars :* [`String`]*, Algorithm 12 terminates.*

Proof outline. The algorithm traverses over every operation in t at most once. Thus, the algorithm cannot recurse more than $\mathrm{size_o}(t)$ times. Algorithm 12 terminates. □

Lemma 4.2.2 (Soundness of EContainsVars)**.** *For any inputs t :* `E`*, vars :* [`String`]*, and corresponding output b : $\mathbb{B}$ for Algorithm 12, the following statements hold:*

- *If any variable in t is in the list vars, $b = true$.*
- *If all variables in t are not in the list vars, $b = false$.*

Proof outline. The algorithm traverses the syntax tree of the expression, visiting each variable within. If any of the variables we visit is in the vars, we propagate a true result all the way back to the root. Since results of sibling branches are combined using a conjunction, $b = true$.

Conversely, if none of the variables we visit are in the vars list, no true result is ever propagated back to the root. $b = \mathit{false}$. Algorithm 12 is sound. □

Lemma 4.2.3 (Termination of FilterVarsF)**.** *For any inputs φ of type* $\mathtt{F}$*, vars :* $[\mathtt{String}]$*, $\mathit{isNegated} \in \mathbb{B}$, Algorithm 13 terminates.*

Proof outline. The algorithm traverses over every operation in φ at most once. Thus, the algorithm cannot recurse more than $\mathrm{size_o}(\varphi)$ times. The algorithm calls Algorithm 12 which, according to Lemma 4.2.1, terminates. Thus, Algorithm 13 terminates. □

Lemma 4.2.4 (Soundness of FilterVarsF)**.** *For any inputs φ of type* $\mathtt{F}$*, vars :* $[\mathtt{String}]$*, $\mathit{isNegated} \in \mathbb{B}$, and corresponding output φ_O :* $\mathtt{F}$ *for Algorithm 13, the following statement holds:*

- *If $\mathit{isNegated} = \mathit{true}$ then $\forall x \in \mathit{vars}(\varphi).\varphi(x) \implies \varphi_O(x')$*
- *If $\mathit{isNegated} = \mathit{false}$ then $\forall x \in \mathit{vars}(\varphi).\neg\varphi(x) \implies \neg\varphi_O(x')$*
- *φ_O does not contain any variable from vars*

where n is the number of variables in φ and x' is the projection of x to the variables in φ_O.

Proof outline. We first discuss the case where $\mathit{isNegated} = \mathit{false}$. The algorithm makes use of recursion. Within the proof outline, if a formula takes x' instead of some universally quantified x, x' is the projection of x to the variables in the formula. We first discuss soundness of the non-recursive branches,

The algorithm switches based on the value of φ. If φ is equal to $t_1 \diamondsuit t_2$ where $\diamondsuit \in \{<, \leq, >, \geq, =\}$, the algorithm calls Algorithm 12 for both t_1 and t_2 along with the vars list. From Lemma 4.2.2, we know that Algorithm 12 will output true if the given term contains a variable which is in the vars list. If EContainsVars outputs false for both cases, from Lemma 4.2.2, we know that both t_1 and t_2 do not contain any variables in vars. In this case, $\varphi_O = \varphi$

Algorithm 13 FilterVarsF : Attempts to filter out given variables from the given formula by weakening the formula

Input: (φ : `F`, vars : [String], $isNegated$: $\mathbb{B}$)
Output: Potentially φ_O : `F`

1: F, isN := φ, $isNegated$
2: **switch** (F, isN) **do**
3: **case** (`FNot` f, any) **return** `FNot` (FilterVarsF(f, vars, $\neg$ isN))
4: **case** (`FComp op` E_1 E_2, any)
5: **switch** (EContainsVars(E_1, vars, isN), EContainsVars(E_2, vars, isN)) **do**
6: **case** ($false$, $false$) **return** `FComp op` E_1 E_2
7: **case** otherwise **return** Could not weaken F by filtering out vars
8: **case** (`FConn And` F_1 F_2, $false$)
9: **switch** (FilterVarsF(F_1, vars, isN), FilterVarsF(F_2, vars, isN)) **do**
10: **case** (F_1', F_2') **return** `FConn And` F_1' F_2'
11: **case** (F_1', no result) **return** F_1'
12: **case** (no result, F_2') **return** F_2'
13: **case** otherwise **return** Could not weaken F by filtering out vars
14: **case** (`FConn Or` F_1 F_2, $false$)
15: **switch** (FilterVarsF(F_1, vars, isN), FilterVarsF(F_2, vars, isN)) **do**
16: **case** (F_1', F_2') **return** `FConn Or` F_1' F_2'
17: **case** otherwise **return** Could not weaken F by filtering out vars
18: **case** (`FConn Impl` F_1 F_2, $false$)
19: **switch** (FilterVarsF(F_1, vars, $\neg$ isN), FilterVarsF(F_2, vars, isN)) **do**
20: **case** (F_1', F_2') **return** `FConn Impl` F_1' F_2'
21: **case** otherwise **return** Could not weaken F by filtering out vars
22: **case** (`FConn And` F_1 F_2, $true$)
23: **switch** (FilterVarsF(F_1, vars, isN), FilterVarsF(F_2, vars, isN)) **do**
24: **case** (F_1', F_2') **return** `FConn And` F_1' F_2'
25: **case** otherwise **return** Could not weaken F by filtering out vars
26: **case** (`FConn Or` F_1 F_2, $true$)
27: **switch** (FilterVarsF(F_1, vars, isN), FilterVarsF(F_2, vars, isN)) **do**
28: **case** (F_1', F_2') **return** `FConn Or` F_1' F_2'
29: Remaining cases are identical to the cases in lines 11–13
30: **case** (`FConn Impl` F_1 F_2, $true$)
31: **switch** (FilterVarsF(F_1, vars, $\neg$ isN), FilterVarsF(F_2, vars, isN)) **do**
32: **case** (F_1', F_2') **return** `FConn Impl` F_1' F_2'
33: **case** (F_1', no result) **return** `FNot` F_1'
34: **case** (no result, F_2') **return** F_2'
35: **case** otherwise **return** Could not weaken F by filtering out vars
36: **case** (`FTrue`, any) **return** `FTrue`
37: **case** (`FFalse`, any) **return** `FFalse`

which is trivially sound. If EContainsVars outputs true for either t_1 or t_2 (or both), from Lemma 4.2.2, we know that t_1 or t_2 (or both) contain at least one variable from the vars list, so the algorithm gives up.

If φ is equal to either `FTrue` or `FFalse`, again we have $\varphi_O = \varphi$ which is trivially sound. Thus, all non-recursive branches are sound. We now discuss the recursive branches.

If φ is equal to $\neg\varphi_1$, the algorithm recurses with inputs φ_1, vars, and $\neg \mathit{isNegated}$, soundly propagating the negation. Let the output of the recursive call be $\varphi_{1,O}$. Assume that the result is sound, i.e., $\forall x \in \mathit{vars}(\varphi_1).\neg\varphi_1(x) \implies \neg\varphi_{1,O}(x')$. Since $\varphi_O := \neg\varphi_{1,O}$, clearly, $\forall x \in \mathit{vars}(\varphi).\varphi(x) \implies \varphi_O(x')$.

If φ is equal to $\varphi_1 \wedge \varphi_2$, the algorithm recursively calls FilterVarsF with inputs φ_1, vars, $\mathit{isNegated}$ and φ_2, vars, $\mathit{isNegated}$. Let the results of these recursive calls be $\varphi_{1,O}$ and $\varphi_{2,O}$, respectively. If both recursive calls completed successfully, let $\varphi_O := \varphi_{1,O} \wedge \varphi_{2,O}$. Assuming that $\forall x \in \mathit{vars}(\varphi_1).\varphi_1(x) \implies \varphi_{1,O}(x')$ and $\forall x \in \mathit{vars}(\varphi_2).\varphi_2(x) \implies \varphi_{2,O}(x')$, it must be true that $\forall x \in \mathit{vars}(\varphi).\varphi(x) \implies \varphi_O(x')$. If the recursive call for φ_1 completed successfully but not for φ_2, we have $\varphi_O = \varphi_{1,O}$. Recall that removing a term from a conjunction weakens the conjunction. Assuming that $\forall x \in \mathit{vars}(\varphi_1).\varphi_1(x) \implies \varphi_{1,O}(x')$, clearly $\forall x \in \mathit{vars}(\varphi).\varphi(x) \implies \varphi_O(x')$. There is a similar case if the recursive call for φ_2 completed successfully but not for φ_1. If both recursive calls fail, the algorithm gives up.

If φ is equal to $\varphi_1 \vee \varphi_2$, the algorithm recursively calls FilterVarsF inputs φ_1, vars, $\mathit{isNegated}$ and φ_2, vars, $\mathit{isNegated}$. Let the results of these recursive calls be $\varphi_{1,O}$ and $\varphi_{2,O}$, respectively. If both recursive calls completed successfully, let $\varphi_O := \varphi_{1,O} \vee \varphi_{2,O}$. Assuming that $\forall x \in \mathit{vars}(\varphi_1).\varphi_1(x) \implies \varphi_{1,O}(x')$ and $\forall x \in \mathit{vars}(\varphi_2).\varphi_2(x) \implies \varphi_{2,O}(x')$, it must be true that $\forall x \in \mathit{vars}(\varphi).\varphi(x) \implies \varphi_O(x')$. If either of the recursive calls fail, the algorithm gives up.

If φ is equal to $\varphi_1 \implies \varphi_2$, the algorithm treats φ as its equivalent disjunction, i.e., $\neg\varphi_1 \vee \varphi_2$. So, the algorithm recursively calls FilterVarsF with inputs φ_1, vars, $\neg\mathit{isNegated}$ and φ_2, vars, $\mathit{isNegated}$. If both recursive calls completed successfully, let $\varphi_O := \varphi_{1,O} \implies \varphi_{2,O}$ which is equivalent to $\neg\varphi_{1,O} \vee \varphi_{2,O}$. Because $\neg\mathit{isNegated}$ is true, we can assume that $\forall x \in$

$vars(\varphi_1).\neg\varphi_1(x) \implies \neg\varphi_{1,O}(x')$ and $\forall x \in vars(\varphi_2).\varphi_2(x) \implies \varphi_{2,O}(x')$. It must be true that $\forall x \in vars(\varphi).\varphi(x) \implies \varphi_O(x')$. If either of the recursive calls fail, the algorithm gives up.

We have now discussed every possible case where $isNegated = false$. The cases where $isNegated = true$ are shown analogously.

Since we have proven that every branch soundly deals with the result of any recursive call, and all non-recursive branches are sound, Algorithm 13 is sound. $\square$

Lemma 4.2.5 (Termination of ScanAndDerive)**.** *For any φ : $\mathtt{F}$, $\mathbf{b}$: $\mathbb{IR}^{*vars(\varphi)}$, $isNegated$: $\mathbb{B}$, Algorithm 14 terminates.*

Proof outline. The algorithm traverses over every non `FComp` operation in φ at most once. Thus, if φ does not contain any `FComp` operations, the algorithm cannot recurse more than $\mathrm{size_o}(\varphi)$ times.

If φ does contain an `FComp` operator, but the operator does not have only a variable on the LHS or RHS, the algorithm will return B once reaching the `FComp` operator.

If φ does contain an `FComp` operator *and* the operator has *only* a variable on the LHS or RHS, the algorithm branches depending on $isNegated$. We first discuss the case where $isNegated$ is true. Let the `FComp` operator we are discussing be described as $v \diamondsuit t$ where v is a variable, t is an expression, and $\diamondsuit \in \{<, \leq, >, \geq, =\}$. If $\diamondsuit$ is $=$, the algorithm returns $\mathbf{b}$. Otherwise, the algorithm recurses with a negation of $v \diamondsuit t$, $\mathbf{b}$, and with $isNegated := false$. Since we set $isNegated$ to $false$, this recursion can only happen once.

Now, we discuss the case where $isNegated$ is false. Here, we start a loop. Within the loop, we attempt to improve the bounds described for v by interval evaluating $v \diamondsuit t$ using floating-point interval arithmetic with a fixed precision of 60. If the interval evaluation shows that $v \diamondsuit t$ describes a better bound for v, improve the bound and loop, repeating the above steps. This repetition is useful because better bounds for B can result in a better interval evaluation for $v \diamondsuit t$ which can, again, result in better bounds for B and so on. Because we use floating-point interval arithmetic with a fixed precision, this

Algorithm 14 ScanAndDerive: Scan through a formula, deriving bounds where possible

Input: (φ : $\texttt{F}$, $\mathbf{b}$: $\mathbb{IR}^{*vars(\varphi)}$, $isNegated$: $\mathbb{B}$)

Output: $\mathbf{b}_S$: $\mathbb{IR}^{*vars(\varphi)}$

F, B, isN := φ, $\mathbf{b}$, $isNegated$

switch F **do**

  case $\texttt{FNot}$ F_1 **return** (ScanAndDerive(F_1, B, $\neg$ isN))

  case $\texttt{FConn And}$ F_1 F_2

    if isN **then**

      return ScanAndDerive($\texttt{FConn Or}$ ($\texttt{FNot}$ F_1) ($\texttt{FNot}$ F_2), B, $\neg$ isN)

    else

      B_2 = ScanAndDerive(F_2, B, isN)

      return ScanAndDerive(F_1, B_2, isN)

    end if

  case $\texttt{FConn Or}$ F_1 F_2

    if isN **then**

      return ScanAndDerive($\texttt{FConn And}$ ($\texttt{FNot}$ F_1) ($\texttt{FNot}$ F_2), B, $\neg$ isN)

    else

      B_1 = ScanAndDerive(F_1, B, isN)

      B_2 = ScanAndDerive(F_1, B, isN)

      return hull($B_1 \cup B_2$)

    end if

  case $\texttt{FConn Impl}$ F_1 F_2

    return ScanAndDerive($\texttt{FConn Or}$ ($\texttt{FNot}$ F_1) F_2, B, isN)

  case $\texttt{FComp op}$ ($\texttt{Var}$ v) t # also its commutative version

    if isN **then**

      switch $\texttt{op}$ **do**

        case $\texttt{Eq}$ **return** B # $v \neq t$ does not give useful information

        case $\texttt{Ge}$ **return** ScanAndDerive($\texttt{FComp Lt}$ (Var v) t, B, $\neg$ isN)

        case $\texttt{Gt}$ **return** ScanAndDerive($\texttt{FComp Le}$ (Var v) t, B, $\neg$ isN)

        case $\texttt{Le}$ **return** ScanAndDerive($\texttt{FComp Gt}$ (Var v) t, B, $\neg$ isN)

        case $\texttt{Lt}$ **return** ScanAndDerive($\texttt{FComp Ge}$ (Var v) t, B, $\neg$ isN)

    else

      repeat

        B_L := B

        t_R := evaluate t over B using floating-point interval arithmetic with precision 60

        if x $\texttt{op}$ t_R describes a *larger* lower or *smaller* upper bound for x than the one in B **then** B := improve the bound of v in B with v $\texttt{op}$ t_R

        end if

      until $B_L = B$

      return B

    end if

  case any **return** B

cycle of improving B by getting a better interval evaluation for $v\diamondsuit t$ must be finite. Thus, the loop terminates. Once the loop terminates, the algorithm returns B.

Since the algorithm always returns B when it comes across an `FComp` operation with only a variable on one side, and traverses through all other operations at most once, Algorithm 14 terminates. $\square$

Lemma 4.2.6 (Soundness of ScanAndDerive)**.** *For any inputs φ : `F`, $\mathbf{b}$: $\mathbb{IR}^{*vars(\varphi)}$, $isNegated$: $\mathbb{B}$, and corresponding output $\mathbf{b}_S$: $\mathbb{IR}^{*vars(\varphi)}$ for Algorithm 14, the following statements hold:*

- $\mathbf{b}_S \subseteq \mathbf{b}$
- *If $isNegated = false$ then $\forall x \in \mathbf{b}.\varphi(x) \implies x \in \mathbf{b}_S$*
- *If $isNegated = true$ then $\forall x \in \mathbf{b}.\neg\varphi(x) \implies x \in \mathbf{b}_S$*

Proof outline. The algorithm traverses through φ, looking for comparisons where there is only a variable on one side and any term on the other. We first discuss the branches where the value of $isNegated$ is not important.

If φ is equal to $\neg\varphi_1$, the algorithm recurses with inputs φ_1, $\mathbf{b}$, and $\neg isNegated$, soundly propagating the negation. Let the output of the recursive call be $\mathbf{b}_S$. Assuming that the result of the recursive call is sound, then $\mathbf{b}_S \subseteq \mathbf{b}$ and $\forall x \in \mathbf{b}.\varphi(x) \implies x \in \mathbf{b}_S$

If φ is equal to $\varphi_1 \implies \varphi_2$, the algorithm treats the implication as its equivalent disjunction, recursing with inputs $\neg\varphi_1 \vee \varphi_2$, $\mathbf{b}$, and $isNegated$. Let the output of the recursive call be $\mathbf{b}_S$. Assuming that the result of the recursive call is sound, then $\mathbf{b}_S \subseteq \mathbf{b}$ and $\forall x \in \mathbf{b}.\varphi(x) \implies x \in \mathbf{b}_S$ We now discuss the cases where $isNegated$ is false.

Let $B := \mathbf{b}$. If φ is a comparison of the form $v\diamondsuit t$ where v is a variable, t is any term, and $\diamondsuit \in \{<, \leq, >, \geq, =\}$, the algorithm attempts to improve the bound for v by interval evaluating $v\diamondsuit t$ using floating-point interval arithmetic with a precision of 60. This is repeated until the bound for v stops improving, at which point, the algorithm outputs B. After this operation, clearly, $\mathbf{b}_S \subseteq \mathbf{b}$. Since the algorithm improves the bounds for v using φ, clearly $\forall x \in \mathbf{b} \setminus \mathbf{b}_S.\neg\varphi$, so $\forall x \in \mathbf{b}.\varphi(x) \implies x \in \mathbf{b}_S$. This branch is sound.

If φ is $true$, $false$, or any comparison other than the one described above, the algorithm outputs $\mathbf{b}$ which is trivially sound. Every non-recursive branch is sound. We now discuss the recursive branches.

First, we discuss the branches where $isNegated$ is false. If φ is $\varphi_1 \wedge \varphi_2$, the algorithm recurses twice. First, the algorithm recurses with inputs φ_2, $\mathbf{b}$, and $isNegated$. Let the output of this recursive call be $\mathbf{b}_2$. Assuming that the result of the recursive call is sound, then $\mathbf{b}_2 \subseteq \mathbf{b}$ and $\forall x \in \mathbf{b}.\varphi(x) \implies x \in \mathbf{b}_2$. Since $\mathbf{b}_2$ soundly describes bounds for φ_2, and $\varphi = \varphi_1 \wedge \varphi_2$, we recurse with inputs φ_1, $\mathbf{b}_2$, and $isNegated$. The output of this recursive call $\mathbf{b}_S$. Assuming that the result of the recursive call is sound, then $\mathbf{b}_S \subseteq \mathbf{b}_1$ and $\forall x \in \mathbf{b}_1.\varphi(x) \implies x \in \mathbf{b}_S$. Since $\mathbf{b}_1 \subseteq \mathbf{b}$, this is sound.

If φ is $\varphi_1 \vee \varphi_2$, the algorithm recurses twice. First, the algorithm recurses with inputs φ_1, $\mathbf{b}$, and $isNegated$. Let the output of this recursive call be $\mathbf{b}_1$. Assuming that the result of the recursive call is sound, then $\mathbf{b}_1 \subseteq \mathbf{b}$ and $\forall x \in \mathbf{b}.\varphi(x) \implies x \in \mathbf{b}_1$. Then, the algorithm recurses with inputs φ_2, $\mathbf{b}$, and $isNegated$. Let the output of this recursive call be $\mathbf{b}_2$. Assuming that the result of the recursive call is sound, then $\mathbf{b}_2 \subseteq \mathbf{b}$ and $\forall x \in \mathbf{b}.\varphi(x) \implies x \in \mathbf{b}_2$. Since $\varphi = \varphi_1 \vee \varphi_2$, $\mathbf{b}_1$ soundly describes bounds for φ_1, and likewise for $\mathbf{b}_2$ and φ_2, the algorithm outputs the hull of inputs $\mathbf{b}_1$ and $\mathbf{b}_2$. Clearly, $\mathbf{b}_1 \subseteq \mathbf{b}_S$ and $\mathbf{b}_2 \subseteq \mathbf{b}_2$. Since $\mathbf{b}_S$ is the box hull of $\mathbf{b}_1$ and $\mathbf{b}_2$, $\mathbf{b}_S$ will never leave the boundaries of either $\mathbf{b}_1$ or $\mathbf{b}_2$, and since both $\mathbf{b}_1$ and $\mathbf{b}_2$ are subboxes of $\mathbf{b}$, $\mathbf{b}_S \subseteq \mathbf{b}$. Since $\mathbf{b}_1 \subseteq \mathbf{b}_S \subseteq \mathbf{b}$ and similar for $\mathbf{b}_2$, $\forall x \in \mathbf{b}.\varphi(x) \implies x \in \mathbf{b}_S$.

Now, we discuss the branches where $isNegated$ is true. If φ is $\varphi_1 \vee \varphi_2$, the algorithm applies the negation and recurses with inputs $\neg\varphi_1 \wedge \neg\varphi_2$, $\mathbf{b}$, $\neg isNegated$, outputting $\mathbf{b}_S$. Assuming that the result of the recursive call is sound, since we soundly apply the negation, then $\mathbf{b}_S \subseteq \mathbf{b}$ and $\forall x \in \mathbf{b}.\neg\varphi(x) \implies x \in \mathbf{b}_S$.

If φ is $\varphi_1 \wedge \varphi_2$, the algorithm applies the negation and recurses with inputs $\neg\varphi_1 \vee \neg\varphi_2$, $\mathbf{b}$, $\neg isNegated$, outputting $\mathbf{b}_S$. Assuming that the result of the recursive call is sound, since we soundly apply the negation, then $\mathbf{b}_S \subseteq \mathbf{b}$ and $\forall x \in \mathbf{b}.\neg\varphi(x) \implies x \in \mathbf{b}_S$.

Since we have proven that every branch soundly deals with the result of any recursive call, and all non-recursive branches are sound, Algorithm 14 is sound. □

Algorithm 15 DeriveBounds: Iteratively derive bounds for variables

Input: (φ : $\texttt{F}$, intVars : [$\texttt{String}$])
Output: $\mathbf{b}_S$: $\mathbb{IR}^{*vars(\varphi)}$ and a formula φ_S

1: F_S := SimplifyF(φ)
2: B := a box for all variables in F_S with initial bounds set to $\pm\infty$
3: **repeat**
4: $B_L = B$
5: B := ScanAndDerive(F_S, B, false)
6: F_I := replace in F_S comparisons with the appropriate truth value if they can be decided using interval arithmetic over B
7: F_S := SimplifyF(F_I)
8: **until** $B_L = B$
9: **for all** variables $v \in$ intVars **do**
10: B = ceil the lower bound and floor the upper bound of $v \in B$ to the nearest integer
11: **end for**
12: **return** B, F_S

Lemma 4.2.7 (Termination of DeriveBounds)**.** *For any inputs φ : $\texttt{F}$, intVars : [$\texttt{String}$], Algorithm 15 terminates.*

Proof outline. The algorithm calls Algorithm 10 with input φ, which terminates (Lemma 4.1.3), outputting F_S. Let B be a box describing all variables in F_S with initial bounds set to $\pm\infty$

The algorithm loops. Let $B_L := B$. The loop terminates when, after any iteration, $B_L = B$. Overwrite B with the output of Algorithm 14 with inputs F_S, B, false. From Lemma 4.2.5, we know that $B \subseteq B_L$. Let F_I be the result of attempting to replacing comparisons in F_S with the appropriate truth value after interval evaluating the comparison over B. If any comparisons were successfully replaced, then $\mathrm{size_o}(F_I) < \mathrm{size_o}(F_S)$. Overwrite F_S with the output of Algorithm 10 with input F_I. If any of the rules in Algorithm 10 were successfully applied, then clearly $\mathrm{size_o}(F_S) < \mathrm{size_o}(F_I)$. Thus, the rule on line 7 along with the rules in Algorithm 10 can be applied, at most, $\mathrm{size_o}(F_S)$

times. Clearly, if Algorithm 14 is called with the same inputs, it will produce the same output. Thus, the first loop must terminate.

In the second loop, we iterate through every integer variable in B once. There are a finite number of variables in B. This loop terminates. Algorithm 15 terminates.

□

Lemma 4.2.8 (Soundness of DeriveBounds)**.** *For any inputs* φ *:* `F`*, intVars :* [`String`]*, and corresponding outputs* $\mathbf{b}_S$ *:* $\mathbb{IR}^{*n}$*,* φ_S *:* `F` *for Algorithm 15, the following statement holds:*

- $\forall x \in \mathbb{IR}^{*vars(\varphi)}.(\forall v \in \text{intVars}.x_v \in \mathbb{Z}).\varphi_S(x) \implies x \in \mathbf{b}_S$
- $\forall x \in \mathbf{b}_S.\varphi(x) \iff \varphi_S(x)$

Proof outline. The algorithm calls Algorithm 10 with input φ, storing the output in F_S From Lemma 4.1.4, $\forall x \in vars(\varphi).\varphi(x) \iff F_S$. Let B be a box of type $\mathbb{IR}^{*vars(\varphi)}$ defined for all variables in φ with initial endpoints set to $\pm\infty$.

The algorithm loops. Let $B_L := B$. Let B_O be the output of Algorithm 14 with inputs F_S, B, and $false$. From Lemma 4.2.6, we know that $B_O \subseteq B$ and $\forall x \in B.F_S(x) \implies x \in B_O$. Clearly, $\forall x \in \mathbb{IR}^{*vars(\varphi)}.F_S(x) \implies x \in B_O$. Let F_I be F_S where we evaluate comparisons over B_O and, if possible, replace the comparison with the evaluated truth value. Clearly, $\forall x \in B_O.F_S(x) \iff F_I(x)$. Let F'_S be the output of Algorithm 10 with input F_I. From Lemma 4.1.4, $\forall x \in vars(F_I).F_I(x) \iff F'_S(x)$. If $\mathbf{b}_6 \neq \mathbf{b}_5$, then it must be true that $\mathbf{b}_6 \subseteq \mathbf{b}_5$. The algorithm loops where, in the next iteration, $\varphi_{S,2} := F'_S$ and $\mathbf{b}_5 := \mathbf{b}_6$. So, we have the following invariants.

1. $B_O \subseteq B$
2. $\forall x \in \mathbb{IR}^{*vars(\varphi)}.F'_S(x) \implies x \in B_O$
3. $\forall x \in B_O.\varphi(x) \iff F'_S(x)$

Let B_E and F_E be the value of B_O and F'_S, respectively, after the loop has finished. Since, in the first iteration, we call Algorithm 14 with inputs B_O, and in the first iteration, $B_O = B$, it must be true that $\forall x \in \mathbb{IR}^{*vars(\varphi)}.\varphi(x) \implies x \in B_E$. Since F_E is a version of φ where SimplifyF has been called repeatedly and comparisons have been replaced with their evaluated truth value over B_E, it must be true that $\forall x \in B_E.\varphi(x) \iff F_E(x)$.

The second loop rounds interval variables in B_E by rounding lower bounds upwards to the nearest integer and upper bounds downwards towards the nearest integer which is clearly sound. The algorithm outputs F_E and B_E after this loop has completed. Due to rounding of integer variables, we now have $\forall x \in \mathbb{IR}^{*vars(\varphi)}.(\forall v \in \mathrm{intVars}.x_v \in \mathbb{Z}).\varphi_S(x) \implies x \in \mathbf{b}_S$. Clearly, $\forall x \in \mathbf{b}_S.\varphi(x) \iff \varphi_S(x)$. Algorithm 15 is sound. □

We now define DeriveBoundsAndFilter, an algorithm which calls DeriveBounds for a given formula and then attempts to filter out variables with at least one unbounded endpoint in such a way that the given formula is weakened.

Algorithm 16 DeriveBoundsAndFilter: Derive bounds for variables and attempt to filter out variables with at least one unbounded endpoint.

Input: (φ : `F`, intVars : `[String]`)
Output: $\mathbf{b}$: $\mathbb{IR}^{vars(\varphi)}$ and a formula φ_D

```
vars := list of all variables in φ
(B_S, F_S) := DeriveBounds(φ, intVars)
unboundedVars := vars in B_S with at least one unbounded endpoint
F_W := FilterVarsF(F_S, unboundedVars)    # Removes statements referring to any variable in unboundedVars
if F_W is defined then
    B := remove unboundedVars from B_S
    F_W := if a variable in B has the same upper and lower bound, remove the variable from B and replace the variable with its value in F_W
    F_W^I := interval evaluate comparisons in F_W over b and replace comparisons with their truth value if possible    # Note that this step will replace comparisons which state the bounds for derived variables with FTrue
    F_D := SimplifyF(F_W^I)
    return B, F_D
else
    return Failed to derive bounds for the variables in φ
end if
```

Lemma 4.2.9 (Termination of DeriveBoundsAndFilter)**.** *For any inputs φ : $\mathtt{F}$, intVars : $[\mathtt{String}]$, Algorithm 16 terminates.*

Proof outline. The algorithm calls Algorithms 15, 13, and 11, all of which terminate according to Lemmas 4.2.7, 4.2.3, and 4.1.5, respectively. The rest of the algorithm clearly terminates. □

Building on the discussion of soundness and termination for Algorithms 15, 13, and 11, we now discuss the soundness of Algorithm 16.

Lemma 4.2.10 (Soundness of DeriveBoundsAndFilter)**.** *For any inputs φ : $\mathtt{F}$, intVars : $[\mathtt{String}]$, and corresponding outputs $\mathbf{b}$, φ_D for Algorithm 16, the following statements hold:*

$$\forall x \in \mathbb{IR}^{*vars(\varphi)}.(\forall v \in \mathrm{intVars}.x_v \in \mathbb{Z}).\varphi(x) \implies \varphi_D(x') \wedge x' \in \mathbf{b} \tag{4.5}$$

Where x' is the projection of x to the variables in φ_D.

Proof outline. The algorithm calls DeriveBounds with inputs φ and intVars, storing the output in B_S and F_S. From Lemma 4.2.8, we know $\forall x \in \mathbb{IR}^{*vars(\varphi)}.(\forall v \in \mathrm{intVars}.x_v \in \mathbb{Z}).F_S(x) \implies x \in B_S$ and $\forall x \in B_S.\varphi(x) \iff F_S(x)$ Let vars be a list of variables in B_S with at least one unbounded endpoint. If any variable in B_S has an unbounded endpoint, the algorithm attempts to weaken F_S by calling Algorithm 13 with inputs F_S and vars, outputting F_W. There are two cases.

If Algorithm 13 failed to weaken F, Algorithm 4.2.10 gives up. Otherwise, proceed by removing variables with unbounded endpoints from B_S. From Lemma 4.2.4, we know that $\forall x \in B_S.F_S(x) \implies F_W(x')$ where x' is the projection of x to the variables in F_W. The algorithm then interval evaluates comparisons in F_W over B_S, replacing the comparisons with the resulting truth value if possible. The new constraint is named F_W^I. Clearly, $\forall x \in B_S.F_W(x) \iff F_W^I(x)$. Algorithm 10 is then called with input F_W^I, outputting the constraint F_D. From Lemma 4.1.4, we know that $\forall x \in vars(F_W^I).F_W^I(x) \iff F_D(x)$.

Let x be an arbitrary value from $\mathbb{IR}^{*vars(\varphi)}$. Let x_W be the projection of x to the variables in F_W. Let x' be the projection of x to the variables

Listing 4.4: `Taylor_Sin` NVC after simplification and bounds derivation

```
Bounds on variables:
x (real) ∈ [-0.5, 0.5]

NVC:
assert to_float(RNA, 1) = 1.0
-- The last assertion is unchanged from Listing 4.3 except
-- turning ≥s into equivalent ≤s.
```

in F_D. Let x' be the projection of x to the variables in F_D. Since $(\forall v \in \mathrm{intVars}.x_v \in \mathbb{Z}).F_S(x) \implies x \in B_S$ and $\forall x \in B_S.F_S(x) \implies F_W(x_W) \iff F_W^I(x_W) \iff F_D(x'), \forall x \in vars(\varphi).\varphi(x) \implies F_D(x') \wedge x' \in B_S$. Thus, Algorithm 16 is sound. $\square$

Similarities with Abstract Interpretation. The iterative process in Algorithm 15 can be thought of as a simple form of Abstract Interpretation (AI) over the interval domain [15], but instead of scanning program steps along paths in loops, we scan a set of mutually recursive variable definitions within a formula φ.

The NVC arising from `Taylor_Sin`, shown in Listing 4.3, is already almost in its simplest form. The symbolic steps described in this section applied on this NVC cause `real_pi` (which is present in the omitted assertions) to be replaced with π and also lead to the removal of assertions bounding `X` and `real_pi`. The resulting bounded NVC is outlined in Listing 4.4.

4.2.1 Eliminating floating-point operations

VCs arising from FP programs are likely to contain FP operations. As most provers for real inequalities do not natively support FP operations, we need to eliminate the FP operations before passing the NVCs to a numerical prover. We propose computing an upper bound on the size of the absolute rounding error in expressions using a tool specialised in this task, replacing FP operations with the corresponding real operations, and compensating for the loss of rounding by adding/subtracting the computed error bound. Note that this action weakens the NVCs. Recall that weakening is safe for proving correctness but may lead to incorrect counter-examples. These steps are defined more formally in Algorithm 17

Algorithm 17 EliminateFloats: eliminate floating-point operations within a formula over some box

---

Input: $(\varphi : \mathtt{F}, \mathbf{b} : \mathbb{IR}^{vars(\varphi)})$
Output: $\varphi_E : \mathtt{F}$

$F_I, B := \varphi, \mathbf{b}$
$F :=$ determine type of FP operations in F_I using a bottom-up type derivation
$F_E := F$
for all $c_f = l \Diamond r :=$ comparisons in F_E containing FP operations **do**
    $e_l :=$ an upper bound of the absolute rounding error in l over $\mathbf{b}$
    $e_r :=$ an upper bound of the absolute rounding error in r over $\mathbf{b}$
    $c'_e :=$ replace FP operations in c_f with exact operations
    $c_e :=$ weaken the LHS and RHS in c'_e using e_l and e_r
    $F_E :=$ weaken F_E by replacing c_f with c_e
end for
return F_E

Lemma 4.2.11 (Termination of EliminateFloats)**.** *For any $\varphi : \mathtt{F}$ and any $\mathbf{b} : \mathbb{IR}^n$ where n is the number of variables in φ, Algorithm 17 terminates.*

Proof outline. Let $\mathrm{size_f}$ be a function which counts the number of floating-point operations within some input of type F. The algorithm loops while there are floating-point operations in φ. Let $\varphi_{E,k}$ be the value of φ after k iterations of the loop which removes floating-point operations. $\mathrm{size_f}(\varphi) = \mathrm{size_f}(\varphi_{E,0}) > \mathrm{size_f}(\varphi_{E,1}) > \cdots > \mathrm{size_f}(\varphi_{E,k})$. Since there are a finite number of floating-point operations, and the loop always removes at least one floating-point operation from the formula, eventually the number of floating-point operations in the formula will become zero. Thus, Algorithm 17 terminates. □

We now discuss the soundness of EliminateFloats.

Lemma 4.2.12 (Soundness of EliminateFloats)**.** *For any inputs $\varphi : \mathtt{F}$, $\mathbf{b} : \mathbb{IR}^{vars\varphi}$ and corresponding output $\varphi_E : \mathtt{F}$ for Algorithm 17, the following statements hold:*

- *φ_E does not contain any floating-point operations*
- $\forall x \in \mathbf{b}.\varphi(x) \implies \varphi_E(x)$

Proof outline. Let $F_I := \varphi$ The algorithm loops over all comparisons con-tain-ing floating-point operations in F_I. The loop has the invariant $\forall x \in \mathbf{b}\varphi(x) \implies F_I(x)$. Let x be an arbitrary value in $\mathbf{b}$. We now discuss why this invariant holds. Let c_f be the floating-point containing comparison we are currently looping on. We compute upper bounds on the absolute rounding error for both the LHS and RHS of c_f. The loop then defines c_e, a version of c_f with only exact operations where both the LHS and RHS have been weakened using the computed upper bounds on the absolute error. Within F_I, we replace c_f with c_e. Clearly, c_f is in φ. Since c_e is a weakening of c_f, and F_I contains c_e instead of c_f, clearly $\forall x \in \varphi(x) \implies F_I(x)$. This logic applies for any iteration of the loop. Thus, the loop invariant holds.

Let F_E be the value of F_I after the loop has ended. Since we loop on all floating-point containing comparisons, replacing the floating-point operations with exact operations, F_E does not contain any floating-point operations. Since the loop invariant holds, clearly $\forall x \in \mathbf{b}\varphi(x) \implies F_E(x)$. Thus, Algorithm 17 is sound. □

Currently, in our implementation of EliminateFloats, we use FPTaylor [60] which supports most of the operations we need. In principle, we can use any tool that gives reliable absolute bounds on the rounding error of our FP expressions, such as Gappa [20], Rosa [19] or PRECiSA [61], perhaps enhanced by FPRoCK [59].

There are expressions containing FP operations in the `Taylor_Sin` NVC. The top-level expressions with FP operators are automatically passed to FPTaylor. Listing 4.5 shows an example of how the expressions are specified to FPTaylor. The error bounds computed by FPTaylor for the `Taylor_Sin` NVC expressions that contain FP operators are summarised in Table 4.1.

We can now use these error bounds to safely replace FP operations with exact real operations. Listing 4.6 shows the resulting NVC for `Taylor_Sin`.

There may be statements which can be further simplified thanks to the elimination of FP operations. For example, in Listing 4.6, we have the trivial

`rnd32(1.0)`	0
`sin(x) + (-1 * rnd32((x - rnd32((rnd32((rnd32((x * x)) * x)) / 6)))))`	1.769513e-8
`-1 * (sin(x) + (-1 * rnd32((x - rnd32((rnd32((rnd32((x * x)) * x)) / 6))))))`	1.769513e-8

Table 4.1: Error bounds computed by FPTaylor

Listing 4.5: FPTaylor file to compute an error bound of the `Taylor_Sin` VC

```
Variables
  real x in [-0.5, 0.5];

Expressions
  sin(x) + (-1 *
    rnd32((x - rnd32((rnd32((rnd32((x*x))*x)) / 6)))));
// Computed absolute error bound: 1.769513e-8
```

tautology $1 \pm 0.0 = 1.0$. To capitalise on such occurrences, we could once again interval-evaluate each statement in the NVC. Instead, we invoke the steps from Section 4.1.2 again, which not only include interval evaluation, but also make any consequent simplifications.

We now have derived bounds for variables and a weakened and simplified NVC with no FP operations, ready for provers. We will call this the 'simplified exact NVC'[1].

The entire process of simplifying, deriving bounds for variables, and eliminating floating-point operations in a VC is described in algorithmic form in Algorithm 18.

Lemma 4.2.13 (Termination of PropaFP)**.** *For any φ : `F`, Algorithm 18 terminates.*

Proof outline. The algorithm relies on Algorithms 11, 16, and 17, all of which have been shown to terminate in Lemmas 4.1.5, 4.2.9, and 4.2.11, respectively. The rest of the algorithm clearly terminates. □

Finally, we discuss the soundness of the PropaFP algorithm.

[1] In Table 5.1, this NVC is referred to as Taylor_Sin.

Listing 4.6: `Taylor_Sin` NVC after removal of FP operations

```
Bounds on variables:
x (real) ∈ [-0.5, 0.5]

NVC:
assert 1 ± 0.0 = 1.0
assert
 ¬((
  0.0 ≤ (sin(x) + (-1·(x − ((x·x)·x/6.0))) + 1.769513e^-8)
  ⟹
  (sin(x) + (-1·(x − ((x·x)·x/6.0))) + 1.769513e^-8) ≤
                                        (25889/100000000)
 )∧(
  ¬ (0.0 ≤ (sin(x) + (-1·(x − ((x·x)·x/6.0)))) − 1.769513e^-8)
  ⟹
  (-1·(sin(x) + (-1·(x − ((x·x)·x/6.0)))) + 1.769513e^-8)≤
                                        (25889/100000000)
 ))
```

Listing 4.7: `Taylor_Sin` simplified exact NVC, ready for provers

```
Bounds on variables:
x (real) ∈ [-0.5, 0.5]

NVC:
–– The last assertion is the same as in Listing 4.6
```

Lemma 4.2.14 (Soundness of PropaFP)**.** *For any inputs φ : $\mathtt{F}$, intVars : $[\mathtt{String}]$ and corresponding outputs $\mathbf{b}_{\mathrm{P}}$: box, φ_P : $\mathtt{F}$ for Algorithm 18, the following statement holds:*

1. $\forall x \in \mathbb{IR}^{*vars(\varphi)}.(\forall v \in \mathrm{intVars}.x_v \in \mathbb{Z}).\varphi(x) \implies \varphi_P(x') \wedge x' \in \mathbf{b}$ *where x' is the projection of x to the variables in φ_P.*

2. *φ_P does not contain any floating-point operations.*

Proof outline. The algorithm first calls Simplify on φ, producing F_S. From Lemma 4.1.6, $\forall x \in vars(\varphi).\varphi(x) \iff F_S(x)$. The algorithm then calls DeriveBoundsAndFilter with inputs F_S, intVars. If DeriveBoundsAndFilter

Algorithm 18 PropaFP: simplify, derive bounds for variables, and eliminate floats within a VC

Input: (φ : `F`, intVars : `[String]`)

Output: Potentially $\mathbf{b}_\mathrm{P}$: $\mathbb{IR}^{*vars(\varphi)}$ and a formula φ_P

F_S := Simplify(φ)
if DeriveBoundsAndFilter(F_S, intVars) succeeds **then**
    B_T, F_T := DeriveBoundsAndFilter(F_S, intVars)
    F_E := EliminateFloats(F_T, B_T)
    F_{B_T} := convert B_T to `F`
    $\mathbf{b}_\mathrm{P}$, F_P := DeriveBoundsAndFilter($F_E \wedge F_{B_T}$, intVars) # This call never fails
    return $\mathbf{b}_\mathrm{P}$, F_P
else
    return failed to derive bounds for F
end if

cannot successfully derive bounds for variables in F_S, the algorithm gives up.

If DeriveBoundsAndFilter does successfully derive bounds for variables in F_S, the algorithm stores the outputs in B_T and F_T. From Lemma 4.2.10, we know that $\forall x \in \mathbb{IR}^{*vars(\varphi)}.(\forall v \in \mathrm{intVars}.x_v \in \mathbb{Z}).\varphi(x) \implies F_T(x') \wedge x' \in B_T$ where x' is the projection of x to the variables in F_T

The algorithm then calls EliminateFloats with arguments F_T and B_T, outputting F_E. From Lemma 4.2.12, we know that $\forall x \in B_T.F_T(x) \implies F_E(x)$ and F_E does not contain floating-point operations. Let F_{B_T} be the `F` equivalent of B_T. Clearly, $\forall x \in B_T.F_T(x) \implies (F_E(x) \wedge F_{B_T}(x))$.

The algorithm now calls DeriveBoundsAndFilter on $F_E(x) \wedge F_{B_T}(x)$, outputting F_P and $\mathbf{b}_\mathrm{P}$. Since we derived bounds on $F_E \wedge F_{B_T}$, and F_{B_T} is the `F` equivalent of B_T, $\mathbf{b}_\mathrm{P} = B_T$. From Lemma 4.2.10, we know that $\forall x \in \mathbb{IR}^{*vars(\varphi)}.(\forall v \in \mathrm{intVars}.x_v \in \mathbb{Z}).F_E \wedge F_{B_T} \implies F_P(x') \wedge x' \in \mathbf{b}_\mathrm{P}$ where x' is the projection of x to the variables in F_P.

We have the following facts:

1. $\forall x \in vars(\varphi).\varphi(x) \iff \varphi_S(x)$.

2. $\forall x \in \mathbb{IR}^{*vars(\varphi)}.(\forall v \in \mathrm{intVars}.x_v \in \mathbb{Z}).\varphi(x) \implies F_T(x') \wedge x' \in B_T$ where x' is the projection of x to the variables in F_T.

3. F_E does not contain floating-point operations.

4. $\forall x \in B_T . F_T(x) \implies F_E(x)$

5. $\forall x \in B_T . F_T(x) \implies (F_E(x) \wedge F_{B_T}(x))$.

6. $\mathbf{b}_\mathrm{P} = B_T$.

7. $\forall x \in \mathbb{IR}^{*vars(\varphi)}.(\forall v \in \mathrm{intVars}.x_v \in \mathbb{Z}).F_E(x) \wedge F_{B_T}(x) \implies F_P(x') \wedge x' \in \mathbf{b}_\mathrm{P}$ where x' is the projection of x to the variables in F_P.

8. Since F_{B_T} is the `F` equivalent of $\mathbf{b}_\mathrm{P}$, we know that $\forall x \in \mathbb{IR}^{*vars(\varphi)}.(\forall v \in \mathrm{intVars}.x_v \in \mathbb{Z}).F_E(x) \implies F_P(x') \wedge x' \in \mathbf{b}_\mathrm{P}$ where x' is the projection of x to the variables in F_P

Thus, $\forall x \in \mathbb{IR}^{*vars(\varphi)}.(\forall v \in \mathrm{intVars}.x_v \in \mathbb{Z}).\varphi(x) \implies F_P(x') \wedge x' \in \mathbf{b}$ where x' is the projection of x to the variables in F_D. Since F_E does not contain floating-point operations, and DeriveBoundsAndFilter does not add any floating-point operations, we know that F_P does not contain any floating-point operations. Thus, PropaFP is sound. $\square$

4.3 Deriving Provable Error Bounds

We now describe how we derived the bound for the post-condition in the specification in Listing 4.2. The bound specifies the difference between `Taylor_Sin(X)` and the exact sine function. Note that the process we describe here is *not* part of the proving process and is not necessary for writing a specification such as the one in Listing 4.2. Rather, we present this process in order to aid the reader in understanding how such a bound can be broken down into its components and why it is difficult to reason about specifications with such bounds. The fact that this process described in this section is not precise nor fully automated does not affect the reliability and automation of the PropaFP proving process.

So, such a bound can be broken down as follows:

- The **subprogram specification error**, i.e. the error inherited from the specification of any subprograms that the implementation relies on.
  - If an implementation relies on some subprogram, the specification, not the implementation, of that subprogram would be used in the Why3 VC.
  - For `Taylor_Sin` this component is 0 as it does not call any subprograms.
- The **maximum model error** [9], i.e. the maximum difference between the *model* used in the computation and the exact *intended* result.
  - For `Taylor_Sin` this is the difference between the degree 3 Taylor polynomial for the sine function and the sine function.
- The **maximum rounding error** [9], i.e. the maximum difference between the *exact model* and the *rounded model* computed with FP arithmetic.
- A **rounding analysis cushion** arising when eliminating FP operations. This is the difference between the *actual maximum* rounding error and the *bound* on the rounding error calculated by a tool such as FPTaylor as well as over-approximations made when deriving bounds for variables.
  - The derived bounds are imperfect due to the accuracy loss of interval arithmetic as well as the over-approximation of FP operations.
  - Imperfect bounds on variables inflate the computed rounding error bound, as more values have to be considered.
- A **proving cushion** is added so that the specification can be decided by the approximation methods in the provers. Without this *cushion*, the provers could not decide the given specification within certain bounds on resources, such as a timeout.

	single precision	double precision
Subprogram Specification Error	0	0
Maximum Model Error	$\sim 2.59E{-4}$	$\sim 2.59E{-4}$
Maximum Rounding Error	$\sim 1.61E{-8}$	$\sim 2.89E{-17}$
Rounding Analysis Cushion	$\sim 1.57E{-9}$	$\sim 4.04E{-18}$
Proving Cushion	$\sim 2.11E{-9}$	$\sim 1.80E{-9}$

Table 4.2: Error bound components for `Taylor_Sin`

To justify our specification in Listing 4.2, we estimated the values of all five components. Our estimates can be seen in Table 4.2. The **maximum model error** and the **maximum rounding error** were calculated using the Monte-Carlo method. We ran a simulation comparing the Taylor series approximation of degree 3 of the sine function and an exact sine function. This simulation was ran for one million with pseudo-random inputs, giving us an approximate model error. To estimate the maximum rounding error, we compared a single precision and a quadruple precision FP implementation of the model for one hundred million pseudo-random inputs. (FP operations are much faster than exact real operations.) We estimate the **rounding analysis cushion** as the difference between the **rounding error** and the bound given by FPTaylor ($\sim 1.77E{-8}$). Note that the actual **rounding analysis cushion** may be larger due to over approximations made when deriving bounds.

The sum of the **maximum model error**, the **maximum rounding error**, and the **rounding analysis cushion** is around 0.0002588878950. Raising the specification bound to 0.00025889 enables provers LPPaver and dReal to verify the specification, using a **proving cushion** of around $2.11E{-9}$.

In this case, most of the error in the program comes from the **maximum model error**. If we increased the number of Taylor terms, the **maximum model error** would become smaller and the **maximum rounding error** would become larger. Increasing the input domain would make both the **maximum model error** and the **maximum rounding error** larger.

Increasing the precision of the FP numbers used is a simple way to reduce both the maximum **rounding error** and the **rounding analysis cushion.** Table 4.2 on the right shows estimates for the components in a double-precision version of `Taylor_Sin`[2].

To demonstrate how the **subprogram specification error** affects provable error bounds, consider function `SinSin` given in Listings 4.8 and 4.9.

Listing 4.8: SinSin function definition in SPARK

```
procedure Taylor_Sin_P (X : Float; R : out Float) is
begin
    R := X - ((X * X * X) / 6.0);
end Taylor_Sin_P;

function SinSin (X : Float) return Float is
    OneSin, TwoSin : Float;
begin
    Taylor_Sin_P(X, OneSin);
    Taylor_Sin_P(OneSin, TwoSin);
    return TwoSin;
end SinSin;
```

Listing 4.9: SinSin function specification in SPARK

```
procedure Taylor_Sin_P (X : Float; R : out Float) with
  Pre => X >= -0.5 and X <= 0.5,
  Post =>
    Rf(R) >= Ri(-48) / Ri(100) and -- Helps verification of
                                   -- calling functions
    Rf(R) <= Ri(48) / Ri(100) and
    abs(Real_Sin(Rf(X)) - Rf(R)) <=
                          Ri(25889) / Ri(100000000);

function SinSin ( X : Float) return Float with
  Pre => X >= -0.5 and X <= 0.5,
  Post =>
    abs(Real_Sin(Real_Sin(Rf(X))) - Rf(SinSin'Result))
     <= Ri(51778) / Ri(100000000);
```

`Taylor_Sin_P` is the procedure version of the `Taylor_Sin` function. Our implementation currently does not support function calls, but it does support

[2]The simplified exact NVC resulting from this example is referred to as Taylor_Sin_Double in Table 5.1.

procedure calls. (This limitation is not conceptually significant.) The specification for `Taylor_Sin_P` has two additional inequalities, bounding the output value `R` to allow us to derive tight bounds for `R` when proving VCs involving calls of this procedure. Verifying this procedure in GNATprove gives one NVC for our proving process, corresponding to the final post-condition. The exact NVC is in folder `examples/taylor_sine/txt` in the PropaFP code repository [54][3].

Function `SinSin` calls `Taylor_Sin_P` with the parameter `X`, storing the result in variable `OneSin`. `Taylor_Sin_P` is then called again with the parameter `OneSin`, storing the result in `TwoSin`, which is then returned. The post-condition for the `SinSin` function specifies the difference between its result and calling the exact $\sin(\sin(\mathtt{X}))$[4].

Since the steps of `SinSin` involve only subprogram calls, there is no **model error** or **rounding error**, and thus no **rounding analysis cushion**. As the value of `SinSin` comes from `Taylor_Sin_P` applied twice, and the derivative of $\sin$ has the maximum value 1, the **subprogram specification error** is a little below $0.00025889 + 0.00025889 = 0.00051778$. Experimenting with different bounds, we estimate the LPPaver **proving cushion** is around 10^{-13}.

There is a delicate trade-off between the five components that a programmer would need to manage by a careful choice of the model used, FP arithmetic tricks, and proof tools used to obtain a specification for a program that is both accurate and does not require large cushions or specification errors. It is not our goal to make this type of optimisation for the example programs, rather we have calculated these values to help improve the understanding of how difficult it is to estimate them in practice. In simple cases, it would be sufficient to tighten and loosen the 'bound' in the specification until the proving process fails and succeeds, respectively.

[3] This NVC is referred to as Taylor_Sin_P in Table 5.1.

[4] The NVC resulting from this post-condition is referred to as SinSin in Table 5.1.

Listing 4.10: Heron's Method Specification

```
function Certified_Heron (X : Float; N : Integer) Return Float with
  Pre => X >= 0.5 and X <= 2.0 and N >= 1 and N <= 5,
  Post =>
    abs(Real_Square_Root(Rf(X)) - Rf(Certified_Heron'Result))
      <= (Ri(1) / (Ri(2 ** (2 ** N)))) -- 1/2^(2^N) model error
             + Ri(3*N)*(Ri(1)/Ri(8388608)); -- 3 · N · ε, rounding error bound
```

Listing 4.11: Heron's Method Implementation

```
function Certified_Heron (X : Float; N : Integer) return Float is
  Y : Float := 1.0;
begin
  for i in 1 .. N loop
    Y := (Y + X/Y) / 2.0;

    pragma Loop_Invariant (Y >= 0.7);
    pragma Loop_Invariant (Y <= 1.8);
    pragma Loop_Invariant
      (abs (Real_Square_Root (Rf(X)) - Rf(Y))
        <= (Ri(1) / (Ri(2 ** (2 ** i)))) -- 1/2^(2^i)
             + Ri(3*i)*(Ri(1)/Ri(8388608))); -- 3 · i · ε
  end loop;
  return Y;
end Certified_Heron;
```

4.4 Verifying Heron's Method for Approximating the Square Root Function

We used PropaFP to verify an implementation of Heron's method. This is an interesting case study because it requires the use of loops and loop invariants.

In Listing 4.10, the term $3 \cdot \mathtt{N} \cdot \varepsilon$ is a heuristic bound for the compound rounding error, guessed by counting the number of operations. Note that five iterations are more than enough to get an accurate approximation of the square root function for `X` in the range $[0.5, 2]$.

The implementation in Listing 4.11 contains loop invariants. The bounds on Y here help generate easier VCs for the loop iterations and post-loop behaviour. The main loop invariant is very similar to the post-condition in the specification, except substituting `i` for `N`, essentially specifying the difference between the exact square root and Heron's method for each iteration of the loop.

Why3 produces 74 NVCs from our implementation of Heron's method. 72 of these NVCs are either trivial or verified by SMT solvers. PropaFP is required for 2 NVCs that come from the main loop invariant. One NVC specifies that the loop invariant holds in the initial iteration of the loop, where `i` is equal to 1. Another VC specifies that the loop invariant is preserved from one iteration to the next, where `i` ranges from 1 to `N`[5]. Note that the third NVC derived from the invariant, i.e., that the invariant on the last iteration implies the postcondition, is trivial here. The corresponding simplified exact NVCs can be found in folder `examples/heron/txt` in the PropaFP repository.

4.5 Verifying AdaCore's Sine Implementation

With the help of PropaFP, we have developed a verified version of an Ada sine implementation written by AdaCore for their high-integrity mathematics library[6]. First, we removed SPARK-violating code such as generic FP types, fixing the type to the single-precision `Float`. We then translated functions into procedures since PropaFP currently does not support function calls.

The code consists of several dependent subprograms. There are functions for computing $\sin(x)$ and $\cos(x)$ for x close to 0 and functions that extend the domain to $x \in [-802, 802]$ by translating x into one of the four basic quadrants near 0. There is also a loop that extends the domain further. We have focused on the code for $x \in [-802, 802]$ and postponed the verification of the loop.

We have translated functions into procedures since PropaFP currently does not support function calls. Next, we discuss all six procedures that we needed to specify and verify.

Listing 4.12: Multiply_Add Implementation

```
procedure Multiply_Add
  (X, Y, Z : Float; Result : out Float) is
begin
    Result := (X * Y + Z);
end Multiply_Add;
```

Listing 4.13: Multiply_Add Specification

```
procedure Multiply_Add
  (X, Y, Z : Float; Result : out Float) with
  Pre =>
    (-3.0 <= X and X <= 3.0) and
    (-3.0 <= Y and Y <= 3.0) and
    (-3.0 <= Z and Z <= 3.0),
  Post =>
    (-12.0 <= Result and Result <= 12.0) and
    Result = X * Y + Z;
```

4.5.1 Multiply_Add

The specification in Listing 4.13 restricts the ranges of the input and output to rule out overflows. We used very small bounds based on how the function is used locally by the other procedures.

4.5.2 My_Machine_Rounding

This is a custom procedure that is used to round a FP number to the nearest integer. In the original version of this code, this was done using the SPARK-violating Ada function, `Float'Machine_Rounding`.

Again, we specify the ranges of the variables based on the local use of this procedure, to make it easier for our provers to verify the resulting VCs.

The other post-conditions state that the difference between `X` and `Y` (which is `X` rounded to the nearest integer) is, at most, 0.500000001[7]. We

[5]We refer to these NVCs as Heron_Init and Heron_Pres in Table 5.1.

[6]We obtained the original code from file `src/ada/hie/s-libsin.adb` in archive `gnat-2021-20210519-19A70-src.tar.gz` downloaded from "More packages, platforms, versions and sources" at https://www.adacore.com/download.

[7]The NVCs resulting from the last two post-conditions are referred to as My_Machine_Rounding$_{\geq}$ and My_Machine_Rounding$_{\leq}$ in Table 5.1.

Listing 4.14: My_Machine_Rounding Implementation

```
procedure My_Machine_Rounding
  (X : Float; Y : out Integer) is
begin
    Y := Integer(X); -- rounding to nearest
end My_Machine_Rounding;
```

Listing 4.15: My_Machine_Rounding Specification

```
procedure My_Machine_Rounding
  (X : Float; Y : out Integer) with
  Pre =>
    (0.0 <= X and X <= 511.0),
  Post =>
    (0 <= Y and Y <= 511) and
    Rf(X) - Ri(Y) >= Ri(-500000001) / Ri(1000000000) and
                                  -- -0.500000001
    Rf(X) - Ri(Y) <= Ri(500000001) / Ri(1000000000);
                                  -- 0.500000001
```

chose this number to avoid any "touching" VCs (such as $x > 0 \implies x > 0$), which solvers using interval methods usually cannot prove. While SMT solvers can usually verify simple touching VCs, here they fail, probably due to the rounding function.

4.5.3 Reduce_Half_Pi

This procedure takes some input value, `X`, and subtracts a multiple of $\frac{\pi}{2}$ to translate it into the interval $[-0.26 \otimes \pi_{fp}, 0.26 \otimes \pi_{fp}]$.

The implementation, seen in Listing 4.16, has some significant differences to the original implementation. First, we limited this procedure to `X` within $[0, 802]$ and removed a loop that catered for larger values, as mentioned earlier. Also, we inlined calls to `Float'Leading_Part`, a SPARK-violating function which removes a specified number of bits from a FP number. This function was used to define the variables `C1`, `C2`, and `C3`, in effect, giving a higher precision version of $\pi/2$ using single-precision FP variables.

The specification in Listing 4.17 includes a new out parameter `R`, which was just a local variable in the original implementation. `R` holds the integer

Listing 4.16: Reduce_Half_Pi Implementation

```
procedure Reduce_Half_Pi
  (X : in out Float; Q : out Quadrant; R : out Integer)
is
  K       : constant       := Pi / 2.0;
  -- Bits_N : constant := 9;
  -- Bits_C : constant := Float'Machine_Mantissa - Bits_N;
  C1      : constant Float := 1.57073974609375;
                               -- Float'Leading_Part (K, Bits_C);
  C2      : constant Float := 0.0000565797090530395508;
                               -- Float'Leading_Part (K - C1, Bits_C);
  C3      : constant Float := 0.0000000000992088189377682284;
                               -- Float'Leading_Part (K - C1 - C2, Bits_C);
  C4      : constant Float := K - C1 - C2 - C3;
  N       : Float := (X / K);
begin
  My_Machine_Rounding(N, R);    -- R is returned for use in the specification

  X :=
    (((X - Float(R)*C1) - Float(R)*C2) - Float(R)*C3) - Float(R)*C4;
  -- The above is roughly equivalent to X := (X - Float(R)*K);
  Q := R mod 4;
end Reduce_Half_Pi;
```

Listing 4.17: Reduce_Half_Pi Specification

```
subtype Quadrant is Integer range 0 .. 3;

Max_Red_Trig_Arg : constant := 0.26 * Ada.Numerics.Pi;
Half_Pi          : constant := Ada.Numerics.Pi / 2.0;

procedure Reduce_Half_Pi
  (X : in out Float; Q : out Quadrant; R : out Integer)
  with Pre =>  X >= 0.0 and X <= 802.0,
  Post =>
    R >= 0 and R <= 511 and
    Rf(X'Old / (Pi/2.0)) - Ri(R) >= Ri(-500000001)/Ri(1000000000)
    and
    Rf(X'Old / (Pi/2.0)) - Ri(R) <= Ri(500000001)/Ri(1000000000)
    and
    Q = R mod 4 and
    X >= -Max_Red_Trig_Arg and X <= Max_Red_Trig_Arg and
    (Rf(X) - (Rf(X'Old) - (Ri(R)*Real_Pi/Rf(2.0)))) >=
      Ri(-18)/Ri(100000)
    and
    (Rf(X) - (Rf(X'Old) - (Ri(R)*Real_Pi/Rf(2.0)))) <=
      Ri(18)/Ri(100000);
```

multiple of $\frac{\pi}{2}$ used to shift the input value close to 0. The final two post-conditions bound the difference between the computed new value of `X` and the ideal model result. Our proving process is needed for the NVCs derived from the last four post-conditions in Listing 4.17[8].

4.5.4 Approx_Sin and Approx_Cos

Approx_Sin and Approx_Cos in Listing 4.18 compute Taylor series approximations of sine and cosine, respectively, using the Horner scheme. In the original AdaCore implementation, variable `X` has a generic type, but we have fixed the type to `Float`. The original implementation uses arrays and loops to adapt the order of the Taylor series to the precision of the float type. Since we have fixed the type of `X`, we perform these computations directly without arrays and loops.

The specifications in Listing 4.19 are quite simple. The preconditions restrict the value of `X` to be within the interval $[-0.26 \otimes \pi_{fp}, 0.26 \otimes \pi_{fp}]$. The first two post-conditions in both procedures restrict the `Result` to be within the interval $[-1, 1]$. The last two post-conditions in both procedures specify the difference between the exact Sine/Cosine and Approx_Sin/Approx_Cos[9].

4.5.5 Sin

Finally, procedure Sin in Listing 4.20 approximates the sine function for inputs from $[-802, 802]$. Compared to the original function, we have replaced uses of the SPARK-violating function `Float'Copy_Sign` with code that has the same effect. Our proving process is needed to verify NVCs arising from the final two post-conditions in Listing 4.21[10]

[8] The resulting NVCs are referred to in Table 5.1 as Reduce_Half_Pi_X$\{\geq, \leq\}$ and Reduce_Half_Pi$\{\geq, \leq\}$, respectively.

[9] The NVCs corresponding to the last two postconditions in both procedures are called Approx_Sin$\{\geq, \leq\}$ and Approx_Cos$\{\geq, \leq\}$ in Table 5.1.

[10] The resulting NVCs are referred to as Sin$_{\{\geq, \leq\}}$ in Table 5.1.

Listing 4.18: Approx_Sin and Approx_Cos Implementation

```
procedure Approx_Sin  (X : Float; Result : out Float) is
  Sqrt_Epsilon_LF : constant Long_Float :=
    Sqrt_2 ** (1 - Long_Float'Machine_Mantissa);

  G  : constant Float := X * X;

  -- Horner Scheme
  H0 : constant Float := (-0.19501_81843E-3);
  H1 : Float;
  H2 : Float;
begin
  Multiply_Add(H0, G, (0.83320_16396E-2), H1);
  Multiply_Add(H1, G, (-0.16666_65022), H2);
  if abs X <= Float(Long_Float (Sqrt_Epsilon_LF)) then
     Result := X;
  else
     Result := (X * (H2 * G) + X);
  end if;
end Approx_Sin;

procedure Approx_Cos (X : Float; Result : out Float) is
  G  : constant Float := X * X;

  -- Horner Scheme
  H0 : constant Float := (0.24372_67909E-4);
  H1 : Float;
  H2 : Float;
  H3 : Float;
  H4 : Float;
begin
  Multiply_Add(H0, G, (-0.13888_52915E-2), H1);
  Multiply_Add(H1, G, (0.41666_61323E-1), H2);
  Multiply_Add(H2, G, (-0.49999_99957), H3);
  Multiply_Add(H3, G, (0.99999_99999), H4);
  Result := H4;
end Approx_Cos;
```

Listing 4.19: Approx_Sin and Approx_Cos Specification

```
Max_Red_Trig_Arg : constant := 0.26 * Ada.Numerics.Pi;
Sqrt_2 : constant :=
  1.41421_35623_73095_04880_16887_24209_69807_85696;

procedure Approx_Sin  (X : Float; Result : out Float) with
  Pre  =>
    X >= -Max_Red_Trig_Arg and X <= Max_Red_Trig_Arg,
  Post =>
    Result >= -1.0 and Result <= 1.0 and
    (Rf(Result) - Real_Sin(Rf(X))) >= Ri(-58) / Ri(1000000000) and
    (Rf(Result) - Real_Sin(Rf(X))) <= Ri(58) / Ri(1000000000);

procedure Approx_Cos (X : Float; Result : out Float) with
  Pre  =>
    X >= -Max_Red_Trig_Arg and X <= Max_Red_Trig_Arg,
  Post =>
    Result >= -1.0 and Result <= 1.0 and
    (Rf(Result) - Real_Cos(Rf(X))) >= Ri(-14) / Ri(100000000) and
    (Rf(Result) - Real_Cos(Rf(X))) <= Ri(14) / Ri(100000000);
```

Listing 4.20: Sin Implementation

```
procedure Sin (X : Float; FinalResult : out Float) is
  Y      : Float := (if X < 0.0 then -X else X);
  Q      : Quadrant;
  R      : Integer;
  Result : Float;
begin
  Reduce_Half_Pi (Y, Q, R);

  if Q = 0 or Q = 2 then
     Approx_Sin (Y, Result);
  else -- Q = 1 or Q = 3
     Approx_Cos (Y, Result);
  end if;

  if X < 0.0 then
     FinalResult := (-1.0) * (if Q >= 2 then -Result else Result);
  else
     FinalResult := (1.0)  * (if Q >= 2 then -Result else Result);
  end if;
end Sin;
```

Listing 4.21: Sin Specification

```
procedure Sin (X : Float; FinalResult : out Float)
  with Pre =>
    X >= -802.0 and X <= 802.0,
  Post =>
    (Rf(FinalResult) - Real_Sin(Rf(X))) >= Ri(-19) / Ri(100000) and
    (Rf(FinalResult) - Real_Sin(Rf(X))) <= Ri(19) / Ri(100000);
```

Table 4.3: Why3 NVCs Generated for each Procedure from our Modified AdaCore Sine Implementation

Procedure	Generated NVCs	Trivial/SMT	Proving Process
Multiply_Add	4	4	0
My_Machine_Rounding	16	14	2
Reduce_Half_Pi	44	40	4
Approx_Sin	33	31	2
Approx_Cos	41	39	2
Sin	20	18	2

4.5.6 Generated Why3 NVCs

In total, Why3 derived 158 NVCs from the six procedures we have described. SMT solvers verified 146 NVCs. The 12 remaining NVCs were verified using our proving process. This is broken down by procedure in Table 4.3.

We discuss only a few of the more interesting NVCs here. All NVCs can be found in folder `examples/hie_sine/txt` in the PropaFP code repository.

Listing 4.22 shows two of the simplified exact NVCs arising from the post-conditions in Reduce_Half_Pi. In both NVCs, the second and third assertions come from the third and fourth post-conditions and define how the `x` and `r1` variables are dependent on each other. In both NVCs, the final assertion comes from the post-condition used to derive the NVC. The final assertion in the first NVC asserts an upper bound on the new value of `X` after calling Reduce_Half_Pi. The final assertion in the second NVC asserts that the difference between the new value of `X` after calling Reduce_Half_Pi and performing the same number of $\pi/2$ reductions on the original value of `X` using the exact π is *not* smaller than or equal to $18/100000$.

The exact NVC in Listing 4.23 comes from the final post-condition from the Approx_Sin procedure in Listing 4.19. The first two assertions specify a dependency on `x` and `result__1`. There are two assertions here due to the two if-then-else branches in the implementation of Approx_Sin in Listing 4.18. The final assertion specifies that the difference between the result of Approx_Sin for `x` and the value of the exact sine function for `x` is not smaller than or equal to $^{58}/_{1000000000}$.

Finally, the NVC in Listing 4.24 comes from the first post-condition in the procedure Sin in Listing 4.20.

This NVC is interesting since the implementation of the Sin procedure depends on the other procedures we have discussed, which results in the derived NVC including assertions derived from specifications of these other procedures. Assertions 1–2 come from the if statement defining `Y`. Assertions 3–6 come from the Reduce_Half_Pi post-conditions as a consequence of calling Reduce_Half_Pi in Listing 4.20. Assertion 7 comes from the `Quadrant` subtype defined in Listing 4.17 Assertions 8–9 contain the final two Approx_Sin/Approx_Cos post-conditions as well as corresponding to one of the if-then-else branches after the call to Reduce_Half_Pi. Assertions 10–11 correspond to the different paths from the final if-then-else. The final assertion comes from the first post-condition in Listing 4.20.

Listing 4.22: Selected Reduce_Half_Pi Simplified Exact NVCs

```
Reduce_Half_Pi_X_≤

Bounds on variables:
r1 (int) ∈ [0, 511]
x (real) ∈ [0, 802]

assert
  -500000001 / 1000000000 <=
    (((x / (13176795/8388608)) - r1) +
      (433681/2000000000000000000000000))

assert
  (((x / (13176795/8388608)) - r1) -
    (433681/2000000000000000000000000)) <=
      500000001 / 1000000000

assert
  ¬(
    ((((x - (r1 * (25735/16384))) - (r1 * (3797/67108864)))
      - (r1 * (17453/17592186044416)))
    - (r1 * (12727493/2361183241434822606848)))
    + (1765573/10000000000)
    <= 6851933/8388608
  )


Reduce_Half_Pi≤

Bounds on variables:
r1 (int) ∈ [0, 511]
x (real) ∈ [0, 802]

assert
  -500000001 / 1000000000 <=
    (((x / (13176795/8388608)) - r1) +
      (433681/2000000000000000000000000))

assert
  (((x / (13176795/8388608)) - r1) -
    (433681/2000000000000000000000000)) <=
      500000001 / 1000000000

assert
  ¬(
    ((((((x - (r1 * (25735/16384))) - (r1 * (3797/67108864)))
        - (r1 * (17453 / 17592186044416)))
       - (r1 * (12727493 / 2361183241434822606848)))
      - (x - ((r1 * π) / 2)))
     + (/ 1765573 10000000000))
    <= 18 / 100000
  )
```

Listing 4.23: Selected Approx_Sin NVC

```
Approx_Sin≤

Bounds on variables:
result__1 (real) ∈ [-7639663/8388608, 3819831/4194304]
                        -- ~ [−0.91072, 0.91072]
x (real) ∈ [-6851933/8388608, 6851933/8388608]
                        -- ~ [−0.81681, 0.81681]

NVC:
assert
  (abs(x) <= 1 / 67108864  ⟹  (x = result__1))

assert
  ¬ (abs(x) <= 1 / 67108864)  ⟹
    (((x*((((((-3350387 / 17179869184)*(x*x)) +
      (4473217 / 536870912))*(x*x)) -
        (349525 / 2097152))*(x*x))) + x) -
    (4498891 / 100000000000000))
    <= result__1
    ∧
    result__1 <=
    (((x*((((((-3350387 / 17179869184)*(x*x)) +
      (4473217 / 536870912))*(x*x)) -
        (349525 / 2097152))*(x*x))) + x) +
    (4498891 / 100000000000000))

assert
  ¬( result__1 - sin(x) <= 58 / 1000000000 )
```

Listing 4.24: Selected Sin NVC

```
Sin≥

Bounds on variables:
finalresult1 (real) ∈ [-1, 1]
o (real) ∈ [-802, 802]
r1 (int) ∈ [0, 511]
result__1 (real) ∈ [-1, 1]
x (real) ∈ [-802, 802]
y (real) ∈ [-6851933/8388608, 6851933/8388608]
          -- 6851933/8388608 = Max_Red_Trig_Arg - 0.26 * pi

NVC:
assert-1 x < 0.0    -> o = -x
assert-2 ¬(x < 0.0) -> o = x
assert-3 -500000001 / 1000000000 <=
  ((o / (13176795 / 8388608)) - r1) +
    (433681 / 2000000000000000000000000)
assert-4 ((o / (13176795 / 8388608)) - r1) -
  (433681 / 2000000000000000000000000) <=
    500000001 / 1000000000
assert-5 -18.0 / 100000.0 <= (y + (o + (r1 * Pi / 2.0)))
assert-6 (y + (o + (r1 * Pi / 2.0))) <= 18.0 / 100000.0
assert-7 mod r1 4 <= 3.0
assert-8
  (mod r1 4 <= 0.0) ∨ (¬(mod r1 4 <= 0.0) ∧ (mod r1 4 == 2.0)) ->
    -58.0 / 1000000000 <= result__1 - (sin y) ∧
    result__1 - (sin y) <= 58.0 / 1000000000
assert-9
  ¬(¬(mod r1 4 <= 0.0) -> mod r1 4 = 2.0) ->
    -14.0 / 100000000 <= result__1 - (cos y) ∧
    result__1 - (cos y) <= 14.0 / 100000000
assert-10
  x < 0 ->
    mod r1 4 <= 2 -> finalresult1 = result__1 ∧
    ¬(mod r1 4 <= 2) -> finalresult1 = -result__1
assert-11
  ¬(x < 0) ->
    mod r1 4 <= 2 -> finalresult1 = -result__1 ∧
    ¬(mod r1 4 <= 2) -> finalresult1 = result__1
assert-12 ¬(-19 / 100000 <= (finalresult1 - (sin x)))
```

Chapter 5

Evaluation

In this chapter, we evaluate both PropaFP and LPPaver. We start by evaluating the PropaFP proving process in Section 5.1. We then evaluate LPPaver in Section 5.2

Hardware All benchmarks in this chapter were ran on the same machine which is using Ubuntu 20.04. The machine has a Ryzen 5 3600 CPU, 16GB of RAM running at 3600MHz/CL16, and a 1TB NVME SSD.

5.1 Evaluation of PropaFP

In this section, we evaluate PropaFP using the examples we described in Chapter 4.

5.1.1 Benchmarking the PropaFP Proving Process

Tables 5.1 shows the performance of our implementation of the proving process on the verification examples described earlier. "VC processing" is the time it takes PropaFP v0.1.2.0 to process the NVCs generated by GNATprove for Why3 v1.4.0, including calls to FPTaylor. The remaining columns in Table 5.1 show the performance on the following provers applied to the resulting simplified exact NVCs:

Table 5.1: Proving Process on Described Examples

VC	VC Processing	dReal	MetiTarski	LPPaver
My_Machine_Rounding$_{\geq}$	0.05s	n/s	n/s	0.55s
My_Machine_Rounding$_{\leq}$	0.06s	n/s	n/s	0.47s
Reduce_Half_Pi_X$_{\geq}$	0.10s	n/s	0.12s	0.36s
Reduce_Half_Pi_X$_{\leq}$	0.10s	n/s	0.07s	0.34s
Reduce_Half_Pi$_{\geq}$	0.10s	n/s	g/u	0.02s
Reduce_Half_Pi$_{\leq}$	0.10s	n/s	g/u	0.02s
Approx_Sin$_{\geq}$	0.14s	1m03s	0.30s	5.67s
Approx_Sin$_{\leq}$	0.12s	1m04s	0.26s	5.65s
Approx_Cos$_{\geq}$	0.09s	3.24s	0.05s	1.83s
Approx_Cos$_{\leq}$	0.11s	1.48s	0.05s	1.52s
Sin$_{\geq}$	0.10s	n/s	n/s	6m01s
Sin$_{\leq}$	0.11s	n/s	n/s	6m03s
Taylor_Sin	0.11s	0.01s	0.17s	0.06s
Taylor_Sin_Double	0.15s	n/s	0.16s	0.06s
Taylor_Sin_P	0.10s	0.01s	0.17s	0.07s
SinSin	0.07s	3m20s	g/u	8.30s
Heron_Init	0.16s	0.00s	0.09s	0.02s
Heron_Pres	0.16s	5m05s	g/u	1m20s

- dReal v4.21.06.2 [32] (see Sections 2.7 and 5.2 for more details).
- MetiTarski v2.4 [1] (see Sections 2.7 and 5.2 for more details).
- LPPaver v0.0.5.0 [53] – our prover described in Chapter 3.

These provers were chosen because most of the problems in this set of benchmarks contain transcendental operations which these provers support.

In Table 5.1, n/s means the NVC contains some operation or number that is not supported by the prover (e.g., dReal does not support very large integers) while g/u means that the prover gave up.

All of the NVCs were solved by at least one of the provers in a reasonable time frame. VC processing takes only a fraction of a second for all of the NVCs.

Some of the NVCs could be decided by only LPPaver due to the following:

- The My_Machine_Rounding NVC contains integer rounding with ties going away from zero.
  - dReal does not support integer rounding.
  - MetiTarski does not support the rounding mode specified in this NVC.
- After our proving process, the bound on the **maximum rounding error** computed by FPTaylor in both the Reduce_Half_Pi and the Taylor_Sin_Double NVCs are very small.
  - This number is represented as a rational number in the exact NVC, and the denominator is outside the range of integers supported by dReal.
- The Reduce_Half_Pi$\{_{\geq}, _{\leq}\}$ NVCs have a tight bound.
  - Slightly loosening the bound from 0.00018 to 0.0002 allows MetiTarski to verify this.
    - After this loosening, the Sin$\{_{\geq}, _{\leq}\}$ would need to be loosened from 0.00019 to 0.00021 due to the increased **subprogram specification error** (see Section 4.3).
- The Sin NVCs contain integer rounding with ties going to the nearest even integer and uses the modulus operator.
  - dReal does not support integer rounding.
  - MetiTarski does not support the modulus operator.

Effect of Specification Bounds on Proving Times

For numerical provers, the tightness of the specification bound is often correlated with the time it takes for a prover to decide a VC arising from said specification. This is not normally the case for symbolic solvers, however, a VC arising from a specification that a symbolic solver could not decide may become decidable with a looser bound on the specification. We illustrate

Table 5.2: Effect of Specification Bound on Proving Time

VC	Bound	VC Processing	dReal	MetiTarski	LPPaver
Approx_Sin$_{\geq}$	-0.000000058	0.14s	1m03s	0.30s	5.67s
Approx_Sin$_{\geq}$	-0.000000075	0.15s	26.74s	0.28s	3.78s
Approx_Sin$_{\geq}$	-0.0000001	0.13s	14.74s	0.29s	2.80s
Approx_Sin$_{\geq}$	-0.00001	0.14s	0.09s	0.28s	0.25s
Approx_Sin$_{\leq}$	0.000000058	0.15s	1m04s	0.26s	5.65s
Approx_Sin$_{\leq}$	0.000000075	0.12s	27.56s	0.27s	3.79s
Approx_Sin$_{\leq}$	0.0000001	0.15s	15.04s	0.26s	2.76s
Approx_Sin$_{\leq}$	0.00001	0.14s	0.09s	0.26s	0.27s
Approx_Cos$_{\geq}$	-0.00000014	0.09s	3.24s	0.05s	1.83s
Approx_Cos$_{\geq}$	-0.0000005	0.08s	0.31s	0.05s	0.62s
Approx_Cos$_{\geq}$	-0.000001	0.08s	0.14s	0.05s	0.52s
Approx_Cos$_{\geq}$	-0.0001	0.11s	0.00s	0.05s	0.09s
Approx_Cos$_{\leq}$	0.00000014	0.09s	1.48s	0.05s	1.52s
Approx_Cos$_{\leq}$	0.0000005	0.07s	0.29s	0.05s	0.61s
Approx_Cos$_{\leq}$	0.000001	0.07s	0.13s	0.04s	0.49s
Approx_Cos$_{\leq}$	0.0001	0.10s	0.00s	0.04s	0.07s
SinSin	0.00051778	0.07s	3m20s	g/u	8.30s
SinSin	0.00052	0.07s	0.13s	g/u	5.36s
SinSin	0.001	0.07s	0.02s	g/u	1.36s
SinSin	0.01	0.06s	0.00s	g/u	0.33s

this in Table 5.2. The ‘Bound’ column states the specification bound for the NVC.

Table 5.2 shows how, in all of our examples, a looser bound results in quicker proving times for the tested numerical provers. In some cases, this improvement can be significant, as seen with the ‘SinSin’ NVCs. The proving time for symbolic provers does not improve with looser bounds. However, MetiTarski failed to decide Reduce_Half_Pi{$_{\geq,\leq}$}, but it could decide these NVCs when the specification bounds were loosened from $1.8E{-4}$ to $2.0E{-4}$.

Counter-examples

When writing specifications, it is not uncommon for a programmer to make a mistake in the specification by, for example, using wrong mathematical operations, setting too tight a bound for a specification, and so on. When this occurs, it would be useful for a programmer to receive a counter-example for their specification.

Our proving process supports producing counter-examples and, with a custom Why3 driver, these counter-examples can be reported back to Why3, which will send the counter-examples to the programmer's IDE. It should be understood that counter-examples produced by PropaFP are *potential* counter-examples [17], since 'simplified exact' NVCs are weakened versions of original NVCs. Nevertheless, these *potential* counter-examples can still be *actual* counter-examples and would be useful for a programmer to have.

To demonstrate how the proving process can produce counter-examples, we modify our `Taylor_Sin` example, introducing three different mistakes which a programmer may feasibly make:

1. Replace the `-` with `+` in the `Taylor_Sin` implementation in Listing 4.1.
2. Invert the inequality in the `Taylor_Sin` post-condition in Listing 4.2.
3. Make our specification bound slightly tighter than the **maximum model error** + **maximum rounding error** + **rounding analysis cushion** in the post-condition from Listing 4.2, changing the value of the right-hand side of the inequality in the post-condition from 0.00025889 to 0.00025887.

These three 'mistakes' are referred to as Taylor_Sin_Plus, Taylor_Sin_Swap, and Taylor_Sin_Tight, respectively, in Table 5.3.

If a specification is incorrect, the resulting NVC must be true or 'sat'. Recall that dReal would report a 'δ-sat' result, which means the δ-weakening of the NVC was 'sat'. In all of our examples, δ is equal to 1×10^{-100}. This makes models produced by dReal a *potential* model for the NVC. Models produced by LPPaver are actual models for the given NVC, but for files

Table 5.3: Proving Process on Described Counter-examples

VC	VC Processing	dReal	CE	LPPaver	CE
Taylor_Sin_Plus	0.12s	0.00s	$x = -0.166\ldots$	0.02s	$x = -0.5$
Taylor_Sin_Swap	0.10s	0.00s	$x = 0.333\ldots$	0.03s	$x = 0.499\ldots$
Taylor_Sin_Tight	0.12s	0.00s	$x = 0.499\ldots$	0.03s	$x = 0.499\ldots$

produced by the proving process, these should still be thought of as *potential* counter-examples due to the weakening of the NVC. The computed potential counter-examples shown in Table 5.3 are all actual counter-examples except those for Taylor_Sin_Tight.

Reflections

The evaluation we present in this Section demonstrates how PropaFP coupled with LPPaver builds on the state-of-the-art techniques used to formally verify specifications of FP programs. Current techniques were not able to prove or disprove any of the (Why3) NVCs shown in Tables 5.1, 5.2, and 5.3.

Using techniques described in Chapter 4, we simplified, derived bounds for variables, and removed FP operations from the Why3 NVCs. The resulting NVCs, which we call 'exact real NVCs', are weakened versions of the NVCs they are based on.

The 'exact real NVCs' are passed to powerful automated provers for nonlinear real formulas. At least one of the provers are able to verify all of the problems we present in a reasonable time frame.

Three of the 'exact real NVCs' are incorrect and PropaFP with LPPaver produce for the original program 'potential counter-examples'. Two of these 'potential counter-examples' are also valid counter-examples; that is, the produced counter-example gives us input for which the original specification is falsified. The third 'potential counter-example' was not a valid counter-example, though this makes sense as the specification was very close to the 'boundary' where it would become true and the weakening of the NVC in these cases makes it more likely for the potential counter-examples produced by PropaFP to not be ‘actual’ counter-examples.

5.2 Evaluation of LPPaver

We evaluate the strength and efficiency of LPPaver v0.0.5.0 alongside the following selected automated solvers for nonlinear real formulas:

- dReal v4.21.06.2 - An automated numerical prover with good support for nonlinear real functions. dReal combines SMT methods with interval methods including interval constraint propagation and a branch-and-prune algorithm.
- ksmt v0.1.7 - An automated SMT solver for quantifier-free nonlinear real arithmetic. ksmt combines a CDCL-style algorithm with linearisations of nonlinear real terms.
- MetiTarski v2.4 - An automatic theorem prover with support for nonlinear real arithmetic. MetiTarski supports the theory of real closed fields and uses the Z3 solver as a backend which implements the DPLL(T) algorithms alongside simplex-based linear arithmetic solving techniques.
- Colibri v0.3.3 - An automatic solver for nonlinear real and FP arithmetic. Colibri uses Constraint Programming techniques including linearisations and the simplex method.

More detail on each of these solvers can be found in Section 2.7.

In our evaluation, we use two sets of benchmarks. The first set of benchmarks is based on the exact real NVCs (negated verification conditions) produced by PropaFP from specifications of FP programs as described in Section 5.1. The second set is a variation of the sphere packing problem, a type of optimisation problem, and is described in Section 5.2.2.

5.2.1 Performance of Provers on PropaFP Examples

We now discuss the performance of the provers, particularly LPPaver, for the examples in Table 5.1. We may ignore the 'VC Processing' column as that relates to PropaFP. All problems in this table are unsatisfiable. n/s in this table means that the prover did not support some function or feature that was present in the problem. The unsupported functions used may include integer rounding or use of extremely large numbers, and is explained in detail in Section 5.1.1. g/u means that the prover gave up.

ksmt does not support any of the problems here, either due to use of division or transcendental functions such as the sine or square root functions which ksmt does not support. Colibri similarly does not support most of the problems we present here, but Colibri does support division and thus can be ran on Reduce_Half_Pi$\{_{\geq},_{\leq}\}$ where it performed better than the other provers, giving an 'unsat' result in 0.02 seconds.

LPPaver performed well on these examples and proved that all of them are unsatisfiable within a reasonable time frame. LPPaver performed better than dReal here in all but one example, namely Heron_Init, where the difference between dReal and LPPaver is negligible. In some cases, MetiTarski performed better than LPPaver; for example, MetiTarski decided that Approx_Sin_$\geq$ is unsatisfiable in 0.3 seconds, better than the 5.67 seconds it took LPPaver. This is much better than the 1 minute and 3 seconds it took dReal to prove that the same problem is unsatisfiable. There is a similar pattern in some of the other problems in this table, where dReal took significantly longer to produce a result than LPPaver and MetiTarski.

How the Numerical Difficulty of a Problem affects Provers

The problems with a $\geq/\leq$ suffix specify the difference between the implementation of the named function and the exact function that the implementation is attempting to approximate. For example Approx_Sin$\{_{\geq},_{\leq}\}$ specify the difference between the approximation of the sine function implemented by Approx_Sin and the exact sine function. The 'bound' here, that is the maximum difference between the approximation and the exact sine function, is very 'tight' or very small. This makes the problem numerically challenging.

'Loosening' the bound would make the problem numerically easier, which should theoretically help the proving times of the numerical solvers, but it should not significantly help the proving times of solvers that mainly use symbolical methods. However, if a symbolical solver is not able to prove some problem with a tight bound, the solver may be able to prove the same VC if the bound is loosened. This is demonstrated with the Reduce_Half_Pi$\{_{\geq}, _{\leq}\}$ problems where loosening the bound by 0.00002 allowed MetiTarski to verify these problems.

Table 5.2 shows how loosening the bounds impacted our solvers. As expected, loosening the bound helped the performance of dReal and LP-Paver but did not speed up proving times for MetiTarski. As the bound loosened, proving times for both dReal and LPPaver improves. With very loose bounds, dReal was able to prove the problems (slightly) faster than LPPaver.

Table 5.3 show models produced by dReal and LPPaver for satisfiable VCs. Both provers produce the models more-or-less instantly, and models produced by both provers are correct for all three problems.

Reflections

LPPaver performed extremely well on the problems presented in 5.1. This implies that the 'proving' algorithm (i.e. Algorithm 5) is very efficient for these types of problems, especially in comparison to dReal which uses similar methods to LPPaver. The performance of the 'model search' algorithm (i.e. Algorithm 8) has also been found to be comparable with other solvers.

Table 5.2 shows how the performance of LPPaver improves when the problem becomes easier numerically. dReal had a similar performance increase and performed slightly better than other solvers with some of the looser bounds that we tested. This may be due to LPPaver being implemented in a higher-level language (Haskell) than dReal (C++).

Table 5.3 shows how LPPaver can quickly produce models useful for satisfiable problems. Both dReal and LPPaver produced models very quickly.

5.2.2 Placing Spheres of an Equal Size in a Cube

We further evaluate LPPaver and other solvers on a set of problems where the solver must determine a valid placement for a fixed number of spheres of radius 1 inside a cube of height 4. The spheres are allowed to 'touch' the faces of the cube but cannot be outside the cube. The spheres are not allowed to 'touch' each other. We also generalise this into two and four dimensions, where the problem becomes placing circles in a square and 3-spheres in a 3-cube respectively.[1] For a dimension $d \geq 2$ and for n number of $(d-1)$-spheres where $n > 0$, we can generalise the problem as follows:

$$\begin{aligned} \exists \mathbf{c^{(1)}}, \ldots, \mathbf{c^{(n)}} \in \mathbf{R}^d : \\ \bigwedge_{1 \leq i \leq n} \|\mathbf{c^{(i)}}\|_\infty \leq 1 \qquad \wedge \\ \bigwedge_{1 \leq i < j \leq n} \|\mathbf{c^{(i)}} - \mathbf{c^{(j)}}\|_2 > 2 \end{aligned} \tag{5.1}$$

In (5.1), the variables $\mathbf{c^{(i)}}$ are used to represent the centres of the generalised spheres. Each variable in $\mathbf{c^{(i)}}$ is first restricted to be between ± 1. The last line in (5.1) states that the Euclidean distance between any two centre points is above 2.

Intuitively, if one visualises circles with a radius of 1, and a square of width 4, (5.1) specifies an arrangement of the (centres of the) circles where all circles are completely within the square and not touching each other. For example, one may place 3 circles in a square as shown in Figure 5.1.

Instances

We present instances of (5.1) that we created to test LPPaver and the chosen provers in Table 5.4. Each instance has the name 'PlaceDC', where D and C represent the dimension and the number of circles of the instance respectively. All instances can be found in the LPPaver code repository [53]

[1] 3-spheres and 3-cubes are four dimensional equivalents of spheres and cubes respectively.

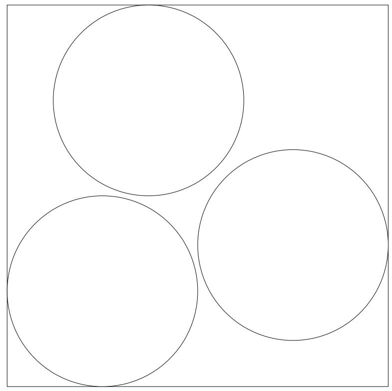


Figure 5.1: An arrangement of 3 equally sized circles in a square that satisfies (5.1).

under folder `benchmarks/place/txt/`. Note that these instances are similar (but not identical) to benchmarks described in the ksmt paper [11].

Increasing the number of circles increases the number of variables in the problem. Increasing the dimension also increases the number of variables but makes the problem conceptually easier at the same time. For example, there is no arrangement of circles in 'Place24' that satisfies (5.1): placing the centres of the four circles at each of the 'extreme' corners would result in all four circles being within the square, but they would be touching, as shown in Figure 5.2. This means Place2n where $n \geq 4$ violates (5.1). Increasing the dimension to 3 allows one to place up to 7 spheres in a cube before they would be touching. Thus, Place24, Place25, Place26, and Place27 are unsatisfiable, and all other instances are satisfiable. In summary, more circles *increase* the difficulty and higher dimensions *decrease* the difficulty of the problem.

In the files generated to instantiate (5.1), a variable is used as a constant to represent the value $\|\mathbf{c^{(i)}} - \mathbf{c^{(j)}}\|_2$, to both aid readability and to model the way a human would typically specify this problem. Constants are internally substituted wherever it is used in LPPaver and presumably other provers do the same, therefore we differentiate these constants from the more interesting variables: the constants should only have a negligible effect on

Table 5.4: Generated Instances of (5.1)

Problem	Dimension	#[a] of Circles	# of Variables	# of Constants
Place22	2	2	4	2
Place23	2	3	6	6
Place24	2	4	8	12
Place25	2	5	10	20
Place26	2	6	12	30
Place27	2	7	14	42
Place32	3	2	6	3
Place33	3	3	9	9
Place34	3	4	12	18
Place35	3	5	15	30
Place36	3	6	18	45
Place37	3	7	21	63
Place42	4	2	8	4
Place43	4	3	12	12
Place44	4	4	16	24
Place45	4	5	20	40
Place46	4	6	24	60
Place47	4	7	28	84

[a] # means number and is used to save space.

the performance of each prover. The variables mentioned in the table are specifically used to represent the values of each element of the vectors used to represent the centre of a circle. For example, Place22 would have the centre of one of the circles represented with vector $\mathbf{c}^{(1)}$, which is modelled using two distinct variables for both coordinates of $\mathbf{c}^{1}$.

Timings

We ran each prover on the instances described in Table 5.4. We allowed each prover 30 minutes to attempt to decide each problem. If a prover took longer than 30 minutes, we stopped the prover and recorded the result as a timeout (t/o). If a prover returned a satisfiable/unsatisfiable result, we

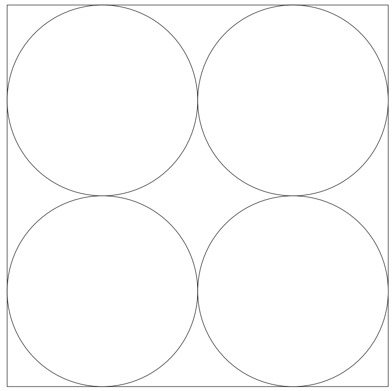


Figure 5.2: Packing of 4 equally sized circles in a square. This does not satisfy (5.1) as the circles are touching; the distances between the centres of the touching circles is exactly $2 \cdot radius$.

recorded this as 'sat'/'unsat' and recorded the time taken for the prover to make this decision. As mentioned in Section 2.7.2, dReal returns either 'unsat' or 'δ-sat'.[2] The δ used here is 10^{-100}. When running LPPaver, we ran the prover using the 'proving' algorithm (i.e. Algorithm 5) for the unsatisfiable problems, and the 'model finding' algorithm (i.e. Algorithm 8) for problems which should be satisfiable. The results are presented in Table 5.5.

Models

LPPaver and Colibri are able to produce models for problems that they decide are satisfiable. dReal produces δ-satisfiable models which are models that satisfy the δ-weakening of the problem but may not satisfy the original problem. This is why dReal gives a 'δ-sat' result for Place24. (5.2) is the δ-satisfiable model, and this model is approximated in Figure 5.3. As one can see, this model does not satisfy Place24 (but does satisfy the δ-weakening of Place24) as there will be a *very* slight overlap between the neighbouring circles and some circles will protrude outside the square.

[2] Recall that δ-sat means that some formula φ is satisfiable when weakened by numerically relaxing equalities and inequalities in φ by δ, e.g. $\sin(0) = 0$ would be weakened into $|\sin(0)| \leq \delta$.

Table 5.5: Results and timings of solvers on (5.1) instances

Problem	LPPaver		dReal		ksmt		Colibri	
Place22	sat	0.03s	δ-sat	0.01s	sat	0.00s	sat	0.05s
Place23	sat	0.08s	δ-sat	0.01s	sat	0.02s	sat	1.69s
Place24	t/o	-	δ-sat	0.02s	t/o	-	t/o	-
Place25	unsat	0.41s	unsat	18.05s	unsat	2.88s	t/o	-
Place26	unsat	0.91s	t/o	-	unsat	13.16s	t/o	-
Place27	unsat	1.58s	t/o	-	unsat	53.83s	t/o	-
Place32	sat	0.03s	δ-sat	0.00s	sat	0.00s	sat	0.06s
Place33	sat	0.23s	δ-sat	0.02s	sat	0.01s	sat	0.21s
Place34	sat	1.65s	δ-sat	0.04s	sat	0.03s	sat	0.74s
Place35	sat	2.08s	δ-sat	0.14s	sat	0.59s	t/o	-
Place36	sat	9.44s	t/o	-	t/o	-	t/o	-
Place37	sat	2m42s	t/o	-	t/o	-	t/o	-
Place42	sat	0.03s	δ-sat	0.01s	sat	0.00s	sat	0.07s
Place43	sat	0.56s	δ-sat	0.03s	sat	0.01s	sat	0.28s
Place44	sat	1m40s	t/o	-	sat	0.04s	sat	1.03s
Place45	sat	1m57s	t/o	-	sat	0.14s	sat	29.53s
Place46	t/o	-	t/o	-	sat	0.3s	t/o	-
Place47	t/o	-	t/o	-	sat	1s	t/o	-

$$
\begin{aligned}
c^{(1)} &= \begin{bmatrix} -1 \\ 1 \end{bmatrix} \\
c^{(2)} &= \begin{bmatrix} -1 \\ -0.99999999999999978 \end{bmatrix} \\
c^{(3)} &= \begin{bmatrix} 0.99999999999999978 \\ 1 \end{bmatrix} \\
c^{(4)} &= \begin{bmatrix} 0.99999999999999978 \\ -0.99999999999999978 \end{bmatrix}
\end{aligned} \tag{5.2}
$$

Table 5.6 shows models given by the three solvers for the Place23 instance. In this case, the δ-model given by dReal is a valid model for both the δ-weakening of Place23 and Place23 itself.

Table 5.7 shows models given by the solvers for each problem presented

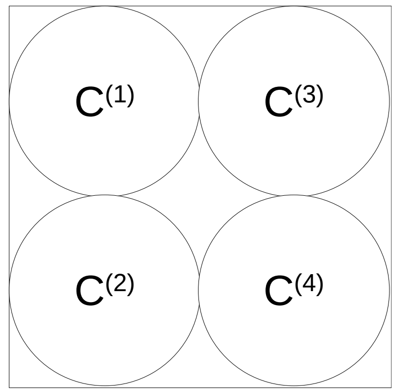


Figure 5.3: Approximation of the δ-model shown in (5.2) given by dReal. This does not satisfy (5.1) as some of the circles are overlapping.

in Table 5.4 that at least one of the model-producing solvers decided is satisfiable. If a given model is correct, we label this appropriately. For dReal, if a model is correct for the δ-weakening of the problem but not the problem itself, we label this as δ-correct.

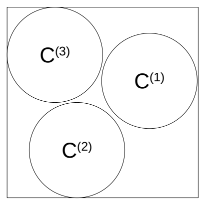
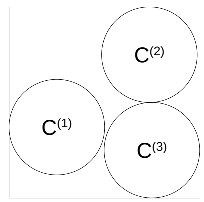
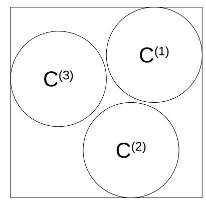
Reflections

The timings for MetiTarski are not present in Table 5.5 as MetiTarski timed out on every problem. This is not surprising as MetiTarski uses mainly symbolical methods, and the problem we present here is very numerical in nature.

LPPaver performed better than the other provers for the unsatisfiable problems, returning an 'unsat' result very quickly. LPPaver also performed well for the satisfiable problems. For the 2-dimensional satisfiable problems, LPPaver and the other provers returned 'sat' or 'δ-sat' results almost instantly.

LPPaver also outperformed other solvers for the 3-dimensional problems, being able to verify satisfiability of all problems. This is particularly impressive for Place36 and Place37 due to both the number of variables and the difficulty of finding an arrangement of spheres that satisfies (5.1). The next best-performing prover for this section was ksmt which was able to prove satisfiability for Place32–Place35. As we increase the number of spheres

Table 5.6: Models for Place23

	LPPaver	dReal	Colibri
Model for $c^{(1)}$	$\begin{bmatrix} 0.5\ldots \\ 0.959... \end{bmatrix}$	$\begin{bmatrix} -0.993\ldots \\ -0.372\ldots \end{bmatrix}$	$\begin{bmatrix} 1 \\ 1 \end{bmatrix}$
Model for $c^{(2)}$	$\begin{bmatrix} -1.0 \\ -0.789\ldots \end{bmatrix}$	$\begin{bmatrix} 0.976\ldots \\ 1 \end{bmatrix}$	$\begin{bmatrix} 0.5 \\ -1 \end{bmatrix}$
Model for $c^{(3)}$	$\begin{bmatrix} 1 \\ -1.0 \end{bmatrix}$	$\begin{bmatrix} 0.983\ldots \\ -0.999\ldots \end{bmatrix}$	$\begin{bmatrix} -1 \\ 0.5 \end{bmatrix}$
Approx. Figures			

for the 3-dimensional problems, LPPaver takes longer to prove satisfiability. This implies that the ‘model finding’ algorithm is affected by the number of variables. Note that the ‘showing unsatisfiability’ algorithm is also affected by the number of variables as can be seen in the results for Place25, Place26, and Place27: these problems become conceptually easier as the number of circles increase but they become harder to solve when using algorithms affected by the number of variables. This is typical for branch-and-prune based algorithms.

LPPaver performed quite well for the 4-dimensional problems, returning satisfiable results in a reasonable time frame. LPPaver took 1m57s to find a model for Place45 however, further implying that LPPaver’s algorithms is more affected by the number of variables when compared to approaches used in other solvers.

dReal performed well for the satisfiable problems and gave a ‘δ-sat’ result for Place33 and Place34. dReal also decided that Place24 is δ-satisfiable, though clearly the model given does not satisfy (5.1). This occurs because the δ-weakening of (5.1) allows a small (δ sized) overlap between circles and also allows circles to have a small (δ sized) overlap with the faces of the cube. dReal struggled with the remaining unsatisfiable problems where it

Table 5.7: Checking models given by provers for (5.1) instances

Problem	LPPaver	dReal	Colibri
Place22	correct	correct	correct
Place23	correct	correct	correct
Place24	N/A	δ-correct	N/A
Place32	correct	δ-correct	correct
Place33	correct	δ-correct	correct
Place34	correct	δ-correct	correct
Place35	correct	δ-correct	N/A
Place36	correct	N/A	N/A
Place37	correct	N/A	N/A
Place42	correct	correct	correct
Place43	correct	δ-correct	correct
Place44	correct	N/A	correct
Place45	correct	N/A	correct

decided ‘unsat’ in 18.05 seconds for Place25 and timed out for Place26 and Place27. dReal also struggled with the 4-dimensional problems, most likely due to the number of variables.

ksmt gave results for all of the problems except Place24, which is exceptionally difficult due to touching, Place36, and Place37. The number of variables may also increase the difficulty of the problem, however ksmt seems to be less affected by the number of variables when compared to other solvers. For example, ksmt decided Place35 with 15 variables in 0.59 seconds and Place45 with 20 variables in 0.04 seconds. One drawback of ksmt is that it does not give a model for problems that it decided is satisfiable.

Colibri performed fairly well on the satisfiable problems, but timed out on all of the unsatisfiable problems. Colibri is more affected by the conceptual difficulty of the problem rather than the number of variables. For example, Colibri timed out on Place35 which has 15 variables, but verified Place45 which has 20 variables in 29.53s.

To summarise, LPPaver performed the best on the unsatisfiable problems. In comparison with the other provers, LPPaver performed well for the satisfiable problems, and was the only solver able to verify Place36 and Place37. Both LPPaver and ksmt were able to solve 15 of the Place problems, more than the other solvers we tested. LPPaver was able to make decisions faster than ksmt for the unsatisfiable problems. ksmt was able to make decisions faster than LPPaver for the satisfiable problems, but ksmt does not produce a model. The results also show how the LPPaver algorithms can slow down the number of variables increases.

Chapter 6

Conclusion

We now conclude the thesis, stating our main contributions and potential avenues for future work. This is done separately for the ideas implemented in both LPPaver and PropaFP.

6.1 LPPaver

We have developed a numerical solver, LPPaver, that targets problems involving nonlinear real arithmetic. LPPaver uses interval methods and implements a variant of a branch-and-prune algorithm. LPPaver also uses a form of interval constraint propagation to help 'prune' boxes by using interval methods to decide the truth value of terms over a given box.

LPPaver implements novel contractions based on linearisations of conjunctions of nonlinear inequalities that are used to *weaken* the conjunctions. These linearisations are used to produce a system of linear inequalities which are analysed using the simplex method. The *weakening* linearisation is used to *contract* a box by removing areas from the box containing values which violate the linearised conjunction. The removed area is guaranteed to also violate the original nonlinear conjunction. Similarly, a *strengthening* linearisation is used to find a *model* for the linearised conjunction within a box. The same model is also valid for the original nonlinear conjunction.

During our evaluation, we found that LPPaver performed comparably with, and in some cases better than, other automated state-of-the-art solvers for problems involving nonlinear real arithmetic, as shown in Tables 5.5 and 5.1. Tables 5.6 and 5.3 show how LPPaver produced useful models for satisfiable problems in a reasonable time-frame. We also discovered how LPPaver is affected by the number of variables in a problem, with a high number of variables causing slowdowns in the time it takes LPPaver to make a decision.

We now discuss potential avenues for future work regarding LPPaver and these novel contractors.

6.1.1 Future Work

Run both algorithms at once. LPPaver has two algorithms, one that focuses on finding models and one that focuses on proving the absence of a model. If a problem has a model, then LPPaver will perform better when using the model-finding algorithm and vice-versa, however, a user may not know whether or not a problem they are giving to LPPaver contains a model. So, we propose a mode where LPPaver runs both algorithms simultaneously, terminating both algorithms as soon as one gives a decisive (i.e. satisfiable or unsatisfiable) result.

Remove variables from the box where possible. As LPPaver is sometimes significantly affected by the number of variables, it would be beneficial to remove variables from a box whenever it is safe to do so. This could be done when filtering out terms in a conjunction: if a variable only occurred in terms that have been filtered out, we can safely remove said variable from the box.

Alternative heuristic for choosing a variable to split. Currently, when choosing a variable to split, LPPaver always chooses to split a variable corresponding to some box dimension with the largest width. This may not be desirable. For example, if one variable has a much larger domain than the others, LPPaver may choose to split this multiple times when it may be more beneficial to split other variables. It would be beneficial to implement alternative heuristics for choosing the variable to split, for example, a ‘round robin’ method where each variable is split in a predetermined order.

Rotations. In some cases, rotating a box may lead to a better contraction, increasing the efficiency of LPPaver’s algorithms. See Figure 6.1 for an example.

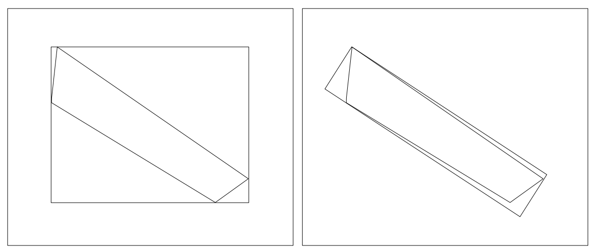


Figure 6.1: On the left, we have a contraction of a box using a system of inequalities. On the right, we rotate the box while contracting, reducing to a much smaller box.

Implement DPLL(T) or similar. One could combine LPPaver with OpenSMT, an open source implementation of the DPLL(T) algorithm that is used in dReal. Alternatively, we could integrate LPPaver’s methods with other solvers that implement DPLL(T), e.g. Z3. This would give users access to powerful symbolic proving methods combined with LPPaver’s powerful numerical proving methods.

Return system used to find a model. It may be beneficial for users for LPPaver to return the system of linear inequalities that was used to produce a model for a given problem. Users would be able to use the system to find an alternative model if desirable, e.g., when searching for a counter-example to the original specification of an FP program.

Implement novel contractors in other solvers. The novel contractors we describe for conjunctions of linear inequalities can be implemented in other numerical solvers that uses similar methods. It would be feasible to implement these contractors in dReal (via RealPaver) which may lead to an increase in performance. The systems produced by our linearisations could be solved by a tool like SoPlex [29] which implements an exact (rational) simplex in C++ [33, 34], the language that both RealPaver and dReal are implemented in. SoPlex is licenced under the ZIB Academic Licence, making it free for academic use.

Verified implementation. LPPaver is implemented in Haskell with the AERN2 library. There is a tool to develop verified AERN2 programs in Coq named coq-aern [42]. To improve a user's trust in LPPaver, it may be worth rewriting and verifying LPPaver in Coq using coq-aern.

6.2 PropaFP

We have also presented an automated proving process for deciding VCs that arise in the verification of FP programs with a strong functional specification. Our implementation of the process builds on SPARK, GNATprove, and Why3, and utilises FPTaylor and the nonlinear real provers dReal, MetiTarski, and LPPaver. This process could be adapted for other tools and languages, as long as one can generate NVCs similar to those generated by GNATprove.

The process can be summarised as follows:

1. Why3 reads a program with its specification and produces NVCs (Negated VCs).
2. PropaFP processes the NVCs as follows:
    (a) Simplify the NVC using simple symbolic rules and interval evaluation.
    (b) Derive bounds for all variables in the NVC, interleaving with (a).
    (c) Derive bounds for rounding errors in expressions with FP operations.

   (d) Using these bounds, safely replace FP operations with exact operations.
   (e) Repeat the simplification steps (a–b).

3. Apply nonlinear real provers on the processed NVCs to either prove them or get *potential* counter-examples.

This proving process should, in principle, work with tools and languages other than Why3 and SPARK, as long as one can generate NVCs similar to those generated by GNATprove.

We demonstrated our proving process on three examples of increasing complexity, featuring loops, real-integer interactions, and subprogram calls. Notably, we have contributed the first fully automatically verified SPARK implementations of the sine and square root functions. The examples demonstrate an improvement on the state-of-the-art in the power of automated FP software verification.

Table 5.1 indicates that our proving process can automatically and fairly quickly decide certain VCs that are currently considered difficult. Table 5.2 demonstrates how the process speeds up when using looser bounds in specifications. Table 5.3 shows that our proving process can quickly find potential, and often even actual, counter-examples for a range of common incorrect specifications.

Our examples may be used as a suite for benchmarking future FP verification approaches, The NVCs resulting from these examples can be used as benchmarks for nonlinear real provers and were used in this thesis to evaluate LPPaver.

Future work

We conclude with thoughts on how our process could be further improved.

Executable exact real specifications. We plan to make specifications containing functions such as $\sqrt{\cdot}$ executable via high-accuracy interval arithmetic, allowing the developer or IDE to check whether the suggested counter-examples are valid.

Adapting the provers. We would like the provers we used in this paper to be improved in some ways. Ideally, the provers will be able to decide all of our examples. Support for integer rounding could be added to dReal, using methods similar to those used in LPPaver. It should also not be difficult to add support for larger integers in dReal. Support for both integer rounding and the modulus operator could be added to MetiTarski. Adding these features will allow both dReal and MetiTarski to have an attempt at deciding all of our examples.

Why3 integration. Our VC processing steps could be integrated into Why3. This would include simplifications, bound derivation, and FP elimination. As Why3 transformations, the VC processing steps would be more accessible for users who are familiar with Why3. Also, the proving process would thus become easily available to the many tools that support Why3.

Support function calls. Having to manually translate functions into procedures is undesirable. Support for function calls could be added, e.g., by a Why3 transformation that translates functions into procedures.

Use Abstract Interpretation. We currently derive bounds for variables using our own iterative process similar to Abstract Interpretation over the interval domain. It would be interesting to see if the proving process would improve if we use an established Abstract Interpretation implementation to derive bounds. If nothing else, such a change would reduce the amount of new code that a user would need to trust.

More Tools and Solvers. Connect PropaFP to other tools and solvers, notably alternative FP analysers such as Rosa, Gappa, and PRECiSA. We could connect PropaFP to other solvers. Colibri, for example, would give

PropaFP access to a solver that uses constraint programming methods. We could also connect PropaFP to Frama-C and Krakatoa which would allow the use of PropaFP's methods when verifying FP C and Java programs, respectively.

Verified implementation. We would like to formally verify some elements of our process to ensure that the transformation steps are performed correctly. Like LPPaver, PropaFP is implemented in Haskell and utilises AERN2, so a rewrite in Coq with coq-aern may be a feasible verification route.

Bibliography

[1] Behzad Akbarpour and Lawrence Charles Paulson. "MetiTarski: An Automatic Theorem Prover for Real-Valued Special Functions". en. In: *Journal of Automated Reasoning* 44.3 (Mar. 2010), pp. 175–205. ISSN: 1573-0670. URL: https://doi.org/10.1007/s10817-009-9149-2 (visited on 01/28/2020).

[2] Clark Barrett, Aaron Stump, Cesare Tinelli, et al. "The SMT-LIB standard: Version 2.0". In: *Proceedings of the 8th international workshop on satisfiability modulo theories (Edinburgh, UK)*. Vol. 13. 2010, p. 14.

[3] Clark Barrett and Cesare Tinelli. "Satisfiability Modulo Theories". en. In: *Handbook of Model Checking*. Ed. by Edmund M. Clarke et al. Cham: Springer International Publishing, 2018, pp. 305–343. ISBN: 978-3-319-10575-8. DOI: 10.1007/978-3-319-10575-8_11. URL: https://doi.org/10.1007/978-3-319-10575-8_11 (visited on 07/28/2022).

[4] Clark Barrett et al. "CVC4". en. In: *Computer Aided Verification*. Ed. by Ganesh Gopalakrishnan and Shaz Qadeer. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2011, pp. 171–177. ISBN: 978-3-642-22110-1. DOI: 10.1007/978-3-642-22110-1_14.

[5] Mohammed Said Belaid, Claude Michel, and Michel Rueher. "Boosting local consistency algorithms over floating-point numbers". In: *International Conference on Principles and Practice of Constraint Programming*. Springer. 2012, pp. 127–140.

[6] Yves Bertot and Pierre Castéran. *Interactive theorem proving and program development: Coq'Art: the calculus of inductive constructions*. Springer Science & Business Media, 2013.

[7] François Bobot et al. “Why3: Shepherd Your Herd of Provers”. en. In: *Boogie 2011: First International Workshop on Intermediate Verification Languages*. 2011, pp. 53–64. URL: https://hal.inria.fr/hal-00790310 (visited on 01/28/2020).

[8] Sylvie Boldo and Claude Marché. “Formal verification of numerical programs: from C annotated programs to mechanical proofs”. In: *Mathematics in Computer Science* 5 (2011), pp. 377–393. URL: https://hal.inria.fr/hal-00777605.

[9] Sylvie Boldo et al. “Wave Equation Numerical Resolution: A Comprehensive Mechanized Proof of a C Program”. en. In: *Journal of Automated Reasoning* 50.4 (Apr. 2013), pp. 423–456. ISSN: 1573-0670. DOI: 10.1007/s10817-012-9255-4. (Visited on 08/06/2022).

[10] Franz Brauße et al. “A CDCL-style calculus for solving non-linear constraints”. In: *International Symposium on Frontiers of Combining Systems*. Springer. 2019, pp. 131–148.

[11] Franz Brauße et al. *The ksmt calculus is a delta-complete decision procedure for non-linear constraints*. Tech. rep. arXiv:2104.13269. arXiv:2104.13269 [cs] type: article. arXiv, Apr. 2021. URL: http://arxiv.org/abs/2104.13269 (visited on 06/12/2022).

[12] Roberto Bruttomesso et al. “The OpenSMT Solver”. en. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by Javier Esparza and Rupak Majumdar. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2010, pp. 150–153. ISBN: 978-3-642-12002-2. DOI: 10.1007/978-3-642-12002-2_12.

[13] Ole Caprani and Kaj Madsen. “Mean value forms in interval analysis”. In: *Computing* 25.2 (1980), pp. 147–154.

[14] Sylvain Conchon et al. "Alt-Ergo 2.2". en. In: *SMT Workshop: International Workshop on Satisfiability Modulo Theories*. July 2018. URL: https://hal.inria.fr/hal-01960203 (visited on 01/28/2020).

[15] Patrick Cousot and Radhia Cousot. "Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints". In: *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. POPL '77. Los Angeles, California: Association for Computing Machinery, 1977, pp. 238–252. ISBN: 9781450373500. DOI: 10.1145/512950.512973.

[16] Pascal Cuoq et al. "Frama-c". In: *International conference on software engineering and formal methods*. Springer. 2012, pp. 233–247.

[17] Sylvain Dailler et al. "Instrumenting a Weakest Precondition Calculus for Counterexample Generation". In: *Journal of Logical and Algebraic Methods in Programming* 99 (2018), pp. 97–113. URL: https://hal.inria.fr/hal-01802488.

[18] George B Dantzig, Alex Orden, Philip Wolfe, et al. "The generalized simplex method for minimizing a linear form under linear inequality restraints". In: *Pacific Journal of Mathematics* 5.2 (1955), pp. 183–195.

[19] Eva Darulova and Viktor Kuncak. "Towards a compiler for reals". In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 39.2 (2017), pp. 1–28. DOI: 10.1145/3014426.

[20] Marc Daumas and Guillaume Melquiond. "Certification of Bounds on Expressions Involving Rounded Operators". In: *ACM Trans. Math. Softw.* 37.1 (Jan. 2010). ISSN: 0098-3500. DOI: 10.1145/1644001.1644003. URL: https://doi.org/10.1145/1644001.1644003.

[21] Ernest Davis. "Constraint propagation with interval labels". en. In: *Artificial Intelligence* 32.3 (July 1987), pp. 281–331. ISSN: 0004-3702. DOI: 10.1016/0004-3702(87)90091-9. URL: https://www.sciencedirect.com/science/article/pii/0004370287900919 (visited on 09/20/2022).

[22] Martin Davis et al. *A machine program for theorem-proving*. Air Force Office of Scientific Research ;AFOSR 819. New York, New York: New York University, Institute of Mathematical Sciences, 1961. URL: https://catalog.hathitrust.org/Record/102755642 (visited on 09/19/2022).

[23] Edsger Wybe Dijkstra et al. *A discipline of programming*. Vol. 613924118. prentice-hall Englewood Cliffs, 1976.

[24] Claire Dross and Johannes Kanig. "Making Proofs of Floating-Point Programs Accessible to Regular Developers". en. In: *Software Verification*. Ed. by Roderick Bloem et al. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2022, pp. 7–24. ISBN: 978-3-030-95561-8. DOI: 10.1007/978-3-030-95561-8_2.

[25] Jan Duracz and Michal Konečný. "Polynomial function intervals for floating-point software verification". In: *Annals of Mathematics and Artificial Intelligence* 70.4 (2014), pp. 351–398. DOI: 10.1007/s10472-014-9409-7.

[26] Thibaut Feydy, Andreas Schutt, and Peter J Stuckey. "Global difference constraint propagation for finite domain solvers". In: *Proceedings of the 10th international ACM SIGPLAN conference on Principles and practice of declarative programming*. 2008, pp. 226–235.

[27] Jean-Christophe Filliâtre and Claude Marché. "The Why/Krakatoa/Caduceus platform for deductive program verification". In: *International Conference on Computer Aided Verification*. Springer. 2007, pp. 173–177.

[28] Clément Fumex, Claude Marché, and Yannick Moy. *Automated Verification of Floating-Point Computations in Ada Programs*. en. report. Inria Saclay Ile de France, Apr. 2017, p. 53. URL: https://hal.inria.fr/hal-01511183/document (visited on 02/13/2019).

[29] Gerald Gamrath et al. *The SCIP Optimization Suite 7.0*. eng. Tech. rep. 20-10. Takustr. 7, 14195 Berlin: ZIB, 2020.

[30] Harald Ganzinger et al. "DPLL(T): Fast Decision Procedures". en. In: *Computer Aided Verification*. Ed. by Rajeev Alur and Doron A. Peled. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer,

2004, pp. 175–188. ISBN: 978-3-540-27813-9. DOI: 10.1007/978-3-540-27813-9_14.

[31] Sicun Gao, Jeremy Avigad, and Edmund M. Clarke. "δ-Complete Decision Procedures for Satisfiability over the Reals". en. In: *Automated Reasoning*. Ed. by Bernhard Gramlich, Dale Miller, and Uli Sattler. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2012, pp. 286–300. ISBN: 978-3-642-31365-3. DOI: 10.1007/978-3-642-31365-3_23.

[32] Sicun Gao, Soonho Kong, and Edmund M. Clarke. "dReal: An SMT Solver for Nonlinear Theories over the Reals". en. In: *Automated Deduction – CADE-24*. Ed. by Maria Paola Bonacina. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2013, pp. 208–214. ISBN: 978-3-642-38574-2.

[33] Ambros Gleixner, Daniel Steffy, and Kati Wolter. *Improving the Accuracy of Linear Programming Solvers with Iterative Refinement*. eng. Tech. rep. 12-19. Takustr. 7, 14195 Berlin: ZIB, 2012.

[34] Ambros Gleixner, Daniel Steffy, and Kati Wolter. *Iterative Refinement for Linear Programming*. eng. Tech. rep. 15-15. Takustr. 7, 14195 Berlin: ZIB, 2015.

[35] Laurent Granvilliers and Frédéric Benhamou. "Algorithm 852: RealPaver: an interval solver using constraint satisfaction techniques". In: *ACM Transactions on Mathematical Software* 32.1 (Mar. 2006), pp. 138–156. ISSN: 0098-3500. DOI: 10.1145/1132973.1132980. URL: https://doi.org/10.1145/1132973.1132980 (visited on 07/25/2022).

[36] Andreas Griewank et al. "On automatic differentiation". In: *Mathematical Programming: recent developments and applications* 6.6 (1989), pp. 83–107.

[37] E. R. Hansen and R. I. Greenberg. "An interval Newton method". en. In: *Applied Mathematics and Computation* 12.2 (May 1983), pp. 89–98. ISSN: 0096-3003. DOI: 10.1016/0096-3003(83)90001-2. URL:

https://www.sciencedirect.com/science/article/pii/0096300383900012 (visited on 08/04/2022).

[38] Timothy Hickey, Qun Ju, and Maarten H Van Emden. "Interval arithmetic: From principles to implementation". In: *Journal of the ACM (JACM)* 48.5 (2001), pp. 1038–1068.

[39] Duc Hoang et al. "SPARK 2014 and GNATprove". en. In: *International Journal on Software Tools for Technology Transfer* 17.6 (Nov. 2015), pp. 695–707. ISSN: 1433-2787. DOI: 10.1007/s10009-014-0322-5. URL: https://doi.org/10.1007/s10009-014-0322-5 (visited on 04/11/2022).

[40] "IEEE Standard for Floating-Point Arithmetic". In: *IEEE Std 754-2019 (Revision of IEEE 754-2008)* (July 2019), pp. 1–84. DOI: 10.1109/IEEESTD.2019.8766229.

[41] Michal Konečný and et al. *AERN2*. July 2022. URL: https://github.com/michalkonecny/aern2 (visited on 08/05/2022).

[42] Michal Konečný, Sewon Park, and Holger Thies. "Axiomatic Reals and Certified Efficient Exact Real Computation". en. In: *Logic, Language, Information, and Computation*. Ed. by Alexandra Silva, Renata Wassermann, and Ruy de Queiroz. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2021, pp. 252–268. ISBN: 978-3-030-88853-4. DOI: 10.1007/978-3-030-88853-4_16.

[43] K Rustan M Leino and Michał Moskal. "Usable auto-active verification". In: *Usable Verification Workshop. http://fm. csl. sri. com/UV10*. Citeseer. 2010.

[44] Claude Marché and Yannick Moy. "The Jessie plugin for deductive verification in Frama-C". In: *INRIA Saclay Île-de-France and LRI, CNRS UMR* (2012).

[45] J.P. Marques Silva and K.A. Sakallah. "GRASP-A new search algorithm for satisfiability". In: *Proceedings of International Conference on Computer Aided Design*. Nov. 1996, pp. 220–227. DOI: 10.1109/ICCAD.1996.569607.

[46] Bruno Marre, Francois Bobot, and Zakaria Chihani. “Real Behavior of Floating Point Numbers”. en. In: (2017), p. 12.

[47] John McCarthy. “Recursive functions of symbolic expressions and their computation by machine, part I”. In: *Communications of the ACM* 3.4 (1960), pp. 184–195.

[48] Ramon E Moore, R Baker Kearfott, and Michael J Cloud. *Introduction to interval analysis*. SIAM, 2009.

[49] Leonardo de Moura and Nikolaj Bjørner. “Z3: An Efficient SMT Solver”. en. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by C. R. Ramakrishnan and Jakob Rehof. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2008, pp. 337–340. ISBN: 978-3-540-78800-3. DOI: 10.1007/978-3-540-78800-3_24.

[50] *PATRIOT MISSILE DEFENSE: Software Problem Led to System Failure at Dhahran, Saudi Arabia*. en. Tech. rep. Section: Technical Reports. URL: https://apps.dtic.mil/sti/citations/ADA344865 (visited on 09/22/2022).

[51] Lawrence C Paulson. *Isabelle: A generic theorem prover*. Springer, 1994.

[52] Junaid Rasheed. *Exact Two-Phase Simplex Method in Haskell*. URL: https://github.com/rasheedja/simplex-method (visited on 09/27/2022).

[53] Junaid Rasheed. *LPPaver*. URL: https://github.com/rasheedja/LPPaver (visited on 06/23/2022).

[54] Junaid Rasheed. *PropaFP*. URL: https://github.com/rasheedja/PropaFP (visited on 06/23/2022).

[55] Junaid Rasheed and Michal Konečný. “Auto-Active Verification Of Floating-Point Programs Via Nonlinear Real Provers”. In: *Software Engineering and Formal Methods: 20th International Conference, SEFM 2022, Berlin, Germany, September 26–30, 2022, Proceedings*. Berlin, Germany: Springer-Verlag, 2022, pp. 20–36. ISBN: 978-3-031-17107-9. DOI: 10.1007/978-3-031-17108-6_2. URL: https://doi.org/10.1007/978-3-031-17108-6_2.

[56] Junaid Rasheed and Michal Konečný. "Auto-Active Verification of Floating-Point Programs via Nonlinear Real Provers". In: *International Conference on Software Engineering and Formal Methods*. Springer. 2022, pp. 20–36.

[57] Junaid Rasheed and Michal Konečný. "Auto-active Verification of Floating-point Programs via Nonlinear Real Provers". In: (2022). DOI: 10.48550/ARXIV.2207.00921. URL: https://arxiv.org/abs/2207.00921.

[58] Siegfried M Rump. "Fast and parallel interval arithmetic". In: *BIT Numerical Mathematics* 39.3 (1999), pp. 534–554.

[59] Rocco Salvia et al. "A Mixed Real and Floating-Point Solver". en. In: *NASA Formal Methods*. Ed. by Julia M. Badger and Kristin Yvonne Rozier. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2019, pp. 363–370. ISBN: 978-3-030-20652-9. DOI: 10.1007/978-3-030-20652-9_25.

[60] Alexey Solovyev et al. "Rigorous Estimation of Floating-Point Round-Off Errors with Symbolic Taylor Expansions". en. In: *ACM Transactions on Programming Languages and Systems* 41.1 (Mar. 2019), pp. 1–39. ISSN: 0164-0925, 1558-4593. DOI: 10.1145/3230733. URL: https://dl.acm.org/doi/10.1145/3230733 (visited on 06/20/2022).

[61] Laura Titolo et al. "An Abstract Interpretation Framework for the Round-Off Error Analysis of Floating-Point Programs". en. In: *Verification, Model Checking, and Abstract Interpretation*. Ed. by Isil Dillig and Jens Palsberg. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2018, pp. 516–537. ISBN: 978-3-319-73721-8. DOI: 10.1007/978-3-319-73721-8_24.