Towards justifying computer algebra algorithms in Isabelle/HOL

Wenda Li

Computer Laboratory
University of Cambridge

This dissertation is submitted for the degree of
Doctor of Philosophy

Queens' College July 2018

To my parents, Qianning and Liming,
and to my beloved girl, Lei

Declaration

I hereby declare that except where specific reference is made to the work of others, the contents of this dissertation are original and have not been submitted in whole or in part for consideration for any other degree or qualification in this, or any other university. This dissertation is my own work and contains nothing which is the outcome of work done in collaboration with others, except as specified in the text and Acknowledgements. This dissertation contains fewer than 60,000 words including appendices, bibliography, footnotes, tables and equations and has fewer than 150 figures.

Wenda Li
July 2018

Acknowledgements

First and foremost, I owe a deep debt of gratitude to my supervisor, Prof. Lawrence C. Paulson, who led me to the field of interactive theorem proving and inspired me with his enthusiasm for science. I cannot thank him enough for his patience and guidance that helped me grow as an independent researcher. He is the best supervisor I can hope for.

I am also immensely grateful to Dr. Grant Passmore , who gave me numerous suggestions about my project. His passion of bringing formal verification techniques to the industry deeply motivated me.

I also thank Prof. James H. Davenport and Dr. Neelakantan R. Krishnaswami for being my examiners and giving insightful comments on the thesis.

My parents, Qianning Li and Liming Zhang, gave me their unconditional love and support, without which I would certainly not have come this far. Last but not least, I would like to thank my beloved girl Lei, for being in my life and lighting up my world. No matter what happens, good or bad, she is the one I share my story with.

Abstract

As verification efforts using interactive theorem proving grow, we are in need of certified algorithms in computer algebra to tackle problems over the real numbers. This is important because uncertified procedures can drastically increase the size of the trust base and undermine the overall confidence established by interactive theorem provers, which usually rely on a small kernel to ensure the soundness of derived results.

This thesis describes an ongoing effort using the Isabelle theorem prover to certify the cylindrical algebraic decomposition (CAD) algorithm, which has been widely implemented to solve non-linear problems in various engineering and mathematical fields. Because of the sophistication of this algorithm, people are in doubt of the correctness of its implementation when deploying it to safety-critical verification projects, and such doubts motivate this thesis.

In particular, this thesis proposes a library of real algebraic numbers, whose distinguishing features include a modular architecture and a sign determination algorithm requiring only rational arithmetic. With this library, an Isabelle tactic based on univariate CAD has been built in a certificate-based way: external, untrusted code delivers solutions in the form of certificates that are checked within Isabelle. To lay the foundation for the multivariate case, I have formalised various analytical results including Cauchy's residue theorem and the bivariate case of the projection theorem of CAD. During this process, I have also built a tactic to evaluate winding numbers through Cauchy indices and verified procedures to count complex roots in some domains.

The formalisation effort in this thesis can be considered as the first step towards a certified computer algebra system inside a theorem prover, so that various engineering projections and mathematical calculations can be carried out in a high-confidence framework.

Table of contents

List of figures **xv**

1 Introduction **1**

1.1 Interactive theorem proving 2

1.2 Why we need to justify algorithms in computer algebra 2

1.3 Thesis overview 3

1.4 Publications 4

1.5 Contributions 5

2 Introduction to cylindrical algebraic decomposition **7**

2.1 Basic idea 7

2.2 The classic algorithm 11

2.3 Remarks 16

3 The Sturm-Tarski theorem **19**

3.1 Formulation 19

3.2 A formal proof of the Sturm-Tarski theorem 20

3.3 Remarks 23

4 Real algebraic numbers **25**

4.1 Construction on an abstract level 26

4.2 Implementation 31

4.2.1 More pseudo-constructors on real numbers 31

4.2.2 Univariate sign determination through the Sturm-Tarski theorem . . 34

4.2.3 Deciding the sign of a bivariate polynomial at a real algebraic point 37

4.2.4 Enable executability on algebraic reals 42

4.2.5 Linking the algebraic reals to the real algebraic numbers 45

4.3 Experiments 45

4.4 Related work 46

4.5 Remarks . . . 48
4.5.1 Modularity . . . 48
4.5.2 A potential problem . . . 49
4.5.3 Intended applications . . . 50

5 Deciding univariate polynomial problems using untrusted certificates 51
5.1 A motivating example . . . 52
5.2 A sketch of the certificate-based design . . . 54
5.3 The formal development of the proof procedure . . . 56
5.3.1 Parsing formulas . . . 56
5.3.2 Existential case . . . 59
5.3.3 Universal case . . . 59
5.4 Linking to an external solver . . . 61
5.5 Experiments and related work . . . 61

6 A formal proof of Cauchy's residue theorem 67
6.1 Background . . . 68
6.1.1 Contour integrals . . . 68
6.1.2 Valid paths . . . 68
6.1.3 Winding number . . . 69
6.1.4 Holomorphic functions and Cauchy's integral theorem . . . 69
6.2 Cauchy's residue theorem . . . 70
6.2.1 Residue . . . 70
6.2.2 Generalisation to a finite number of singularities . . . 72
6.2.3 Applications . . . 74
6.2.4 Remarks on the formalisation . . . 76
6.3 The argument principle . . . 76
6.3.1 Zeros and poles . . . 77
6.3.2 The main proof . . . 80
6.3.3 Remarks . . . 81
6.4 Rouché's theorem . . . 81
6.5 Related work . . . 82

7 Cauchy indices on the complex plane 85
7.1 A motivating example . . . 85
7.2 The intuition . . . 88
7.3 Evaluate winding numbers . . . 91

7.3.1 A formal proof of Proposition 7.8 92
7.3.2 A tactic for evaluating winding numbers 99
7.3.3 Subtleties 102
7.4 Counting the number of complex roots 105
7.4.1 Roots in a rectangle 106
7.4.2 Roots in a half-plane 111
7.5 Limitations and future work 115
7.6 Remarks and potential applications 116

8 Towards certifying multivariate CAD 119
8.1 Polynomial roots continuously depend on coefficients 119
8.2 Formal development towards the projection theorem of CAD 121
8.3 Towards certifying multivariate CAD 124

9 Conclusion 125
9.1 On formalised mathematics 125
9.2 Computer algebra in proof assistants 126

References 129

List of figures

2.1 Plot with $p_1(x_1,x_2) = x_2^2 + x_1^2 - 3 = 0$ and $p_2(x_1,x_2) = x_2 - x_1^2/2 = 0$ 8
2.2 Stack with $S = \{(x_1,x_2) \mid x_1^2 + x_2^2 \le 1\}$, $D_2 = \{(x,x') \in S \times \mathbb{R} \mid x' = f_1(x)\}$ and $D_5 = \{(x,x') \in S \times \mathbb{R} \mid f_2(x) < x'\}$ where $f_1 = (x_1^2 - x_2^2)/2$ and $f_2 = (-x_1^2 + x_2^2 + 3)/2$ 10

3.1 Plot of the rational function $(x-4)/((x-3)(x-1)^2(x+1))$ 21

4.1 Comparison between verified evaluation and unverified evaluation 45
4.2 A large bivariate polynomial 47
4.3 Dependence tree of my formalisation of real algebraic numbers 49

5.1 The plot of $p(x) = \frac{1}{2}x^2 - 1$ and $q(x) = x + 3$ 52
5.2 Comparison between my tactic in Isabelle and the tarski strategy in PVS: univ_rcf includes certificate searching and checking, while univ_rcf_cert includes only checking 63

6.1 Circlepath c_e and c_ε around an isolated singularity z 71
6.2 Induction on the number of singularities 73
6.3 A semicircular path centred at 0 with radius $R > 1$ 75
6.4 The path image of $\lambda t. 1 + \frac{g(\gamma(t))}{f(\gamma(t))}$ when $|f(w)| > |g(w)|$ for all w on the image of γ 83

7.1 Complex points $(0,-i)$ and $(0,i)$, and a closed path $L_r + C_r$ 86
7.2 Left: a path γ crosses the line $\{z \mid \mathrm{Re}(z) = \mathrm{Re}(z_0)\}$ at $\gamma(t_0)$ such that $\mathrm{Re}(\gamma(t_0)) > \mathrm{Re}(z_0)$. Right: the image of f as a point travels through γ . . . 88
7.3 Evaluating $n(L_r + C_r, i)$ and $n(L_r + C_r, -i)$ through the way that the path $L_r + C_r$ crosses the imaginary axis 91
7.4 Inductive cases when applying Lemma 7.13 96
7.5 To derive $n(\gamma, z_0) = -\frac{\mathrm{Indp}(\gamma, z_0)}{2}$ when γ is a loop 98
7.6 Different ways a path γ can intersect with the line $\{z \mid \mathrm{Re}(z) = \mathrm{Re}(z_0)\}$ . . . 103

7.7 Complex roots of a polynomial (red dots) and a rectangular path ($L_1 + L_2 + L_3 + L_4$) on the complex plane 106
7.8 A complex point i and a rectangle defined by its lower left corner -1 and upper right corner $2+2i$ 111
7.9 Complex roots of a polynomial (red dots) and a linear path (L_r) concatenated by a semi-circular path (C_r) on the complex plane 111
7.10 Complex roots of a polynomial (red dots) and a vector $(0,i)$ 115

Chapter 1

Introduction

Modern society is built upon a staggeringly complex tangle of software and hardware systems. As the size of those systems grows, it becomes increasingly hard for traditional verification techniques, such as pen-and-paper reasoning and testing, to guarantee the system we build is actually what we want. System failures due to elusive bugs have already caused catastrophic accidents that cost money and even human lives: famous examples include

- Intel's Pentium FDIV bug in 1994 that cost the company $475 million,[1]
- the failure of the Ariane 5 rocket in 1996 due to a software bug of integer overflow, which led to a loss of $370 million,[2]
- the error in the trading software of Knight Capital in 2012, which cost the company about $460 million,[3]
- and the very recent WannaCry ransomware attack launched by malicious hackers who exploited Windows' vulnerability, which even took down some NHS services in the UK.[4]

As a result, people have begun to resort to modern verification techniques such as model checking [19] and interactive theorem proving.

[1]https://en.wikipedia.org/wiki/Pentium_FDIV_bug
[2]https://en.wikipedia.org/wiki/Cluster_(spacecraft)
[3]https://en.wikipedia.org/wiki/Knight_Capital_Group
[4]https://en.wikipedia.org/wiki/WannaCry_ransomware_attack

1.1 Interactive theorem proving

Interactive theorem provers, such as Coq [9], Isabelle [75] and HOL Light [44], usually have an expressive logic implemented on a small trusted kernel. Inferences are carried out through the kernel to ensure the soundness of derived theorems/properties. Compared to fully automatic approaches like model checking, the main advantages of interactive theorem proving include

- extremely high confidence in soundness, due to the minimal trust base (i.e., the kernel) and the ongoing effort to further verify the kernel down to the machine-code level [54],
- the expressiveness of the logic, which is usually enough to encode all the functional properties we are interested in,
- the compositionality of the deduction steps, which enable us to derive the target properties with enough human guidance.

Thanks to these features, interactive theorem provers have been used to build realistic systems with full functional correctness verified. Major examples include: seL4 [53], a commercial operating system kernel (verified by Isabelle), and CompCert [56], an optimised C compiler (verified by Coq). Interactive theorem provers have also been used to verify gigantic mathematical proofs, such as the proof of the Feit–Thompson theorem [36] and that of the Kepler conjecture [40].

In this thesis, I will work with the Isabelle theorem prover, within which a (classical) higher-order logic has been implemented. Isabelle is known for its structured proof style, which leads to formal proofs that are both human and machine understandable, and its automation, which incorporates machine learning techniques and various automatic theorem provers [76]. Other than the projects mentioned earlier, there is also the fast-growing and well-maintained Archive of Formal Proofs (https://www.isa-afp.org), which, at the time of writing, contains about 1.7 million lines of code in Isabelle, contributed by 266 authors.

1.2 Why we need to justify algorithms in computer algebra

As the verification effort with interactive theorem proving continues, people realise the necessity for more formalised mathematics and verified calculations. For example, when carrying out verification projects in engineering and mathematics, it is common to encounter non-linear problems over the real numbers such as

$$\exists xy.\, x^2 - 2 = 1 \wedge xy = 1, \tag{1.1}$$

$$\forall x > 0. \frac{1-e^{-2x}}{2x(1-e^{-x})^2} - \frac{1}{x^2} \leq \frac{1}{12}. \tag{1.2}$$

Some of these problems are within reach of current computer algebra algorithms, and solvable by modern implementations including Z3 [28], Mathematica and MetiTarski [2]. However, those implementations are generally of gigantic size and with unclear mathematical semantics, as have been pointed out by Harrison and Théry [46]. For example, in recent version of Mathematica (i.e., Mathematica 10), we can type the following command:

$$\text{Simplify}\left[\frac{x^2-1}{x-1}, x \in \mathbb{R}\right]$$

to simplify the expression $(x^2-1)/(x-1)$ when $x \in \mathbb{R}$, and the result is, unsurprisingly, $x+1$. This behaviour does not appear mathematically valid when considering the case $x = 1$, and we may be baffled by the result due to lack of formal semantics. Therefore, when facing non-linear problems in an interactive theorem prover, we may be reluctant to deploy those sophisticated implementations, because they can severely reduce our confidence in the soundness of the overall verification framework. This dilemma has been noted by many authors who attempted to make use of sophisticated computer algebra algorithms in their verification projects [30, 69, 64, 35, 77].

The long-term objective of this thesis is to formalise relevant mathematics and algorithms using Isabelle, in order to soundly derive formulas like (1.1) and (1.2) within the logic. The main algorithm I will focus on is cylindrical algebraic decomposition (CAD) [22], which is known for its capability to tackle non-linear polynomial problems over the real numbers.

1.3 Thesis overview

This thesis demonstrates an ongoing effort to certify the CAD algorithm: during this process, various formalised theorems and verified procedures have been produced, which should benefit other verification projects as well.

Below I summarise the remainder of this thesis.

Chapter 2: Introduction to cylindrical algebraic decomposition (CAD). I will present a minimal introduction to Collins' CAD algorithm, including essential definitions and examples of how to calculate a CAD.

Chapter 3: The Sturm-Tarski theorem. I will describe a formal proof of the Sturm-Tarski theorem, which yields an effective way to reason with real roots of a polynomial: we can effectively compute the Tarski query and the Cauchy index using a polynomial remainder sequence.

Chapter 4: Real algebraic numbers. I will present a formalisation of real algebraic numbers in Isabelle/HOL. The formalisation is carried out at the abstraction and implementation level. At the abstraction level, I formalise real algebraic numbers as a subset of the real numbers and show that they form an ordered field following non-constructive proofs. At the implementation level, I utilise the Sturm-Tarski theorem to build sign determination procedures for a polynomial at a real algebraic point. Thus, I establish the executability of algebraic arithmetic.

Chapter 5: Deciding univariate polynomial problems using untrusted certificates. Based on previous sign determination procedures, I will present a tactic for univariate polynomial problems over the real numbers. The tactic is essentially a certified univariate CAD procedure but in a certificate-based manner (i.e., we sceptically invoke external programs and certify the results).

Chapter 6: A formal proof of Cauchy's residue theorem. To proceed to the multivariate case of CAD, I develop more mathematics in complex analysis. In this chapter, I will describe a formal proof of Cauchy's residue theorem along with proofs for two of its consequences: the argument principle and Rouché's theorem.

Chapter 7: Cauchy indices on the complex plane. Motivated by the difficulty of formally evaluating winding numbers, I build a tactic to evaluate winding numbers through Cauchy indices. By further exploiting the relationship between winding numbers and Cauchy indices, I build verified procedures to count complex roots of a polynomial inside a rectangle or a half-plane.

Chapter 8: Towards certifying multivariate CAD. With the objective of certifying multivariate CAD, I derived results towards the projection theorem of CAD. In this chapter, I will present the formal development and my rough ideas towards the multivariate case.

Chapter 9: Conclusion. I will present some reflections and conclude the thesis.

All definitions and lemmas in Chapter 3-8 have been mechanised (unless otherwise stated), and the code is available at the following URL:

https://doi.org/10.5281/zenodo.1306305

except for Chapter 6, where the mechanised proofs described have already been part of the Isabelle distribution (since Isabelle2016-1).

1.4 Publications

Chapters 3-6 in this thesis are based on publications during my PhD study:

- [57] Li, W. (2014). The Sturm-Tarski Theorem. *Archive of Formal Proofs*. http://isa-afp.org/entries/Sturm_Tarski.html, Formal proof development
- [62] Li, W. and Paulson, L. C. (2016b). A modular, efficient formalisation of real algebraic numbers. In Avigad, J. and Chlipala, A., editors, *Proceedings of the 5th ACM SIGPLAN Conference on Certified Programs and Proofs, CPP 2016*, pages 66–75, St. Petersburg, FL, USA. ACM
- [60] Li, W., Passmore, G. O., and Paulson, L. C. (2017). Deciding Univariate Polynomial Problems Using Untrusted Certificates in Isabelle/HOL. *Journal of Automated Reasoning*, 44(3):175–23
- [61] Li, W. and Paulson, L. C. (2016a). A formal proof of Cauchy's residue theorem. In Blanchette, J. C. and Merz, S., editors, *Proceedings of the 4th International Conference on Interactive Theorem Proving, ITP 2013*, pages 235–251, Nancy, France. Springer
- [59] Li, W. (2017b). Evaluate Winding Numbers through Cauchy Indices. *Archive of Formal Proofs*. http://isa-afp.org/entries/Winding_Number_Eval.html, Formal proof development
- [58]Li, W. (2017a). Count the Number of Complex Roots. *Archive of Formal Proofs*. http://isa-afp.org/entries/Count_Complex_Roots.html, Formal proof development

Some of them are with other authors, but the work presented in this thesis is principally mine, except when stated otherwise in the relevant chapters.

1.5 Contributions

Chapters 3-8 demonstrate my original formal proofs in Isabelle/HOL. Major contributions include:

- A novel formalisation of real algebraic numbers (Chapter 4). Compared to previous work, it is modular and has verified sign determination procedures that only require rational arithmetic.
- A novel tactic for univariate polynomial problems over the real numbers (Chapter 5). This tactic is based on univariate CAD and compares favourably to previous work.
- Novel formal proofs for Cauchy's residue theorem and two of its consequences – the argument principle and Rouché's theorem (Chapter 6). These results are fundamental in complex analysis and should pave the way for future development in this direction.

- A novel tactic to facilitate formal evaluation of winding numbers and novel verified procedures to count complex roots of a polynomial (Chapter 7).
- A novel formal proof towards the projection theorem of CAD (Chapter 8), which is believed crucial for the multivariate case (of CAD).

A minor contribution is the formal proof of the Sturm-Tarski theorem (Chapter 3), as the formalisation itself is not novel among theorem provers (i.e., there are similar formalisations in Coq and PVS). Nevertheless, this formalisation – the first in Isabelle – should still be beneficial to other Isabelle developments.

Chapter 2

Introduction to cylindrical algebraic decomposition

Ever since its introduction by George E. Collins [22] in the 1970s, CAD has become one of the most important algorithms to tackle non-linear first-order problems over the real numbers. CAD procedures have been incorporated into modern computer algebra systems and SMT solvers including Mathematica, Maple, Z3 [28] and QEPCAD [12].

Chapter outline. In this chapter I will present a minimal introduction to CAD starting with the basic idea (§2.1), followed by a classic algorithm to compute a CAD (§2.2). Finally, I will briefly discuss some related work (§2.3).

2.1 Basic idea

Given a set of polynomials $P \subseteq \mathbb{R}[x_1, x_2, \ldots, x_n]$, a CAD procedure can decompose $\mathbb{R}^n$ into disjoint connected cells, over each of which every $p \in P$ has constant sign (i.e., either positive, negative or zero).

For example, given $P = \{x_2^2 + x_1^2 - 3, x_2 - x_1^2/2\} \subseteq \mathbb{R}[x_1, x_2]$ (see Fig. 2.1), a CAD procedure can decompose $\mathbb{R}^2$ into disjoint connected cells $\mathfrak{D}$:

$$\mathfrak{D} = \{D_{1,1}, D_{1,2}, D_{1,3}, D_{2,1}, D_{2,2}, D_{2,3}, D_{2,4}, D_{2,5}, D_{3,1}, \ldots, D_{9,2}, D_{9,3}\}$$

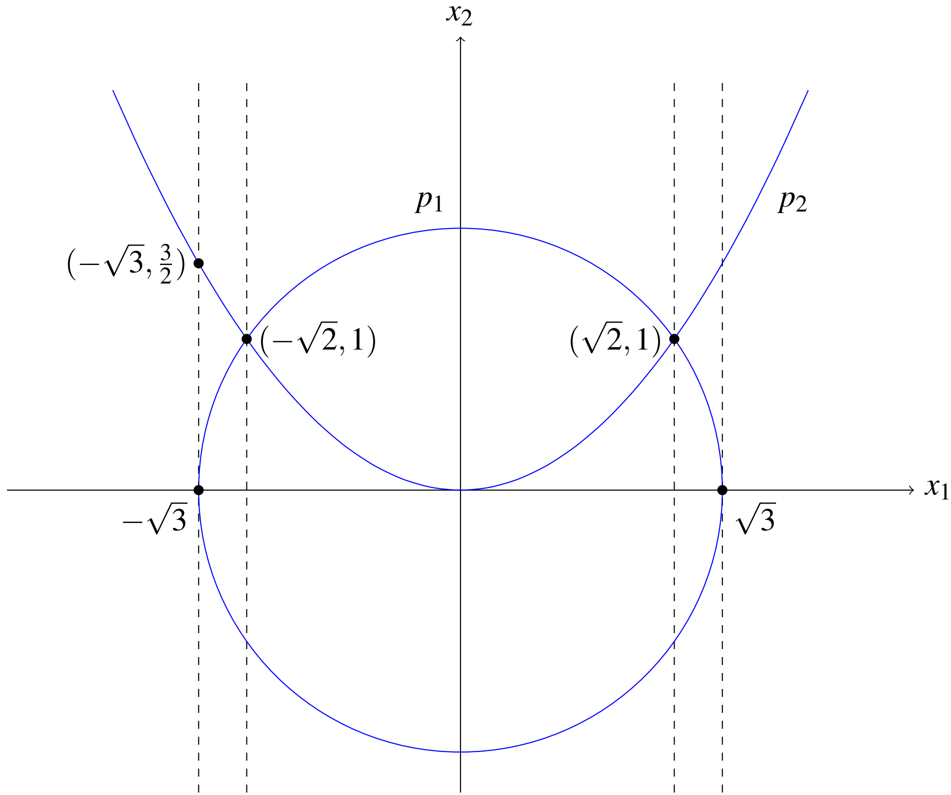

Fig. 2.1 Plot with $p_1(x_1,x_2) = x_2^2 + x_1^2 - 3 = 0$ and $p_2(x_1,x_2) = x_2 - x_1^2/2 = 0$

where

$$\begin{aligned}
D_{1,1} &= \{(x_1,x_2) \mid x_1 < -\sqrt{3} \wedge x_2 < x_1^2/2\} \\
D_{1,2} &= \{(x_1,x_2) \mid x_1 < -\sqrt{3} \wedge x_2 = x_1^2/2\} \\
D_{1,3} &= \{(x_1,x_2) \mid x_1 < -\sqrt{3} \wedge x_2 > x_1^2/2\} \\
D_{2,1} &= \{(x_1,x_2) \mid x_1 = -\sqrt{3} \wedge x_2 < 0\} \\
D_{2,2} &= \{(x_1,x_2) \mid x_1 = -\sqrt{3} \wedge x_2 = 0\} \\
D_{2,3} &= \{(x_1,x_2) \mid x_1 = -\sqrt{3} \wedge 0 < x_2 < 3/2\} \\
D_{2,4} &= \{(x_1,x_2) \mid x_1 = -\sqrt{3} \wedge x_2 = 3/2\} \\
D_{2,5} &= \{(x_1,x_2) \mid x_1 = -\sqrt{3} \wedge x_2 > 3/2\} \\
D_{3,1} &= \{(x_1,x_2) \mid -\sqrt{3} < x_1 < -\sqrt{2} \wedge x_2 < -\sqrt{3 - x_1^2}\} \\
&\vdots \\
D_{9,2} &= \{(x_1,x_2) \mid x_1 > \sqrt{3} \wedge x_2 = x_1^2/2\} \\
D_{9,3} &= \{(x_1,x_2) \mid x_1 > \sqrt{3} \wedge x_2 > x_1^2/2\}
\end{aligned}$$

such that

$$\bigcup \mathfrak{D} = \mathbb{R}^2$$

$$\forall X \in \mathfrak{D}. \forall Y \in \mathfrak{D}. X \neq Y \to X \cap Y = \emptyset$$

and both $p_1(x_1,x_2) = x_2^2 + x_1^2 - 3$ and $p_2(x_1,x_2) = x_2 - x_1^2/2$ have constant sign over every $X \in \mathfrak{D}$.

In practice, a CAD procedure usually produces a set of sample points drawn from every cell of the decomposition. In the case of $P = \{x_2^2 + x_1^2 - 3, x_2 - x_1^2/2\}$, we can have

$$\begin{array}{lcl}
(-2,0) & \in & D_{1,1} \\
(-2,2) & \in & D_{1,2} \\
(-2,4) & \in & D_{1,3} \\
(-\sqrt{3},-1) & \in & D_{2,1} \\
(-\sqrt{3},0) & \in & D_{2,2} \\
(-\sqrt{3},1) & \in & D_{2,3} \\
(-\sqrt{3},3/2) & \in & D_{2,4} \\
 & \vdots & \\
(2,4) & \in & D_{9,3}
\end{array}$$

which lead to the set of sample points $S \subseteq \mathbb{R}^2$ that represents $\mathfrak{D}$:

$$S = \{(-2,0),(-2,2),(-2,4),(-\sqrt{3},-1),(-\sqrt{3},0),(-\sqrt{3},1),(-\sqrt{3},3/2),\ldots,(2,4)\}.$$

Since both p_1 and p_2 have constant sign over each $X \in \mathfrak{D}$, a first-order sentence over p_1 and p_2 can be then decided by evaluating the sign of p_1 and p_2 at those sample points. For instance, to decide $\forall x_1 x_2.\, p_1(x_1,x_2) = 0 \wedge p_2(x_1,x_2) > 0$ we have

$$\begin{aligned}
&\forall x_1 x_2.\ p_1(x_1,x_2) = 0 \wedge p_2(x_1,x_2) > 0 \\
&= \forall (x_1,x_2) \in S.\, p_1(x_1,x_2) = 0 \wedge p_2(x_1,x_2) > 0 \\
&= (p_1(-2,0) = 0 \wedge p_2(-2,0) > 0) \wedge (p_1(-2,2) = 0 \wedge p_2(-2,2) > 0) \wedge \cdots \\
&\qquad \wedge (p_1(2,4) = 0 \wedge p_2(2,4) > 0) \\
&= \text{False}.
\end{aligned}$$

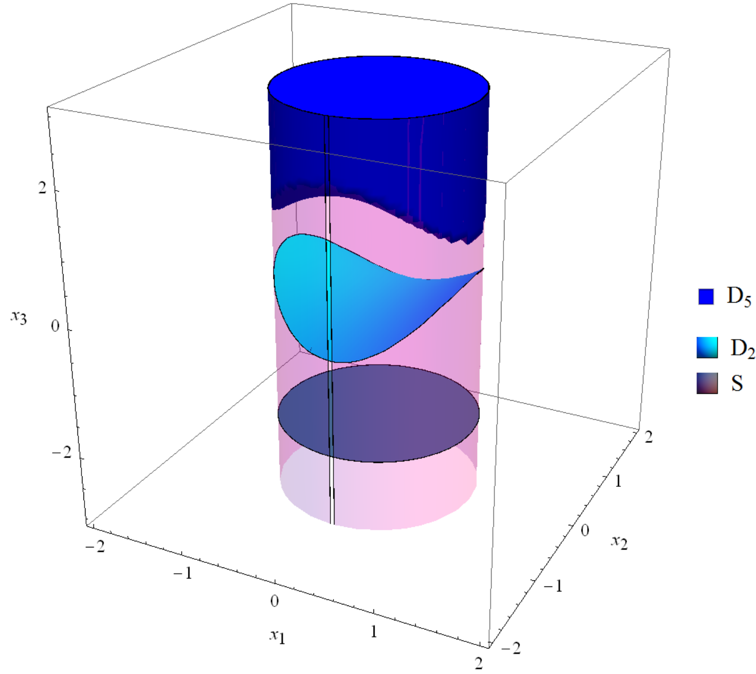

Fig. 2.2 Stack with $S = \{(x_1,x_2) \mid x_1^2 + x_2^2 \leq 1\}$, $D_2 = \{(x,x') \in S \times \mathbb{R} \mid x' = f_1(x)\}$ and $D_5 = \{(x,x') \in S \times \mathbb{R} \mid f_2(x) < x'\}$ where $f_1 = (x_1^2 - x_2^2)/2$ and $f_2 = (-x_1^2 + x_2^2 + 3)/2$

Similarly, $\exists x_1 x_2.\ p_1(x_1,x_2) = 0 \wedge p_2(x_1,x_2) = 0$ can be decided as follows:

$$\begin{aligned}
&\exists x_1 x_2.\, p_1(x_1,x_2) = 0 \wedge p_2(x_1,x_2) = 0 \\
&= \exists (x_1,x_2) \in S.\, p_1(x_1,x_2) = 0 \wedge p_2(x_1,x_2) = 0 \\
&= (p_1(-2,0) = 0 \wedge p_2(-2,0) > 0) \vee (p_1(-2,2) = 0 \wedge p_2(-2,2) > 0) \vee \cdots \\
&\qquad \vee (p_1(2,4) = 0 \wedge p_2(2,4) > 0) \\
&= \text{True}.
\end{aligned}$$

Even when quantifiers alternate (e.g., $\forall x_1.\exists x_2.\ \cdots$), with some modifications the approaches above still work.

To be more precise with CAD, we may need a few definitions.

Definition 2.1 (Stack)**.** A stack $\mathfrak{D} = \{D_1, D_2, \ldots, D_{2k+1}\}$ over $S \subseteq \mathbb{R}^n$ is a decomposition of the cylinder $S \times \mathbb{R}$ such that

- there is a sequence of continuous functions $f_0, f_1, \ldots, f_{k+1} : S \longrightarrow \mathbb{R}$, such that $f_0(x) < f_1(x) < \cdots < f_{k+1}(x)$ for all $x \in S$, $f_0(x) = -\infty$, $f_{k+1}(x) = +\infty$,
- $D_{2i+1} = \{(x,x') \in S \times \mathbb{R} \mid f_i(x) < x' < f_{i+1}(x)\}$, for $i = 0,1,\ldots,k$,
- $D_{2i} = \{(x,x') \in S \times \mathbb{R} \mid x' = f_i(x)\}$, for $i = 1,2,\ldots,k$.

Fig. 2.2 shows an example of a stack with $f_1 = (x_1^2 - x_2^2)/2$ and $f_2 = (-x_1^2 + x_2^2 + 3)/2$.

Definition 2.2 (Cylindrical)**.** A decomposition $\mathfrak{D}$ of $\mathbb{R}^n$ is cylindrical if

- $n = 1$, $\mathfrak{D}$ decomposes $\mathbb{R}$: there exist a finite number of points $a_i \in \mathbb{R}$ for $1 \leq i \leq k$, such that $a_i < a_{i+1}$ $(1 \leq i \leq k-1)$ and

$$\mathfrak{D} = \{(-\infty, a_1), \{a_1\}, (a_1, a_2), \{a_2\}, \ldots, (a_{k-1}, a_k), \{a_k\}, (a_k, \infty)\}.$$

- $n > 1$, there exists a cylindrical decomposition $\mathfrak{D}'$ of $\mathbb{R}^{n-1}$ such that over each $X \in \mathfrak{D}'$ there is a stack $t(X)$ and
$$\mathfrak{D} = \bigcup_{X \in \mathfrak{D}'} t(X).$$

Here, a decomposition $\mathfrak{D}$ is said to be a CAD if it is cylindrical and each cell $X \subseteq \mathbb{R}^n \in \mathfrak{D}$ is semi-algebraic (i.e., can be constructed by sets of the form $\{x \mid p(x) \geq 0, p \in \mathbb{R}[x_1, x_2, \ldots, x_n]\}$ with finitely many applications of union, intersection and complementation). Moreover, the sign-invariant property is more commonly referred to 'adapted' in CAD terminology:

Definition 2.3 (Decomposition adapted to P)**.** A decomposition $\mathfrak{D}$ is adapted to a set of polynomials P, if every polynomial $p \in P$ has constant sign over every cell $D \in \mathfrak{D}$.

2.2 The classic algorithm

In the previous section, the basic ideas of CAD were covered as well as how it can be used to solve non-linear first-order sentences over the real numbers. In this section, I will briefly talk about the classic algorithm to compute (sample points of) a CAD given a set of polynomials.

Before understanding the algorithm for CAD calculation, we may need to know the subresultant operation.

Definition 2.4 (Subresultants)**.** Let $p(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_0$ and $q(x) = b_m x^m + b_{m-1} x^{m-1} + \cdots + b_0$ be univariate polynomials of degree n and m respectively. The k-th

(principal) subresultant $\mathrm{sRes}_k(p,q)$ is defined as the determinant

$$\mathrm{sRes}_k(p,q) = \begin{vmatrix} a_n & & & & b_m & & & & \\ a_{n-1} & a_n & & & b_{m-1} & b_m & & & \\ \vdots & & \ddots & & \vdots & & \ddots & & \\ a_{n-m+k+1} & \cdots & \cdots & a_n & b_{k+1} & \cdots & \cdots & b_m & \\ \vdots & & & \vdots & \vdots & & & & \ddots \\ a_{k+1} & \cdots & \cdots & a_m & b_{m-n+k+1} & \cdots & \cdots & \cdots & \cdots & b_m \\ \vdots & & & \vdots & \vdots & & & & & \vdots \\ \vdots & & & \vdots & \vdots & & & & & \vdots \\ \underbrace{a_{2k-m+1} \quad \cdots \quad \cdots \quad a_k}_{\text{m-k}} & & & & \underbrace{b_{2k-n+1} \quad \cdots \quad \cdots \quad \cdots \quad \cdots \quad b_k}_{\text{n-k}} \end{vmatrix}$$

where $k \leq \min(\deg(p), \deg(q))$ and $a_j, b_j = 0$ if $j < 0$.

Note, sRes_0 is also referred as the resultant. Moreover, we have the following interesting property about subresultants:

Theorem 2.5. *Polynomials p and q have a common factor of degree k ($0 < k \leq \min(\deg(p), \deg(q))$) if and only if $sRes_j(p,q) = 0$ for $0 \leq j \leq k-1$ and $sRes_k(p,q) \neq 0$.*

Here is an example of Theorem 2.5: let $p(x) = (x-1)(x-2)(x-3) = x^3 - 6x^2 + 11x - 6$ and $q(x) = (x-2)(2x-1) = 2x^2 - 5x + 2$. We have

$$\begin{aligned} \mathrm{sRes}_0(p,q) &= \begin{vmatrix} 1 & & 2 & & \\ -6 & 1 & -5 & 2 & \\ 11 & -6 & 2 & -5 & 2 \\ -6 & 11 & & 2 & -5 \\ & -6 & & & 2 \end{vmatrix} = 0 \\ \mathrm{sRes}_1(p,q) &= \begin{vmatrix} 1 & 2 & \\ -6 & -5 & 2 \\ 11 & 2 & -5 \end{vmatrix} = 5 \\ \mathrm{sRes}_2(p,q) &= \begin{vmatrix} 2 \end{vmatrix} = 2 \end{aligned}$$

and the degree of the common factor of p and q is $\deg(\gcd(p,q)) = \deg(x-2) = 1$, as claimed by Theorem 2.5.

To have a *cylindrical* decomposition, we need the following theorem.

Theorem 2.6. *Let $P \subseteq \mathbb{R}[x_1, \ldots, x_{n-1}][x_n]$ be a set of polynomials and C be a connected subset of $\mathbb{R}^{n-1}$. If*

(i) *for every $p \in P$, the total number of complex roots (counting multiplicities) of $p(\beta, x)$ is constant as β varies over C, where $p(\beta, x)$ is a univariate polynomial in which the variables $x_1, \dots, x_{n-1}$ are instantiated by $\beta \in \mathbb{R}^{n-1}$,*

(ii) *for every $p \in P$, the number of distinct complex roots of $p(\beta, x)$ is constant as β varies over C,*

(iii) *for every $p, q \in P$, the total number of common complex roots (counting multiplicities) of $p(\beta, x)$ and $q(\beta, x)$ is constant as β varies over C,*

then the total number of distinct real roots of $(\prod P)(\beta, x)$ is constant as β varies over C.

Theorem 2.6 is important as we can have a stack over C that is adapted to P only when $(\prod P)(\beta, x)$ has a constant number of distinct real roots (as β varies over C), in which we can let $f_j(\beta)$ be the j-th root of $(\prod P)(\beta, x)$ such that $f_j(\beta) < f_{j+1}(\beta)$ for all $\beta \in C$.

In the classic CAD algorithm, we will also need the following truncation operation on a set of polynomials:

Definition 2.7 (Truncation of polynomials)**.** For a set of polynomials $P \subseteq \mathbb{R}[x_1, \dots, x_{n-1}][x_n]$, the truncation of it with respect to x_n, $\mathrm{Tru}_{x_n}(P)$, is defined as

$$
\begin{aligned}
&\mathrm{Tru}(\{\}) = \emptyset \\
&\mathrm{Tru}(\{a_0, p_1, \dots, p_m\}) = \mathrm{Tru}(\{p_1, \dots, p_m\}) \\
&\mathrm{Tru}(\{a_k x^k + a_{k-1} x^{k-1} + \dots + a_0, p_1, \dots, p_m\}) \\
=&\{a_k x^k + a_{k-1} x^{k-1} + \dots + a_0\} \cup \mathrm{Tru}(\{a_{k-1} x^{k-1} + \dots + a_0, p_1, \dots, p_m\}).
\end{aligned}
$$

For example, the truncation of $\{(x_1+1)x_2^3 + 2x_2^2 + 3x_2 + 4, 5x_2 + 6\}$ with respect to x_2 is given by:

$$
\begin{aligned}
&\mathrm{Tru}_{x_2}(\{(x_1+1)x_2^3 + 2x_2^2 + 3x_2 + 4, 5x_2 + 6\}) \\
=&\{(x_1+1)x_2^3 + 2x_2^2 + 3x_2 + 4, 2x_2^2 + 3x_2 + 4, 3x_2 + 4, 5x_2 + 6\}.
\end{aligned}
$$

We can now come to the classic CAD procedure as illustrated in Algorithm 1, which takes a set of polynomials $P \subseteq \mathbb{R}[x_1, \dots, x_n]$ and returns a set of sample points $S_n \subseteq \mathbb{R}^n$ from each cell of a CAD adapted to P. In particular, lines 3-10 are commonly referred as the *projection phase*, where for any $Q \subseteq \mathbb{R}[x_1, \dots, x_{k-1}, x_k]$ we have $\mathrm{proj}(Q) \subseteq \mathbb{R}[x_1, \dots, x_{k-1}]$, such that

1. if $C \subseteq \mathbb{R}^{k-1}$ is a region over which each $p \in \mathrm{proj}(Q)$ has constant sign,

Algorithm 1 The classic CAD procedure

Require: a finite set of polynomials $P \subseteq \mathbb{R}[x_1,\ldots,x_n]$
Ensure: Return a set of sample points $S_n \subseteq \mathbb{R}^n$ from each cell of a CAD adapted to P

1: **procedure** CAD(P)
2: $P_n \leftarrow P$
3: **for** $i = n$ to 2 **do** ▷ Projection phase, where $P_i \subseteq \mathbb{R}[x_1,\ldots,x_i]$
4: $P_{i-1} \leftarrow \mathrm{proj}(P_i)$
5: where
6: $\mathrm{proj}_1(Q) = \{\mathrm{lcof}_{x_i}(p) \mid p \in \mathrm{Tru}_{x_i}(Q)\}$
7: $\mathrm{proj}_2(Q) = \{\mathrm{sRes}_j(p, \frac{\partial p}{\partial x_i}) \mid j = 0,\ldots,\deg_{x_i}(p)-1, p \in \mathrm{Tru}_{x_i}(Q)\}$
8: $\mathrm{proj}_3(Q) = \{\mathrm{sRes}_j(p,q) \mid j = 0,\ldots,\min(\deg_{x_i}(p),\deg_{x_i}(q)), p,q \in \mathrm{Tru}_{x_i}(Q)\}$
9: $\mathrm{proj}(Q) = \mathrm{proj}_1(Q) \cup \mathrm{proj}_2(Q) \cup \mathrm{proj}_3(Q)$
10: **end for**
11: $S_1 \leftarrow \mathrm{base}(P_1)$ ▷ Base case
12: where $\mathrm{base}(Q)$ returns a set of sample points adapted to $Q \subseteq \mathbb{R}[x]$
13: **for** $i = 1$ to $n-1$ **do** ▷ Lifting phase, where $S_i \subseteq \mathbb{R}^i$
14: $S_{i+1} \leftarrow \bigcup_{\beta \in S_i}(\{\beta\} \times \mathrm{base}(P_{i+1}(\beta,x)))$
15: **end for**
16: **return** S_n
17: **end procedure**

2. then the total number of distinct real roots of $(\prod Q)(\beta,x)$ is constant as β varies over C.

To understand the projection operation, recall that a CAD is a recursive structure, (1) indicates that C is a cell of a CAD of $\mathbb{R}^{k-1}$ adapted to $\mathrm{proj}(Q)$, and (2) indicates that we can construct a stack over C adapted to Q. Overall, the projection step corresponds to the $n > 1$ case in Definition 2.2. As for the projection operator $\mathrm{proj}(-)$, we can employ Theorem 2.6:

- The first assumption of Theorem 2.6 is guaranteed by

$$\mathrm{proj}_1(Q) = \{\mathrm{lcof}_{x_i}(p) \mid p \in \mathrm{Tru}_{x_i}(Q)\}$$

  where $\mathrm{lcof}_{x_i}(P)$ is the leading coefficient of P with respect to variable x_i. This is because the number of complex roots of a polynomial is determined by its degree, which is further determined by whether its leading coefficient with respect to x_i (i.e., lcof_{x_i}) is zero.

- The second assumption is guaranteed by $\mathrm{proj}_1(Q)$ and

$$\mathrm{proj}_2(Q) = \{\mathrm{sRes}_j(p, \frac{\partial p}{\partial x_n}) \mid j = 0,\ldots,\deg_{x_i}(p)-1, p \in \mathrm{Tru}_{x_i}(Q)\}$$

since for a polynomial with a fixed degree, the number of distinct complex roots is determined by the degree of the greatest common divisor between it and its first derivative, and that degree is further determined by the subresultants between them (i.e., Theorem 2.5).

- The third assumption is guaranteed by $\mathrm{proj}_1(Q)$ and

$$\mathrm{proj}_3(Q) = \{\mathrm{sRes}_j(p,q) \mid j = 0,\ldots,\min(\deg_{x_i}(p),\deg_{x_i}(q)), p,q \in \mathrm{Tru}_{x_i}(Q)\}$$

since the total number of common complex roots of p and q is determined by the degree of their greatest common divisor, which, again, is determined by the subresultants.

Therefore, combining proj_1, proj_2 and proj_3 we can have the projection operator as desired (i.e., line 9).

After the projection phase, we will reach the *base case* (i.e., lines 11-12 of Algorithm 1) where we need to extract sample points from a CAD that is adapted to a set of univariate polynomials $P_1 \subseteq \mathbb{R}[x]$. In this case, the sample points are real roots of $\prod P_1$ plus samples from intervals delimited by those roots.

Finally, it is the *lifting phase* (i.e., lines 13-15 of Algorithm 1): for each $1 \leq i \leq n-1$, given a set of sample points $S_i \subseteq \mathbb{R}^i$ and a set of polynomials $P_{i+1} \subseteq \mathbb{R}[x_1,\ldots,x_{i+1}]$, the lifting step will construct another set of points $S_{i+1} \subseteq \mathbb{R}^{i+1}$ that represents a decomposition adapted to P_{i+1}. The idea is to use $\beta \in S_i$ to instantiate variables $x_1,\ldots,x_k$ in each polynomial of P_{i+1}, and this leads to a set of univariate polynomials $P_{i+1}(\beta,x)$ over which we can apply the base operation. Here, $\{\beta\} \times \mathrm{base}(P_{i+1}(\beta,x)) \subseteq \mathbb{R}^{i+1}$ are ,essentially, sample points from the stack over a set represented by β.

Consider the example in Fig. 2.1, where we have $P = \{x_2^2 + x_1^2 - 3, x_2 - x_1^2/2\}$ and

$$\begin{array}{rcl}
\mathrm{Tru}_{x_2}(P) & = & \{x_2^2 + x_1^2 - 3, x_2 - x_1^2/2\} \\
\mathrm{proj}_1(P) & = & \{\mathrm{lcof}_{x_2}(p) \mid p \in \mathrm{Tru}_{x_2}(P)\} \\
 & = & \{1\} \\
\mathrm{proj}_2(P) & = & \{\mathrm{sRes}_j(p, \frac{\partial p}{\partial x_2}) \mid j = 0,\ldots,\deg_{x_2}(p) - 1, p \in \mathrm{Tru}_{x_2}(P)\} \\
 & = & \{4x_1^2 - 12, 2, 1\} \\
\mathrm{proj}_3(P) & = & \{\mathrm{sRes}_j(p,q) | j = 0,\ldots,\min(\deg_{x_2}(p),\deg_{x_2}(q)), p,q \in \mathrm{Tru}_{x_2}(P)\} \\
 & = & \{x_1^4/4 + x_1^2 - 3, 1\}.
\end{array}$$

Hence,

$$\mathrm{proj}(P) = \{x_1^4/4 + x_1^2 - 3, 4x_1^2 - 12, 2, 1\}$$

With the base step, we can have sample points from a decomposition adapted to $\mathrm{proj}(P)$:

$$S_1 = \mathrm{base}(\mathrm{proj}(P)) = \{-2, -\sqrt{3}, -\frac{3}{2}, -\sqrt{2}, 0, \sqrt{2}, \frac{3}{2}, \sqrt{3}, 2\}$$

where $\pm\sqrt{2}$ and $\pm\sqrt{3}$ are real roots of polynomials from $\mathrm{proj}(P)$. Subsequently, we enter the lifting phase of Algorithm 1:

$$\begin{array}{lcl}
P(-2,x_2) = & = & \{x_2^2+1, x_2-2\} \\
\mathrm{base}(P(-2,x_2)) & = & \{0,2,4\} \\
\{-2\} \times \mathrm{base}(P(-2,x_2)) & = & \{(-2,0),(-2,2),(-2,4)\} \\
 & & \\
P(-\sqrt{3},x_2) & = & \{x_2^2, x_2-3/2\} \\
\mathrm{base}(P(-\sqrt{3},x_2)) & = & \{-1,0,1,3/2,2\} \\
-\sqrt{3} \times \mathrm{base}(P(-\sqrt{3},x_2)) & = & \{(-\sqrt{3},-1),(-\sqrt{3},0),(-\sqrt{3},1),(-\sqrt{3},3/2),(-\sqrt{3},2)\} \\
 & \vdots & \\
2 \times \mathrm{base}(P(2,x_2)) & = & \{(2,0),(2,2),(2,4)\}
\end{array}$$

combining which yields

$$\begin{aligned}
S_2 &= \bigcup_{\beta \in S_1} (\{\beta\} \times \mathrm{base}(P(\beta,x))) \\
&= \{(-2,0),(-2,2),(-2,4),(-\sqrt{3},-1),(-\sqrt{3},0),(-\sqrt{3},1),(-\sqrt{2},3/2),\ldots,(2,4)\},
\end{aligned}$$

which are sample points from each cell of a CAD adapted to P.

2.3 Remarks

Algorithm 1 in the previous section is a very basic CAD-computing procedure adapted from textbooks [8, Chapter 5;70, Chapter 8;52]. Modern CAD implementations have incorporated numerous improvements including but not limited to:

- Better projection operations [68, 49, 11], which lead to fewer polynomials after each projection iteration.
- Partial CAD [23], which aborts in the lifting phase early if the produced sample points are sufficient to decide the truth value of the target sentence.
- Validated numerics [82] or field extensions [29], which reduce costly exact algebraic arithmetic in the lifting phase of CAD.

- Better ordering on the variables in the projection phase [50], as it has been shown that variable ordering can have a significant impact on the computational complexity of a CAD procedure [13].
- CAD through regular chains [17], which is a vastly different way to compute a CAD (i.e., other than the traditional projection-lifting style).

Since the purpose of this thesis is mainly about verification rather efficient computation, in the following chapters I will mostly focus on simple formulations like Algorithm 1 and leave sophisticated optimisations to future work.

Chapter 3

The Sturm-Tarski theorem

The Sturm-Tarski theorem (also referred as Tarski's theorem [8, Theorem 2.61]) was first formulated by Alfred Tarski [15, p. 24] to show that the elementary theory of real closed fields admits quantifier elimination. A formal proof of this theorem is of great importance to us as it leads to an effective way to manipulate real roots of a polynomial, which are ubiquitous in a CAD procedure.

This formalisation mainly follows Basu et al.'s book [8, Theorem 2.61] and Cohen's PhD thesis [21, p. 119].

Chapter outline. This chapter starts with a formulation of the Sturm-Tarski theorem (§3.1), followed by a formal proof (§3.2) and some remarks (§3.3).

3.1 Formulation

We abbreviate $\mathbb{R} \cup \{-\infty, \infty\}$ as $\overline{\mathbb{R}}$, the extended real numbers.

Definition 3.1 (Tarski Query)**.** The Tarski query $\mathrm{TaQ}(q, p, a, b)$ is

$$\mathrm{TaQ}(q, p, a, b) = \sum_{x \in (a,b), p(x)=0} \mathrm{sgn}(q(x))$$

where $a, b \in \overline{\mathbb{R}}$, $p, q \in \mathbb{R}[X]$, $p \neq 0$ and $\mathrm{sgn} : \mathbb{R} \to \{-1, 0, 1\}$ is the sign function.

The Sturm-Tarski theorem is essentially an effective way to compute Tarski queries through some remainder sequences:

Theorem 3.2 (Sturm-Tarski)**.** *The Sturm-Tarski theorem states*

$$\mathrm{TaQ}(q, p, a, b) = \mathrm{Var}(\mathrm{SRemS}(p, p'q); a, b)$$

where $p \neq 0$*,* $p, q \in \mathbb{R}[X]$*,* p' *is the first derivative of p,* $a, b \in \overline{\mathbb{R}}$*,* $a < b$ *and are not roots of p,* $\mathrm{SRemS}(p, p'q)$ *is the signed remainder sequence of p and* $p'q$*, and*

$$\begin{aligned} &\mathrm{Var}([p_0, p_1, ..., p_n]; a, b) \\ &= \mathrm{Var}([p_0(a), p_1(a), ..., p_n(a)]) - \mathrm{Var}([p_0(b), p_1(b), ..., p_n(b)]) \end{aligned}$$

is the difference in the number of sign variations (after removing zeroes) in the polynomial sequence $[p_0, p_1, ..., p_n]$ *evaluated at a and b.*

Note that the more famous Sturm's theorem, which counts the number of distinct real roots (of a univariate polynomial) within an interval, is a special case of the Sturm-Tarski theorem when $q = 1$.

3.2 A formal proof of the Sturm-Tarski theorem

The core idea of the formal proof is built around a concept called the *Cauchy index*, which I will discuss more in Chapter 7. The Cauchy index is usually defined using the jump function:

Definition 3.3. Given $p, q \in \mathbb{R}[x]$ and $x \in \mathbb{R}$, $\text{jump_poly}(p, q, x)$ is defined as

$$\text{jump_poly}(p,q,x) = \begin{cases} -1 & \text{if } \lim_{u \to x^-} \frac{q(u)}{p(u)} = \infty \text{ and } \lim_{u \to x^+} \frac{q(u)}{p(u)} = -\infty, \\ 1 & \text{if } \lim_{u \to x^-} \frac{q(u)}{p(u)} = -\infty \text{ and } \lim_{u \to x^+} \frac{q(u)}{p(u)} = \infty, \\ 0 & \text{otherwise.} \end{cases}$$

For example, let $q(x) = x - 4$ and $p(x) = (x-3)(x-1)^2(x+1)$. The graph of q/p is shown in Fig. 3.1. We have

$$\text{jump_poly}(p,q,x) = \begin{cases} 1 & \text{when } x = -1, \\ -1 & \text{when } x = 3, \\ 0 & \text{otherwise.} \end{cases}$$

The Cauchy index `cindex_poly a b q p` is the sum of jump_polys over the interval (a, b):

```
definition cindex_poly:: "real ⇒ real ⇒ real poly ⇒ real poly ⇒ int"
  where
 "cindex_poly a b q p≡ (∑x∈{x. poly p x=0 ∧ a < x ∧ x < b}. jump_poly q p x)"
```

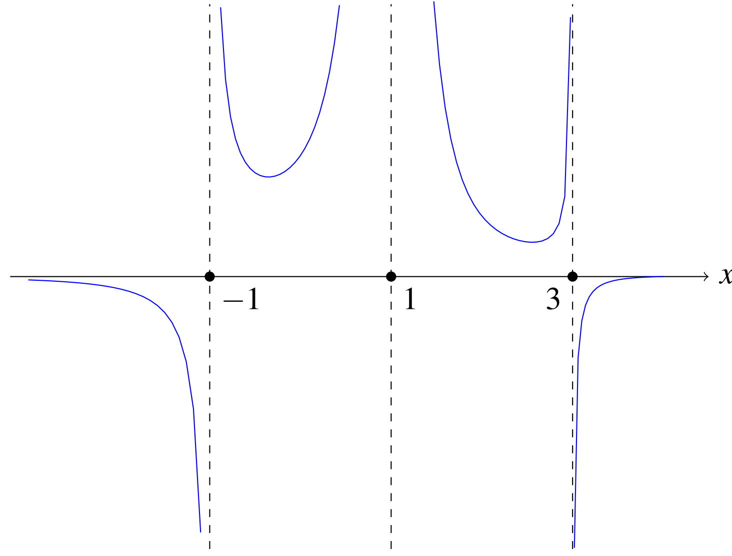

Fig. 3.1 Plot of the rational function $(x-4)/((x-3)(x-1)^2(x+1))$

where `{x. poly p x = 0 ∧ a < x ∧ x < b}` is a subset of reals such that for each x in this set, we have $p(x) = 0$, $a < x$ and $x < b$. The reason we sum over this set is that $\mathrm{jump_poly}(p,q,x) = 0$ if $p(x) \neq 0$, and the set can be shown to be finite (provided $p \neq 0$), which makes the sum well-defined.

The overall structure of the formal proof can be divided into two parts:

(a) The Tarski query is equal to some Cauchy index.

(b) The Cauchy index can be effectively calculated through some remainder sequences.

For (a), we can establish the following equality relation between the Tarski query and the Cauchy index:

Lemma 3.4 (`cindex_poly_taq`).

```
fixes p q::"real poly" and a b::real
shows "taq {x. poly p x = 0 ∧ a < x ∧ x < b} q
        = cindex_poly a b (pderiv p * q) p"
```

where `taq` is the definition of the Tarski query in Isabelle (following Definition 3.1):

```
definition taq :: "'a::linordered_idom set ⇒ 'a poly ⇒ int" where
  "taq s q = (∑x∈s. sign (poly q x))"
```

and `pderiv p` is the first derivative of `p`.

To derive (b), we first relate the Cauchy index to Euclidean division (`mod`) on a recurrence relation:

Lemma 3.5 (*cindex_poly_rec*).

```
fixes p q::"real poly" and a b::real
assumes "a < b" and "poly (p * q) a ≠0" and "poly (p * q) b ≠0"
shows "cindex_poly a b q p = cross (p * q) a b
        + cindex_poly a b (- (p mod q)) q"
```

where

$$\text{cross } p\ a\ b = \begin{cases} 0 & \text{if } p(a)p(b) \geq 0, \\ 1 & \text{if } p(a)p(b) < 0 \text{ and } p(a) < p(b), \\ -1 & \text{if } p(a)p(b) < 0 \text{ and } p(a) \geq p(b). \end{cases}$$

We can find out a similar recurrence relation holding for sign variations of the signed remainder sequences (*changes_itv_smods*):

Lemma 3.6 (*changes_itv_smods_rec*).

```
fixes p q::"real poly" and a b::real
assumes "a < b" and "poly (p * q) a ≠ 0" and "poly (p * q) b ≠ 0"
shows "changes_itv_smods a b p q = cross (p * q) a b
        + changes_itv_smods a b q (- (p mod q))"
```

where *changes_itv_smods* is defined as

```
definition changes_itv_smods::
    "real ⇒ real ⇒ real poly ⇒ real poly ⇒ int" where
  "changes_itv_smods a b p q = (
      let
        ps = smods p q
      in
        changes_poly_at ps a - changes_poly_at ps b)"
```

and the signed remainder sequence (*smods*) is defined as

```
function smods:: "real poly ⇒ real poly ⇒ (real poly) list" where
  "smods p q= (if p = 0 then
                 []
               else
                 p # (smods q (-(p mod q))))"
```

and *changes_poly_at ps a* returns the number of sign changes when evaluating a list of polynomials (*ps*) at *a*.

Combining Lemma 3.5 with Lemma 3.6 yields an effective method to compute the Cauchy index *cindex_poly*:

Lemma 3.7 (*cindex_poly_changes_itv_mods*).

```
fixes p q::"real poly" and a b::real
assumes "a < b" and "poly p a ≠ 0" and "poly p b ≠ 0"
shows "cindex_poly a b q p = changes_itv_smods a b p q"
```

Finally, putting Lemma 3.4 and 3.7 together leads to an effective way to compute the Tarski query (i.e., the Sturm-Tarski theorem):

Theorem 3.8 (*sturm_tarski_interval*).

```
fixes p q::"real poly" and a b::real
assumes "a < b" and "poly p a ≠ 0" and "poly p b ≠ 0"
shows "taq {x. poly p x = 0 ∧ a < x ∧ x < b} q
    = changes_itv_smods a b p (pderiv p * q)"
```

Note, this is just the bounded case of the Sturm-Tarski theorem. Proofs for the unbounded and half-bounded cases are similar.

3.3 Remarks

The Sturm-Tarski theorem is of great theoretical importance, as Alfred Tarski [15, p. 24] used it to build the first complete decision procedure for first-order sentences over the real numbers. Prior to CAD, Tarski's method (along with some of its variations) was the only approach for such a task. However, people later realised that Tarski's method is impractical due to its non-elementary complexity (i.e., the complexity is a tower of exponents whose height is linear to the number of variables in the sentence), while Collins' CAD procedure is only of double exponential complexity [8, Chapter 11].

Despite being the first in Isabelle/HOL, my formalisation of the Sturm-Tarski theorem is not novel in the theorem prover community: prior work includes a formalisation by Mahboubi and Cohen in Coq [65] and another one by Narkawicz et al. in PVS [71]. The main difference between my work and theirs is about the purpose: they formalise the Sturm-Tarski theorem to build a verified quantifier elimination procedure similar to Tarski's, while I use it for sign determination at real algebraic points (in Chapter 4) and complex root counting (in Chapter 7).

Chapter 4

Real algebraic numbers

As has been demonstrated in the examples in Chapter 2, sample points from a CAD may contain real algebraic numbers such as $\sqrt{2}$ and $\sqrt{3}$. Those numbers are defined as particular roots of non-zero polynomials with rational (or integer) coefficients. They are important in computer algebra as each one can be encoded precisely (unlike most real numbers), and their arithmetic and comparison operations are decidable. In this chapter, I will cover how to represent, reason and compute with those real algebraic numbers in Isabelle/HOL.

The overall formalisation follows Isabelle's tradition of separation of abstraction and implementation. That is,

- **abstraction**: I first formalise real algebraic numbers on an abstract level without considering executability (see §4.1). More specifically, I formalise real algebraic numbers as a subset of real numbers, and show them to form an ordered field using classic proofs in abstract algebra.
- **implementation (restoring executability)**: I then establish executability on real algebraic numbers (see §4.2). More specifically, I define a pseudo constructor for algebraic real numbers and prove code equations for algebraic arithmetic on this constructor. Some of the code equations for algebraic arithmetic are based on a verified decision procedure to decide the sign of a bivariate polynomial with rational coefficients at real algebraic points.

Chapter outline. The chapter continues as follows. The first component of the modular design is the abstract specification of real algebraic numbers (§4.1), which is then followed by an implementation (§4.2) in the form of Isabelle/HOL code equations. In particular, the

The content of this chapter is adapted from a previous publication [62].

implementation is concerned with deciding the signs of polynomials at a given real algebraic point. Experiments (§4.3) is then described along with related work (§4.4). Finally, some remarks (§4.5) are given about limitations and applications of this work.

4.1 Construction on an abstract level

This section presents my formalisation of real algebraic numbers as an abstract data type. Definitions on this level will be as abstract as possible without considering executability.

Mathematically, a real algebraic number α is a real number for which there exists a non-zero univariate polynomial $p(x)$ with integer (or rational) coefficients, such that $p(x) = 0$ when $x = \alpha$.

It is then straightforward to define the real algebraic numbers as a subset of the real numbers. We can formalise this construction by defining type alg on the top of type real using the **typedef** command[1]:

```
typedef alg = "{x::real. ∃p::int poly. p ≠ 0 ∧ poly (of_int_poly p) x = 0}"
```

where *of_int_poly* converts coefficients of a polynomial from *int* to *real*, and *poly p x* means evaluating polynomial p at x.

To prove non-trivial properties about real algebraic numbers, we need at least to prove that they are closed under the basic arithmetic operations and hence form a field. For example, to show that real algebraic numbers are closed under addition, suppose we have two real algebraic numbers α and β, given by polynomials p and q:

$$\alpha \in \mathbb{R}, p \in \mathbb{Z}[x] \qquad p \neq 0 \wedge p(\alpha) = 0$$

$$\beta \in \mathbb{R}, q \in \mathbb{Z}[x] \qquad q \neq 0 \wedge q(\beta) = 0$$

Then we have to show that

$$\exists r \in \mathbb{Z}[x].\ r \neq 0 \wedge r(\alpha + \beta) = 0. \tag{4.1}$$

One way to show this is to compute r constructively using resultants as in Cyril Cohen's proof in Coq [20]. However, as we are working on an abstract level and not concerned with executability, a non-constructive but usually simpler proof (to show the mere existence of such a polynomial) seems more appealing. Therefore, I decided to follow a classic proof in abstract algebra.

[1] A description of this command can be found in the Tutorial [73, §8.5.2]

Definition 4.1 (vector space)**.** A vector space V over a field F is an abelian group associated with scalar multiplication αv for all $\alpha \in F$ and $v \in V$, satisfying the standard additivity and identity axioms.

In Isabelle/HOL, the notion of a vector space is formalised using a `locale`:

```
locale vector_space =
  fixes scale :: "'a::field ⇒ 'b::ab_group_add ⇒ 'b"
  assumes "scale a (x + y) = scale a x + scale a y"
  and "scale (a + b) x = scale a x + scale b x"
  and "scale a (scale b x) = scale (a * b) x"
  and "scale 1 x = x"
```

where `scale::'a ⇒'b ⇒'b` denotes scalar multiplication for `'a` a field and `'b` an abelian group.

The standard library development of vector spaces has been extended by Jose Divasón and Jesús Aransay in their formalisation of the Rank-Nullity Theorem in linear algebra, including definitions of *span* and of *linearly dependent* [31].

Definition 4.2 (Span)**.** Let $S = \{v_1, v_2, ..., v_n\}$ be a set of vectors in a vector space, then span(S) is defined as

$$\{w \mid w = a_1 v_1 + a_2 v_2 + \cdots + a_n v_n, \text{and } a_1, \ldots, a_n \text{ are scalars}\}$$

Divasón and Aransay [31] formalise `span` slightly differently, but the following lemma can be considered as an alternative definition that matches standard mathematical definitions:

Lemma 4.3 (`span_explicit`)**.**

```
"span P = {y. ∃S u. finite S ∧ S ⊆ P ∧ sum (λv. scale (u v) v) S = y}"
```

where `u` of type `'b ⇒'a` maps each vector in `S` to the corresponding scalar. And `sum (λv. scale (u v) v) S` maps each element in `S` using `(λv. scale (u v) v)` and sums the results.

Definition 4.4 (linearly dependent)**.** Let $S = \{v_1, v_2, \ldots, v_n\}$ be a set of vectors in a vector space, we say S is *linearly dependent* if there exist scalars $a_1, a_2, ..., a_n$, at least one of which is non-zero, such that

$$a_1 v_1 + a_2 v_2 + \cdots + a_n v_n = \mathbf{0}$$

Divasón and Aransay formalise `dependent` as

```
definition "dependent S ⟷ (∃a ∈ S. a ∈ span (S - {a}))"
```

since

$$a_1v_1 + a_2v_2 + \cdots + a_nv_n = \mathbf{0} \longleftrightarrow v_n = \frac{a_1}{-a_n}v_1 + \frac{a_2}{-a_n}v_2 + \cdots + \frac{a_{n-1}}{-a_n}v_{n-1}$$

assuming a_n is the non-zero scalar.

Now, back to the problem of showing (4.1), we can consider the vector space of reals with rational scalars:

interpretation `rat: vector_space`
  `"(λx y. (of_rat x * y))::rat ⇒ real ⇒ real"`

where `of_rat :: rat ⇒ real` embeds `rat` into `real` and the `scale` function in `vector_space` is instantiated as

```
(λx y. (of_rat x * y))::rat ⇒ real ⇒ real
```

After the interpretation, we have new constants, such as

```
rat.span :: real set ⇒ real set  and
rat.dependent :: real set ⇒ bool
```

that instantiate constants such as `vector_space.span` and `vector_space.dependent`, and inherit all associated lemmas from `vector_space`.

If we can show that $\{1, x, x^2, \ldots, x^n\}$ is linearly dependent, then (by the definition of linear dependence) it is not hard to see that there exists a non-zero polynomial with rational coefficients and degree at most n that vanishes at x:

Lemma 4.5 (`dependent_imp_poly`).
  fixes `x::real` **and** `n::nat`
  assumes `"rat.dependent {x ^ k | k. k ≤ n}"`
  shows `"∃p::rat poly. p ≠ 0 ∧ degree p ≤ n ∧ poly (of_rat_poly p) x = 0"`

where `of_rat_poly` converts coefficients of `p` from rational to real.

Now the problem becomes, how can we deduce the linear dependence of a set of vectors? The solution is based on a lemma: if m vectors live in the span of n vectors with $m > n$, then these m vectors are linearly dependent.

Lemma 4.6 (`(`**in** `vector_space) span_card_imp_dependent`).
  fixes `S B::"'b set"`
  assumes `"S ⊆ span B"` **and** `"finite B"` **and** `"card S > card B"`
  shows `"dependent S"`

Moreover, we show that for all $n \in \mathbb{N}$

$$(\alpha+\beta)^n \in \text{span}\{\alpha^i \beta^j \mid i\ j \in \mathbb{N}.i \leq \deg(p) \wedge j \leq \deg(q)\},$$

which is formally encoded by the following lemma in Isabelle:

Lemma 4.7 (*bpoly_in_rat_span*).

```
fixes p q::"rat poly" and x y::real
  and bp::"rat bpoly"
assumes "poly (of_rat_poly p) x = 0" and "p ≠ 0"
assumes "poly (of_rat_poly q) y = 0" and "q ≠ 0"
shows "bpoly (of_rat_bpoly bp) x y ∈ rat.span {x ^ k1 * y ^ k2
    |k1 k2. k1 ≤ degree p ∧ k2 ≤ degree q}"
```

Above, `bp::"rat bpoly"` means a bivariate polynomial with rational coefficients, and those coefficients are embedded into reals by the function `of_rat_bpoly`. In addition, `bpoly bp x y` evaluates `bp` at (x,y). It follows that

$$1, (\alpha+\beta), \ldots, (\alpha+\beta)^{(\deg(p)+1)(\deg(q)+1)}$$

are linearly dependent by applying Lemma 4.6,[2] since

$$\begin{aligned}(\deg(p)+1)(\deg(q)+1)+1 > \text{card}\{\alpha^i \beta^j \mid i\ j \in \mathbb{N}.\, i \leq \deg(p) \wedge j \leq \deg(q)\} \\ = (\deg(p)+1)(\deg(q)+1).\end{aligned}$$

Hence, there exists a non-zero polynomial with integer coefficients[3] vanishing at $\alpha+\beta$. Similarly, there exist such polynomials for the difference $\alpha-\beta$ and the product $\alpha\beta$:

Lemma 4.8 (*root_exist*).

```
fixes x y::real and p q::"rat poly"
assumes "poly (of_rat_poly p) x = 0" and "p≠0"
assumes "poly (of_rat_poly q) y = 0" and "q≠0"
defines "rt≡(λz::real. ∃r::int poly. r≠0 ∧ poly (of_int_poly r) z = 0)"
shows "rt (x+y)" and "rt (x-y)" and "rt (x*y)"
```

Every rational number r is real algebraic, given by the root of the first degree polynomial $x-r$. Therefore $0-\alpha$ is real algebraic, covering the case of $-\alpha$.

[2] In fact, there are corner cases when $\alpha+\beta = -1$, 0, 1, but all of them can be satisfied, so the conclusion holds.

[3] We have a lemma to convert a polynomial with rational coefficients into one with integer coefficients, multiplying out the denominators.

As for the multiplicative inverse, let

$$p(x) = a_n x^n + a_{n-1}x^{n-1} + \cdots + a_0.$$

Then clearly

$$p(\alpha) = 0 \wedge \alpha \neq 0 \Longrightarrow a_0 \left(\frac{1}{\alpha}\right)^n + a_{n-1}\left(\frac{1}{\alpha}\right)^{n-1} + \cdots + a_n = 0,$$

and hence we get vanishing polynomials for $1/\alpha$:

Lemma 4.9 (*inverse_root_exist*).

```
fixes x::real and p::"rat poly"
assumes "poly (of_rat_poly p) x = 0" and "p≠0"
shows "∃q::int poly. q≠0 ∧ poly q (inverse x) = 0"
```

as well as α/β (treated as $\alpha \times (1/\beta)$).

Finally, to define arithmetic operations on `alg`, we can lift the corresponding operations from its underlying type, `real`. For example, addition on `alg` is defined as

```
lift_definition plus_alg :: "alg ⇒ alg ⇒ alg"
  is "plus::real ⇒ real ⇒ real"
```

which leaves us a goal to show that the invariant condition on `alg` is maintained (that `alg` is closed under addition):

```
⋀r1 r2 ::real.
      ∃p. p ≠ 0 ∧ poly (of_int_poly p) r1 = 0 ⟹
      ∃p. p ≠ 0 ∧ poly (of_int_poly p) r2 = 0 ⟹
      ∃p. p ≠ 0 ∧ poly (of_int_poly p) (r1 + r2) = 0
```

and this goal can be discharged by our previous Lemma 4.8. Similarly, we obtain `0::alg` and `1::alg`, and the ordering operations are lifted from `real` as well:

```
lift_definition zero_alg::alg is "0::real"

lift_definition one_alg::alg is "1::real"

lift_definition less_alg::"alg ⇒ alg ⇒ bool"
  is "less::real ⇒ real ⇒ bool"
```

The command **lift_definition** is part of Isabelle's Lifting and Transfer package [51].

With zero, one, arithmetic and ordering operations defined, it follows that `alg` forms an ordered field:

instantiation *alg :: linordered_field*

Because *alg* is a subset of *real*, all the instance proofs of the **instantiation** above are one-liners, again thanks to the Lifting and Transfer package [51]. For example, the associativity of *alg* multiplication is proved by the following tactic:

show *"(a * b) * c = a * (b * c)"*
by *transfer auto*

And so, we have constructed the real algebraic numbers on an abstract level and proved that they form an ordered field. But now it is time to consider the question of executability.

4.2 Implementation

Executability is a key property of real algebraic numbers. They are a countable subset of the real numbers and can be represented exactly in computers. This section will demonstrate how I have implemented algebraic real numbers and achieved executability on their arithmetic operations through verified bivariate sign tests.

4.2.1 More pseudo-constructors on real numbers

Recall that my *alg* is actually a subset of *real*, hence executability on *real* operations can be reflected in *alg*. Therefore, my following focus is to extend executability on type *real*.

The set of real numbers, as we know, is uncountable, hence not every real number can be encoded finitely. That is, arithmetic operations can only be executable on a strict subset of the real numbers. Prior to our work, arithmetic operations on type *real* in Isabelle were only executable on rational numbers embedded into the reals (rational reals). For example, the following expression could be evaluated to be true:

value *"Ratreal (3/4) * Ratreal 2 > (0::real)"*

where *Ratreal* of type *rat* ⇒ *real* is a pseudo-constructor [37] that constructs a *real* from a *rat*. Executability of rational real numbers is established by *c*ode equations (i.e., equational theorems from the logic that can serve as a rewrite system) such as

Lemma 4.10 (*real_plus_code [code]*)**.** *"Ratreal x + Ratreal y = Ratreal (x + y)"*

so that sub-expressions match the left-hand side (of a code equation) will be replaced with the right-hand side when evaluating expressions. In the case of Lemma 4.10, the addition on the left is on real numbers, which is merely defined within the logic (as the addition of two

Cauchy sequences) and cannot be executed at all, while the one on the right is the executable rational addition. When both operands of the real addition happen to constructed by rational numbers (through `Ratreal`), the evaluator will rewrite with Lemma 4.10 and eliminate the (not-executable) real additions. More about the idea of code equations can be found in the tutorial [37].

Similar to the constructor of rational reals (`Ratreal`), we may first want to have a constructor `Alg` of type `_ ⇒ real` to construct algebraic reals from some encodings of real algebraic numbers.

An encoding (of a real algebraic number) is essentially a polynomial (with integer or rational coefficients) and a root selection strategy to distinguish a particular real root of the polynomial from any others. There are several such strategies, such as using a rational (or dyadic rational[4] for efficiency reasons) interval that only includes the target root, a natural number to indicate the index of the root and Thom encoding [8, Proposition 2.28]. I have decided to use the interval strategy, which is straightforward to implement. Therefore, `Alg` is of type `int poly ⇒ float ⇒ float ⇒ real`, where the two `float` arguments represent a dyadic rational interval.

As each `real` number in Isabelle is presented as a Cauchy sequence of type `nat ⇒ rat`, we explicitly construct such a sequence using a suitable encoding:

```
fun to_cauchy:: "rat poly × rat × rat ⇒ nat ⇒ rat"
where
  "to_cauchy (_, lb, ub) 0 = (lb+ub)/2"
| "to_cauchy (p, lb, ub) (Suc n) = (
    let c = (lb+ub)/2
    in if poly p lb * poly p c ≤ 0
       then to_cauchy (p, lb, c) n
       else to_cauchy (p, c, ub) n)"
```

where `poly p x` evaluates the polynomial `p` at the point `x`. Note, `rat poly × rat × rat` encodes a real algebraic number here (rather than `int poly × float × float`), as we can embed `int` and `float` into `rat`.

It can be then shown that the sequence constructed by `to_cauchy (p, lb, ub)` is indeed a Cauchy sequence and the real number represented by this sequence resides within the interval $[lb, ub]$, provided $lb < ub$:

Lemma 4.11 (`to_cauchy_cauchy`).
fixes `p::"rat poly"` **and** `lb ub ::rat`

[4] A dyadic rational number is a rational number of the form $a2^e$ for $a, e \in \mathbb{Z}$.

```
  assumes "lb < ub"
  defines "X ≡ to_cauchy (p,lb,ub)"
  shows "cauchy X"
```

Lemma 4.12 (*to_cauchy_bound*).

```
  fixes p::"rat poly" and lb ub ::rat
  defines "X ≡ to_cauchy (p,lb,ub)"
  assumes "lb < ub"
  shows "lb ≤ Real X" "Real X ≤ ub"
```

Note, the function `Real` of type `(nat ⇒ rat) ⇒ real` constructs a real number from its underlying representation (i.e., a Cauchy sequence).

Finally, we can finish the definition of `Alg`:

```
definition valid_alg::"int poly ⇒ float ⇒ float ⇒ bool" where
  "valid_alg p lb ub = (lb < ub ∧ poly p lb * poly p ub < 0
    ∧ card ({x::real. poly p x = 0 ∧ lb < x ∧ x < ub}) = 1)"
```

```
definition Alg:: "int poly ⇒ float ⇒ float ⇒ real" where
  "Alg p lb ub = (if valid_alg p lb ub
         then Real (to_cauchy (p, lb, ub))
         else undefined)"
```

where `valid_alg p lb ub` ensures

- `lb < ub`,
- the polynomial `p` is of different signs (and non-zero) at `lb` and `ub`,
- the polynomial `p` has exactly one real root within the interval `(lb,ub)`.

With the help of `Alg`, we can now encode the real algebraic number $\sqrt{2}$ as

```
Alg [:-2,0,1:] 1 2
```

where `[:-2,0,1:]` corresponds to the polynomial $-2x^0 + 0x^1 + 1x^2 = x^2 - 2$, and 1 and 2 are the lower bound and upper bound respectively, such that $\sqrt{2}$ is the only root of $x^2 - 2$ within the interval $(1,2)$.

Furthermore, we can formally derive that `Alg p lb ub` is indeed a root of `p` within the interval `(lb,ub)`:

Lemma 4.13 (`alg_bound_and_root`).
fixes `p::"int poly"` **and** `lb ub::float`
assumes `"valid_alg p lb ub"`
shows `"lb < Alg p lb ub"` **and** `"Alg p lb ub < ub"`
and `"poly (of_int_poly p) (Alg p lb ub) = 0"`

where `of_int_poly p` embeds the integer polynomial `p` into a real one.

4.2.2 Univariate sign determination through the Sturm-Tarski theorem

Given a polynomial q with rational coefficients and my encoding of a real algebraic number α

$$\alpha = (p, lb, ub)$$

where p is an integer polynomial, and lb and ub are dyadic rationals, we can effectively decide the sign of $q(\alpha)$ using the Sturm-Tarski theorem in Chapter 3, provided `valid_alg p lb ub` holds. The rationale behind is that `valid_alg p lb ub` ensures α is the only root of p within the interval (lb, ub), hence

$$\begin{aligned} \operatorname{sgn}(q(\alpha)) &= \sum_{x \in (lb,ub), p(x)=0} \operatorname{sgn}(q(x)) \\ &= \operatorname{TaQ}(q, p, lb, ub) \\ &= \operatorname{Var}(\operatorname{SRemS}(p, p'q); lb, ub). \end{aligned}$$

Importantly, it can be observed that evaluating $\operatorname{Var}(\operatorname{SRemS}(p, p'q); lb, ub)$ requires only rational arithmetic rather than costly algebraic arithmetic.

To be even more efficient, we can refine the procedure further to make use of dyadic rational arithmetic. The main advantage of dyadic rational arithmetic over rational arithmetic is reduced normalisation steps and possible bit-level operations. For example, consider two rational numbers $\frac{a_1}{b_1}$ and $\frac{a_1}{b_2}$ where $a_1, b_1, a_2, b_2 \in \mathbb{Z}$, their sum is

$$\frac{a_1}{b_1} + \frac{a_2}{b_2} = \frac{a_1 b_2 + a_2 b_1}{b_1 b_2} = \frac{(a_1 b_2 + a_2 b_1)/c}{(b_1 b_2)/c}$$
$$\text{where } c = \gcd(a_1 b_2 + a_2 b_1, b_1 b_2).$$

To counter the growth in the size of representations, we usually need to normalise the result by factoring out the gcd. Such gcd operations can be the source of major computational expense. Thankfully, they are unnecessary in the context of dyadic rationals. The sum of two

dyadic rationals (a_1, e_1) and (a_2, e_2) where $a_1, e_1, a_2, b_2 \in \mathbb{Z}$ is

$$a_1 2^{e_1} + a_2 2^{e_2} = \begin{cases} (a_1 2^{e_1 - e_2} + a_2) 2^{e_2} & \text{if } e_1 > e_2 \\ (a_1 + a_2 2^{e_2 - e_1}) 2^{e_1} & \text{otherwise.} \end{cases}$$

Moreover, multiplications by powers of two, such as $a_1 2^{e_1 - e_2}$, can be optimised by shift operations.

However, the problem with dyadic rational numbers is that they do not have the division operation (e.g. 1×2^0 divided by 3×2^0 is no longer a dyadic rational), hence they do not form a field, while Euclidean division only works for polynomials over a field. This problem can be solved if we switch from Euclidean division (`mod` and `div`):

$$p = (p \operatorname{div} q)\, q + (p \bmod q) \text{ and } (q = 0 \vee \deg(p \bmod q) < \deg(q))$$

to pseudo-division (`pmod` and `pdiv`) [29]:

$$\begin{aligned} \operatorname{lcoef}(q)^{1+\deg(p)-\deg(q)} p = (p \operatorname{pdiv} q)\, q + (p \operatorname{pmod} q) \\ \text{and } (q = 0 \vee \deg(p \operatorname{pmod} q) < \deg(q)) \\ \text{where } \operatorname{lcoef}(q) \text{ is the leading coefficient of } q, \end{aligned}$$

since pseudo-division can be carried out by polynomials over an integral domain (rather than a field).

Based on pseudo-division, the signed pseudo-remainder sequence (SPRemS) can be defined:

```
function spmods :: "'a::idom poly ⇒ 'a poly ⇒ ('a poly) list" where
  "spmods p q = (if p = 0 then [] else
      let
        m = (if even(degree p+1-degree q) then -1 else -lead_coeff q)
      in
        Cons p (spmods q (smult m (p pmod q))))"
```

where `smult` is the scalar product on polynomials and `lead_coeff q` is the leading coefficient of `q`. Accordingly, the function to count the difference in sign variations can be refined:

```
definition changes_itv_spmods::
  "'a ::linordered_idom ⇒ 'a ⇒ 'a poly ⇒ 'a poly ⇒ int" where
  "changes_itv_spmods a b p q = (let ps = spmods p q in
        changes_poly_at ps a - changes_poly_at ps b)"
```

and linked to the previous one based on signed remainder sequences (SRemS):

Lemma 4.14 (*changes_spmods_smods*).

```
fixes p q::"float poly" and a b::"float"
shows "changes_itv_spmods a b p q
        = changes_itv_smods (real_of_float a) (real_of_float b)
                            (of_float_poly p) (of_float_poly q)"
```

where `real_of_float` embeds a `float` into `real` and `of_float_poly` coverts a `float poly` (i.e., polynomial with dyadic rational coefficients) to a `real poly` by embedding each of the coefficients into `real`.

Finally, I define a function `sgn_at` that returns the sign of a univariate polynomial at some point:

```
definition "(sgn_at::real poly⇒real⇒real) = (λq x. sgn (poly q x))"
```

Note, for now, if either `x` or any coefficient of `q` is an irrational real number (e.g. an irrational real algebraic number), evaluating `sgn_at q x` will raise an exception, as Isabelle/HOL, by default, only supports rational arithmetic. By proving some code equations, we can establish the executability of `sgn_at q x` when `x` is constructed by `Alg p lb ub` and coefficients of `q` are rational reals:

Lemma 4.15 (*sgn_at_code_alg[code]*).

```
fixes q::"real poly" and p::"int poly" and lb ub::float
shows "sgn_at q (Alg p lb ub) = (
  if valid_alg p lb ub ∧ (∀x∈set (coeffs q). is_rat x) then
    (let
        p'::float poly = of_int_poly p;
        q'::float poly = of_int_poly (int_poly q)
     in
    of_int (changes_itv_spmods lb ub p' (pderiv p' * q')))
  else Code.abort (STR ''Invalid sgn_at'')
      (λ_. sgn_at q (Alg p lb ub)))"
```

where

- `∀x∈set (coeffs q). is_rat x` checks if each coefficient of `q` is rational,
- `of_int_poly` converts an integer polynomial into a dyadic rational one,
- `int_poly` clears denominators in the coefficients by multiplying each coefficient by the least common multiple (of the denominators),

- `Code.abort` throws an exception, if either (p, lb, ub) is an invalid representation of a real algebraic number or the polynomial `q` has any non-rational coefficient.

And note that evaluating `changes_itv_spmods lb ub p' (pderiv p' * q')` requires only dyadic arithmetic, which is much more efficient than exact algebraic arithmetic.

Moreover, the executability of `valid_alg` is established similarly as well:

Lemma 4.16 (`[code]`).

```
fixes p::"int poly" and lb ub::float
shows "valid_alg p lb ub = (lb < ub
  ∧ (sgn (poly (of_int_poly p) lb) * sgn (poly (of_int_poly p) ub) < 0)
  ∧ changes_itv_spmods lb ub (of_int_poly p) (pderiv (of_int_poly p)) = 1)"
```

where

```
changes_itv_spmods lb ub (of_int_poly p) (pderiv (of_int_poly p)) = 1
```

checks if the polynomial p has exactly one real root within the interval (lb, ub) by exploiting Sturm's theorem (a special case of the formalised Sturm-Tarski theorem).

After establishing the executability of `sgn_at` on real algebraic numbers, we can now check the sign of $p(x) = \frac{1}{2}x^2 - 1$ at $\sqrt{2}$ by typing the following command:

```
value "sgn_at [:-1,0,1/2:] (Alg [:-2,0,1:] 1 2)"
```

which returns 0 (i.e. $p(\sqrt{2}) = 0$).

4.2.3 Deciding the sign of a bivariate polynomial at a real algebraic point

In the previous section, I have demonstrated how to algorithmically decide the sign of a *univariate* polynomial (with rational coefficients) at real algebraic points with only (dyadic) rational arithmetic. In this section, I will generalise this idea to *bivariate* cases.

To illustrate the idea for a bivariate sign determination procedure, suppose we want to decide the sign of $q(y,x) \in \mathbb{Q}[y,x]$ at (α, β) with $\alpha = (p_1, a_1, b_1)$ and $\beta = (p_2, a_2, b_2)$. By substituting y by β, we have $q(\beta, x)$ as a univariate polynomial in $\mathbb{Q}(\beta)[x]$, where $\mathbb{Q}(\beta)$ is the field $\mathbb{Q}$ extended by β. Pretending to have arithmetic of real algebraic numbers, we can still use the univariate sign determination procedure:

$$\begin{aligned} &\mathrm{TaQ}(q(\beta,x), p_1(x), a_1, b_1) \\ =&\mathrm{Var}(\mathrm{SPRemS}(p_1(x), p_1(x)' q(\beta,x)); a_1, b_1) \end{aligned} \tag{4.2}$$

To proceed from (4.2), we need to somehow eliminate algebraic arithmetic in the operation pmod inside SPRemS. A key lemma is

Lemma 4.17 (*poly_y_dist_pmod*).

```
fixes p::"'a::idom bpoly" and y::'a
assumes "poly (lead_coeff p) y ≠ 0" and "poly (lead_coeff q) y ≠ 0"
shows "(poly_y p y) pmod (poly_y q y) = poly_y (p pmod q) y"
```

where *'a bpoly* is, in fact, a synonym of *'a poly poly*, which is the type we use to represent bivariate polynomials in Isabelle/HOL. This is the so-called *recursive representation*, where for example, the bivariate polynomial

$$4xy+3x+2y+1 = 1+2y+(3+4y)x \in (\mathbb{Z}[y])[x]$$

is encoded as *[:[:1,2:],[:3,4:]:]*. Moreover, the function *poly_y p a* substitutes the value *a* for variable y in *p*. For example,

```
value "poly_y [:[:1,2:],[:3,4:]:] (2::int)"
```

evaluates to *[:5, 11:]*, which can be mathematically interpreted as $(4xy+3x+2y+1)[y \to 2] = 5+11x$.

An important property about Lemma 4.17 is that the left-hand occurrence *pmod* operates over $\mathbb{Q}(\beta)[x]$ (as *poly_y p y* can be considered to be of type $\mathbb{Q}(\beta)[x]$, provided $p \in \mathbb{Q}[y,x]$ and *y* is instantiated to β), which demands algebraic arithmetic, while the right-hand occurrence of *pmod* operates over $\mathbb{Q}[y,x]$, which only requires arithmetic over rational numbers. Therefore, provided the leading coefficients of *p* and *q* do not vanish when evaluating at *y* (i.e., *poly (lead_coeff p) y* $\neq$ *0* and *poly (lead_coeff q) y* $\neq$ *0*), we can eliminate algebraic arithmetic in *pmod*.

In order to rewrite with Lemma 4.17 inside a remainder sequence, we need to satisfy its assumptions. Therefore, I have defined a function *degen* (for 'degenerates') of type *'a bpoly* $\Rightarrow$ *'a* $\Rightarrow$ *'a bpoly*, such that *degen p y* iteratively removes the leading coefficient of *p* until it does not vanish at *y* or *p* becomes *0*:

```
lift_definition degen::"'a bpoly ⇒ 'a ⇒'a bpoly" is
  "λp y n. (if poly_y p y ≠ 0 ∧ n ≤degree (poly_y p y)
           then coeff p n else 0)"
```

Note that the term *(λp y n. ...)* above is of type *'a bpoly* $\Rightarrow$ *'a* $\Rightarrow$ *nat* $\Rightarrow$ *'a poly*, so *degen* is defined in a way where *degen p y* (of type *'a bpoly*) is lifted from its underlying representation,[5] which is of type *nat* $\Rightarrow$ *'a poly*.

[5] *'a poly* is constructed as a subset of *nat* $\Rightarrow$ *'a* (i.e., a mapping from exponents to coefficients). Haftmann et al. [38] discuss how polynomials are formalised in Isabelle/HOL.

For example, a bivariate polynomial $1+y+(y^2-2)x^2$ degenerates to $1+y$ when $y=\sqrt{2}$, hence the command

```
value "degen[:[:1,1:],0,[:-2,0,1:]:] (Alg [:-2,0,1:] 1 2)"
```

evaluates to `[:[:1,1:]:]`.

Properties of *degen* include that degenerating the bivariate polynomial *p* with respect to *y* does not affect the result of evaluating it at *y*:

Lemma 4.18 (*poly_y_degen*). *"poly_y (degen p y) y = poly_y p y"*

This holds because only leading coefficients that vanish at *y* are removed. Moreover, the leading coefficient of *degen p y* will not vanish at *y* unless *p* vanishes at *y*:

Lemma 4.19 (*degen_lc_not_vanish*).

```
assumes "degen p y ≠ 0"
shows "poly (lead_coeff (degen p y)) y ≠0"
```

With the help of *degen*, we can define another remainder sequence *spmods_y* that is similar to the previous signed pseudo remainder sequence *spmods* except for that *spmods_y p q y* keeps degenerating each remainder with respect to *y*:

```
function spmods_y :: "'a::idom bpoly ⇒ 'a poly poly
   ⇒'a  ⇒ ('a poly poly) list" where
  "spmods_y p q y = (if p = 0 then [] else
     let
       mul = (if even(degree p+1-degree q)
                then -1
                else -lead_coeff q);
       r = degen (smult mul (p pmod q)) y
     in
       Cons p (spmods_y q r y))"
```

By exploiting Lemma 4.17, we have established the relationship between *spmods* and *spmods_y*:

Lemma 4.20 (*spmods_poly_y_dist*).

```
fixes p q :: "'a::idom bpoly" and y::"'a::idom"
assumes "poly (lead_coeff p) y ≠ 0" and "poly (lead_coeff q) y ≠ 0"
shows "spmods (poly_y p y) (poly_y q y)
        = map (λp. poly_y p y) (spmods_y p q y)"
```

Note, similar to what I have stated for Lemma 4.17, the importance of Lemma 4.20 is that the left-hand remainder sequence (`spmods`) requires arithmetic over $\mathbb{Q}(\beta)[x]$ (provided `p q` $\in \mathbb{Q}[y,x]$ and $y = \beta$) while the right-hand sequence (`spmods_y`) only requires arithmetic over $\mathbb{Q}[y,x]$.

Let `spmods_y p q y` be represented as $\mathrm{SPRemS}'(p,q,y)$, we can rewrite SPRemS with Lemma 4.20:

$$\begin{aligned}&\mathrm{lcoef}_x(q)(\beta) \neq 0 \Longrightarrow\\ &\qquad \mathrm{SPRemS}(p_1(x), p_1(x)'q(\beta,x)) = \mathrm{SPRemS}'(p_1(x), p_1(x)'q(y,x), \beta)[y \to \beta] \quad (4.3)\end{aligned}$$

where $\mathrm{lcoef}_x(q) \in \mathbb{Q}[y]$ is the leading coefficient of the bivariate polynomial $q \in \mathbb{Q}[y,x]$ with respect to x. $[y \to \beta]$ performs substitution on a list of polynomials. For example, let $[x, x+y]$ be a list of polynomials, then $[x,x+y][y \to 3] = [x,x+3]$.

By Equations (4.2) and (4.3), we have

$$\begin{aligned}&\mathrm{lcoef}_x(q)(\beta) \neq 0 \Longrightarrow\\ &\mathrm{TaQ}(q(\beta,x), p_1(x), a_1, b_1) = \mathrm{Var}(\mathrm{SPRemS}'(p_1(x), p_1(x)'q(y,x), \beta)[y \to \beta]; a_1, b_1) \quad (4.4)\end{aligned}$$

Note, SPRemS' operates over $\mathbb{Q}[y,x]$ and Var requires deciding the sign of some univariate polynomial $r \in \mathbb{Q}(\beta)[x]$ when $x = a_1 \vee x = b_1$. Fortunately, as both a_1 and b_1 are rational numbers, the sign of $r(a_1)$ and $r(b_1)$ can be decided again using our univariate sign determination procedure. Hence, evaluating the right-hand side of Equation (4.4) requires only arithmetic on rational numbers, and we can now decide the sign of $q(\beta,\alpha)$ with only rational arithmetic (provided $\mathrm{lcoef}_x(q)(\beta) \neq 0$).

To give an example, suppose we want to decide the sign of $\alpha - \beta$ when $\alpha = \sqrt{2} = (x^2-2,1,2)$ and $\beta = \sqrt{3} = (x^2-3,1,2)$, the calculation is as follows:

$$\begin{aligned}\mathrm{TaQ}(x-\beta, x^2-2,1,2) &= \mathrm{Var}(\mathrm{SPRemS}'(x^2-2,(2x)(x-y),\beta)[y \to \beta];1,2)\\ &= \mathrm{Var}([x^2-2, 2x^2-2xy, -4xy+8, 64y^2-128][y \to \beta];1,2)\\ &= \mathrm{Var}([x^2-2, 2x^2-2xy, -4xy+8, 64y^2-128][x \to 1, y \to \beta])\\ &\quad - \mathrm{Var}([x^2-2, 2x^2-2xy, -4xy+8, 64y^2-128][x \to 2, y \to \beta])\\ &= \mathrm{Var}([-1, -2y+2, -4y+8, 64y^2-128][y \to \sqrt{3}])\\ &\quad - \mathrm{Var}([2, -4y+8, -8y+8, 64y^2-128][y \to \sqrt{3}])\\ &= 1-2 = -1,\end{aligned}$$

provided $\text{lcoef}_x(x-y)(\beta) = 1 \neq 0$. Therefore, we know that $(x-y)[x \to \sqrt{2}, y \to \sqrt{3}]$ is negative.

In Isabelle/HOL, I have defined the bivariate sign determination procedure as `bsgn`:

```
definition bsgn_at::"real bpoly ⇒ real ⇒ real ⇒ real" where
  "bsgn_at q x y=sgn (bpoly q x y)"
```

and the executability of `bsgn_at` on the algebraic reals is established by the following code equation:

Lemma 4.21 (`bsgn_at_code2[code]`)**.**

```
"bsgn_at q (Alg p1 lb1 ub1) y =
    (if valid_alg p1 lb1 ub1
     then
       (let
          q' = degen q y
        in (if q' = 0 then 0 else
              let ps = spmods_y (lift_x p1) (lift_x (pderiv p1) * q') y
              in changes_bpoly_at ps lb1 y - changes_bpoly_at ps ub1 y))
     else
       Code.abort (STR ''invalid Alg'') (λ_. bsgn_at q (Alg p1 lb1 ub1) y))"
```

where letting `q' = degen q y` enables `q'` to satisfy the assumption of Equation (4.4), `pderiv` means derivation and `lift_x :: 'a::zero ⇒ 'a poly poly` lifts a univariate polynomial to bivariate. Moreover,

```
changes_bpoly_at ps lb1 y - changes_bpoly_at ps ub1 y
```

implements the Var operation. And also, `Code.abort` throws an exception when `Alg p1 lb1 ub1` fails to be a valid real algebraic number. Essentially, Lemma 4.21 implements Equation (4.4).

Thanks to `bsgn_at`, we can now formally evaluate the sign of $(x-y)[x \to \sqrt{2}, y \to \sqrt{3}]$ by the following command:

```
value "bsgn_at [:[:0,-1:],[:1:]:] (Alg [:-2,0,1:] 1 2) (Alg [:-3,0,1:] 1 2)"
```

which returns `-1`.

To restate: I have implemented a decision procedure (called `bsgn_at`) to decide the sign of a *bivariate* polynomial with rational coefficients at real algebraic points. This procedure uses no real algebraic arithmetic, just arithmetic on (dyadic) rational numbers.

4.2.4 Enable executability on algebraic reals

Although it is possible to do verified algebraic arithmetic as in Coq [20], with the help of `bsgn_at`, we can do better. We can actually use untrusted external code to do such arithmetic, validate the result and bring it back the framework of higher order logic. The rationale behind this methodology is that untrusted but sophisticated code usually offers by far the best performance. Using untrusted code when building decision procedures improves performance in most cases; on the other hand, to provide our own trustworthy code would require costly formal verification. Another benefit of using external untrusted code is modularity: we can easily substitute one piece of code by another without modifying any proofs.

The following lemma illustrates the idea of using untrusted code in algebraic arithmetic:

Lemma 4.22 (`alg_add_bsgn`).

```
fixes p1 p2 p3::"int poly" and lb1 lb2 lb3 ub1 ub2 ub3::"float"
defines "x ≡ Alg p1 lb1 ub1" and "y ≡ Alg p2 lb2 ub2"
    and "pxy ≡ [:[:0::real,1:],[:1:]:]"
assumes valid:"valid_alg p3 lb3 ub3"
  and bsgn1:"bsgn_at ((lift_x (of_int_poly p3)) ∘p pxy) x y = 0"
  and bsgn2:"bsgn_at ([:[:- real_of_float lb3,1:],[:1:]:]) x y > 0"
  and bsgn3:"bsgn_at ([:[:- real_of_float ub3,1:],[:1:]:]) x y < 0"
shows "Alg p3 lb3 ub3 = x + y"
```

Here, let `x = Alg p1 lb1 ub1` and `y = Alg p2 lb2 ub2`,

- the assumption `valid` checks if `Alg p3 lb3 ub3` is a valid real algebraic number, which guarantees that `p3` has exactly one real root within interval $(lb1, ub1)$,
- `bsgn1` checks if `p3` vanishes at `x+y`, within which $\circ_p$ is polynomial composition and `pxy` stands for the bivariate polynomial $x+y$,
- `bsgn2` and `bsgn3` checks if `lb3 < x+y` and `x+y < ub3` respectively.

With these three assumptions, all of which can be computationally checked, we can show `Alg p3 lb3 ub3 = x + y`. Therefore, to calculate real algebraic addition, we can use untrusted code to compute `p3`, `lb3` and `ub3`, and obtain the result as a sound Isabelle theorem with the help of Lemma 4.22.

In order to interact with untrusted code, I have followed the idea of foreign function interface [63]. First, I declare a constant `alg_add` without attaching any definitions:

```
consts alg_add::"integer list × (integer × integer) × (integer × integer)
              ⇒ integer list × (integer × integer) × (integer × integer)
```

```
⇒ integer list × (integer× integer) × (integer × integer)
    × ((integer × integer) option)"
```

where *integer × integer* encodes *float* and *integer list* encodes *int poly*. As Isabelle does not directly link *float* to the target language, I decide to use the quotient of two integers (which appears more primitive and closer to the native level) to represent *float*, and similar reasons apply to *int poly*. Also note, in Isabelle, *integer* is an equivalent type to *int* but directly maps to arbitrary precision integers (e.g. IntInf.int in SML) in the target language when doing evaluations. Essentially, I let *alg_add* be an unspecified constant that takes representations of two algebraic numbers and returns the representation of their addition and *(integer × integer) option*, where *(integer × integer) option* is a possible optimisation in case the result is a rational number.

To enable *alg_add* to do calculations, I use the adaptation technique to link a constant in Isabelle/HOL to a target language constant, so that when the logical constant gets called in evaluation, the target language constant gets invoked instead:

```
code_printing constant alg_add ⇀ (SML) "untrustedAdd"
```

where *untrustedAdd* is currently backed up by Grant Passmore's code for algebraic operations in MetiTarski [74]. After such linking, *alg_add* becomes executable:

```
value "alg_add([-2,0,1],(1,1),(2,1))([-3,0,1],(1,1),(2,1))"
```

evaluates the sum of $\sqrt{2} = (x^2 - 2, 1, 2)$ and $\sqrt{3} = (x^3 - 3, 1, 2)$, and returns the result *([1, 0, - 10, 0, 1], (2, 1), (4, 1), None)*, which encodes $\sqrt{2} + \sqrt{3}$ as $(x^4 - 10x^2 + 1, 2, 4)$.

The code equation for real algebraic addition is the following:

Lemma 4.23 (*[code]*).

```
"Alg p1 lb1 ub1 + Alg p2 lb2 ub2 =
   (let
     (ns,(lb3_1,lb3_2),(ub3_1,ub3_2),_) = alg_add (to_alg_code p1 lb1 ub1)
                                                   (to_alg_code p2 lb2 ub2);
     (p3,lb3,ub3) = of_alg_code ns lb3_1 lb3_2 ub3_1 ub3_2
   in
     (if (*assumptions in Lemma 4.22*) then
       Alg p3 lb3 ub3
     else
       Code.abort (STR ''alg_add fails to compute a valid answer'')
                   (λ_. Alg p1 lb1 ub1 + Alg p2 lb2 ub2)))"
```

where *to_alg_code* encodes *int poly* and *float* to *integer list* and *integer* $\times$*integer* respectively, while *of_alg_code* does the reverse. The command *Code.abort* inserts an exception with an error message, that is, when the untrusted computation *alg_add* fails to give a correct result, an exception will be thrown. This code equation can be shown to be correct using Lemma 4.22.

In a very similar way of exploiting untrusted code, I have defined subtraction, multiplication and inversion. As for negation, the code equation does not require untrusted code:

Lemma 4.24 (*[code]*).

```
"- Alg p lb ub =
          (if valid_alg p lb ub then
                Alg (p ∘p [:0,-1:]) (-ub) (-lb)
            else
                Code.abort (STR ''invalid Alg'') (%_. - Alg p lb ub))"
```

where *p* $\circ_p$ *[:0,-1:]* substitutes variable x in a univariate polynomial *p* by $-x$. The rationale behind this code equation is

$$p(\alpha) = 0 \wedge q(x) = p(-x) \Longrightarrow q(-\alpha) = 0.$$

Also, *p(-x)* can be shown to have exactly one real root within the interval $(-lb, -ub)$, provided that *p(x)* has exactly one within the interval (lb, ub).

By composing multiplicative inverse and multiplication, we obtain division:

Lemma 4.25 (*[code]*). *"Alg p1 lb1 ub1 / Alg p2 lb2 ub2*
*= Alg p1 lb1 ub1 * (inverse (Alg p2 lb2 ub2))"*

Finally, the executability of the arithmetic of our algebraic reals can be illustrated by the following example:

value *"Alg [:-2,0,1:] 1 2 / Alg [:-3,0,1:] 1 2*
+ Alg [:-5,0,1:] 2 3 > Alg [:-7,0,2:] 1 2"

which stands for $\sqrt{2}/\sqrt{3}+\sqrt{5} > \sqrt{7/2}$ and returns *true*.

To repeat, I have enabled executable arithmetic and comparison operations on algebraic reals by deriving new code equations for the pseudo constructor *Alg*. Some of these code equations, such as the one for algebraic addition, depend on untrusted code, whose results are verified using the bivariate sign determination algorithm *bsgn_at*, and thus brought back into higher-order logic.

Expression	Verified evaluation	Unverified evaluation (MetiTarki)
$(-\sqrt{2})+(-\sqrt{3})-(-\sqrt{5})$	0.24s	0.02s
$(\prod_{n=2}^{10}\sqrt{n})(\sqrt{17}-\sqrt{19})$	0.84s	0.15s
$\sum_{n=2}^{5}\sqrt{n}$	1.9s	1.4s
$(\sqrt{2}+\sqrt{6})^3$	1.18s	0.26s

Fig. 4.1 Comparison between verified evaluation and unverified evaluation

4.2.5 Linking the algebraic reals to the real algebraic numbers

We have just seen executable arithmetic and ordering operations on algebraic reals constructed by the constructor *Alg*, of type `int poly ⇒ float ⇒ float ⇒ real`. To enable the same executability on type `alg`, we only need to build a constructor for `alg` lifted from *Alg*:

```
lift_definition RAlg:: "int poly ⇒ float ⇒ float ⇒ alg" is
  "λp lb ub. if valid_alg p lb ub then Alg p lb ub else 0"
```

and we can then have executable arithmetic and ordering operations on `alg` as well:

```
value "RAlg [:-2,0,1:] 1 2 * RAlg [:-3,0,1:] 1 2
        > RAlg [:-5,0,1:] 2 3"
```

where `op *` and `op >` in the command above operate over `alg` instead of `real`.

4.3 Experiments

This section presents a few examples to demonstrate the efficiency of my implementation. All the experiments are run on an Intel Core 2 Quad Q9400 (quad core, 2.66 GHz) and 8 gigabytes RAM. When benchmarking verified operations, the expression to evaluate is first defined in Isabelle/HOL, and then extracted and evaluated in Poly/ML. The reason for this is that when invoking **value** in Isabelle/HOL to evaluate an expression, a significant and unpredictable amount of time is spent generating code, so I evaluate an extracted expression to obtain more precise results.

Firstly, I compare evaluations of the same expression using verified arithmetic from my implementation and unverified ones from MetiTarski (see Fig. 4.1). The data in Fig. 4.1 indicate that my verified arithmetic is 2 to 15 times slower than unverified ones due to overhead in various validity checks and inefficient data structures. I expect to narrow this gap by further refining code equations in the implementation. The experiments have

also demonstrated inefficiencies in algebraic arithmetic in the current version of MetiTarski, which evaluates $(\sqrt{2}+\sqrt{6})^3$ to

$$(x^8 - 3584x^6 + 860160x^4 - 14680064x^2 + 16777216, \frac{2601}{128}, \frac{6125}{8})$$

while Mathematica[6] can evaluate the same expression to

$$(x^4 - 3328x^2 + 4096, 2, 59)$$

instantly. By basing our untrusted code on more sophisticated algebraic arithmetic implementations such as Z3 and Mathematica, which effectively control coefficient and degree growth, we should obtain further improvements in my algebraic arithmetic.

I have also experimented with the bivariate sign determination procedure alone, which appears to be quite efficient. For example, given the large bivariate polynomial $p(x,y)$ shown in Fig. 4.2, `bsgn_at` can decide $p(\sqrt{6},\sqrt{7}) = 0$ or $p(\sqrt{13},\sqrt{29}) > 0$ in less than $0.05s$. Note, the current `bsgn_at` always calculates a remainder sequence no matter whether the result is -1, 0 or 1, so `bsgn_at` should take similar amount of time if the input argument is of similar complexity. In the future, I may optimise `bsgn_at` by letting it attempt to decide the sign using interval arithmetic before calculating a remainder sequence; in this case `bsgn_at` may run much faster if the polynomial does not vanish at the algebraic point.

4.4 Related work

Prior to this work, Cyril Cohen constructed real algebraic numbers in Coq [20], from which I have gained much inspiration. There are some major differences between my work and his:

- Cohen's work is part of the gigantic formalisation of the odd order theorem [36] and is mainly of theoretical interest. My work, on the contrary, is for practical purposes, as I intend to build effective decision procedures on the top of the current formalisation. This difference in intent is fundamental and leads to different design choices, such as whether to use efficient untrusted code.

- My formalisation follows Isabelle's tradition of separating abstraction and implementation, that is, formalising theories first and restoring executability afterwards. It is possible to switch to another encoding of real algebraic numbers (such as Thom

[6] We use the RootReduce and IsolatingInterval command in Mathematica 9 to find the defining polynomial and root isolation interval.

$$P(x,y) = y^{14}x^{24} - 49y^{12}x^{24} + 1029y^{10}x^{24} - 12005y^{8}x^{24} + 84035y^{6}x^{24} - 352947y^{4}x^{24} + 823543y^{2}x^{24} -
823543x^{24} + 4y^{15}x^{23} - 196y^{13}x^{23} + 4116y^{11}x^{23} - 48020y^{9}x^{23} + 336140y^{7}x^{23} - 1411788y^{5}x^{23} +
3294172y^{3}x^{23} - 3294172yx^{23} + 6y^{16}x^{22} - 380y^{14}x^{22} + 10388y^{12}x^{22} - 160524y^{10}x^{22} + 1536640y^{8}x^{22} -
9344692y^{6}x^{22} + 35294700y^{4}x^{22} - 75765956y^{2}x^{22} + 70824698x^{22} + 4y^{17}x^{21} - 488y^{15}x^{21} +
18424y^{13}x^{21} - 348488y^{11}x^{21} + 3841600y^{9}x^{21} - 25950008y^{7}x^{21} + 106354696y^{5}x^{21} - 243768728y^{3}x^{21} +
240474556yx^{21} + y^{18}x^{20} - 435y^{16}x^{20} + 23124y^{14}x^{20} - 565068y^{12}x^{20} + 7991214y^{10}x^{20} - 70978362y^{8}x^{20} +
404376420y^{6}x^{20} - 1441435548y^{4}x^{20} + 2937577881y^{2}x^{20} - 2619690283x^{20} - 240y^{17}x^{19} + 21360y^{15}x^{19} -
717360y^{13}x^{19} + 12759600y^{11}x^{19} - 135416400y^{9}x^{19} + 891443280y^{7}x^{19} - 3585941520y^{5}x^{19} +
8103663120y^{3}x^{19} - 7906012800yx^{19} - 60y^{18}x^{18} + 14220y^{16}x^{18} - 682560y^{14}x^{18} + 15664320y^{12}x^{18} -
210533400y^{10}x^{18} + 1786632120y^{8}x^{18} - 9753438240y^{6}x^{18} + 33374668320y^{4}x^{18} - 65372843340y^{2}x^{18} +
56083278300x^{18} + 6480y^{17}x^{17} - 505440y^{15}x^{17} + 15876000y^{13}x^{17} - 271162080y^{11}x^{17} + 2800526400y^{9}x^{17} -
18078953760y^{7}x^{17} + 71662358880y^{5}x^{17} - 160096759200y^{3}x^{17} + 154760200560yx^{17} + 1620y^{18}x^{16} -
277020y^{16}x^{16} + 12299040y^{14}x^{16} - 269256960y^{12}x^{16} + 3483988200y^{10}x^{16} - 28557590040y^{8}x^{16} +
150730554240y^{6}x^{16} - 498587050080y^{4}x^{16} + 943236739620y^{2}x^{16} - 780471701100x^{16} -
103680y^{17}x^{15} + 7516800y^{15}x^{15} - 226074240y^{13}x^{15} + 3751816320y^{11}x^{15} - 37962691200y^{9}x^{15} +
241343141760y^{7}x^{15} - 945333244800y^{5}x^{15} + 2091930986880y^{3}x^{15} - 2006546048640yx^{15} -
25920y^{18}x^{14} + 3576960y^{16}x^{14} - 148770432y^{14}x^{14} + 3128969088y^{12}x^{14} - 39196700928y^{10}x^{14} +
311791939200y^{8}x^{14} - 1597046855040y^{6}x^{14} + 5119436939904y^{4}x^{14} - 9362458478016y^{2}x^{14} +
7462643602176x^{14} + 1088640y^{17}x^{13} - 75333888y^{15}x^{13} + 2197746432y^{13}x^{13} - 35697376512y^{11}x^{13} +
355480151040y^{9}x^{13} - 2232206242560y^{7}x^{13} + 8658032737536y^{5}x^{13} - 19006687252224y^{3}x^{13} +
18110145400704yx^{13} + 272160y^{18}x^{12} - 32169312y^{16}x^{12} + 1263911040y^{14}x^{12} - 25636818816y^{12}x^{12} +
311754598848y^{10}x^{12} - 2411253230400y^{8}x^{12} + 11999023392384y^{6}x^{12} - 37270525541760y^{4}x^{12} +
65761344808992y^{2}x^{12} - 50251170777696x^{12} - 7838208y^{17}x^{11} + 525159936y^{15}x^{11} - 14978815488y^{13}x^{11} +
239276975616y^{11}x^{11} - 2352442176000y^{9}x^{11} + 14622780566016y^{7}x^{11} - 56251597312512y^{5}x^{11} +
122646925287936y^{3}x^{11} - 116191823956992yx^{11} - 1959552y^{18}x^{10} + 205752960y^{16}x^{10} -
7683683328y^{14}x^{10} + 150666034176y^{12}x^{10} - 1780750718208y^{10}x^{10} + 13398166381824y^{8}x^{10} -
64739208683520y^{6}x^{10} + 194443460499456y^{4}x^{10} - 329440707211392y^{2}x^{10} + 239069288578176x^{10} +
39191040y^{17}x^{9} - 2564213760y^{15}x^{9} + 71876367360y^{13}x^{9} - 1133012966400y^{11}x^{9} + 11022871910400y^{9}x^{9} -
67938530042880y^{7}x^{9} + 259521420856320y^{5}x^{9} - 562515973125120y^{3}x^{9} + 530240466470400yx^{9} +
9797760y^{18}x^{8} - 936385920y^{16}x^{8} + 33399164160y^{14}x^{8} - 634130622720y^{12}x^{8} + 7287769843200y^{10}x^{8} -
53313060971520y^{8}x^{8} + 249688212560640y^{6}x^{8} - 722246796875520y^{4}x^{8} + 1165376329568640y^{2}x^{8} -
789597216374400x^{8} - 134369280y^{17}x^{7} + 8633226240y^{15}x^{7} - 238673433600y^{13}x^{7} + 3721659540480y^{11}x^{7} -
35891546342400y^{9}x^{7} + 219624001551360y^{7}x^{7} - 833893702548480y^{5}x^{7} + 1798206799334400y^{3}x^{7} -
1687547919375360yx^{7} - 33592320y^{18}x^{6} + 2972920320y^{16}x^{6} - 101683952640y^{14}x^{6} +
1871528924160y^{12}x^{6} - 20912730854400y^{10}x^{6} + 148566805309440y^{8}x^{6} - 672422071587840y^{6}x^{6} +
1861440445025280y^{4}x^{6} - 2821801438955520y^{2}x^{6} + 1729044999360000x^{6} + 302330880y^{17}x^{5} -
19147622400y^{15}x^{5} + 523435530240y^{13}x^{5} - 8088560363520y^{11}x^{5} + 77428953907200y^{9}x^{5} -
470864825948160y^{7}x^{5} + 1778446285056000y^{5}x^{5} - 3817731358586880y^{3}x^{5} + 3568748878679040yx^{5} +
75582720y^{18}x^{4} - 6273365760y^{16}x^{4} + 206451680256y^{14}x^{4} - 3686763838464y^{12}x^{4} +
40038373799424y^{10}x^{4} - 275719665553920y^{8}x^{4} + 1200874682004480y^{6}x^{4} - 3151406817119232y^{4}x^{4} +
4386241354376448y^{2}x^{4} - 2269890275159808x^{4} - 403107840y^{17}x^{3} + 25234550784y^{15}x^{3} -
683429031936y^{13}x^{3} + 10480561975296y^{11}x^{3} - 99689778155520y^{9}x^{3} + 602977978552320y^{7}x^{3} -
2266926198018048y^{5}x^{3} + 4846858942205952y^{3}x^{3} - 4514882302328832yx^{3} - 100776960y^{18}x^{2} +
7921069056y^{16}x^{2} - 251539292160y^{14}x^{2} + 4361304342528y^{12}x^{2} - 46001094580224y^{10}x^{2} +
306328298895360y^{8}x^{2} - 1276416305160192y^{6}x^{2} + 3130065461698560y^{4}x^{2} - 3834330190580736y^{2}x^{2} +
1377703055490048x^{2} + 241864704y^{17}x - 14995611648y^{15}x + 402946596864y^{13}x - 6139009916928y^{11}x +
58071715430400y^{9}x - 349591726891008y^{7}x + 1308936465801216y^{5}x - 2788603774967808y^{3}x +
2589417791041536yx + 60466176y^{18} - 4534963200y^{16} + 139314069504y^{14} - 2346571358208y^{12} +
24016802310144y^{10} - 154180404467712y^{8} + 609753012019200y^{6} - 1365846746923008y^{4} +
1344505391502336y^{2} - 49796495981568$$

Fig. 4.2 A large bivariate polynomial

encoding) without modifying any definition or lemma on the abstract level. It is also possible to have multiple implementations of one abstraction [38], so that when doing proof by reflection the code generator can choose the most efficient one depending on the situation. On the other hand, Cohen's formalisation is constructive and therefore should be executable, though it may not be very efficient.

- In Cohen's formalisation, arithmetic on real algebraic numbers is defined via verified bivariate resultants, while mine is mainly based on a bivariate sign determination procedure and some untrusted code.

After the publication of this chapter in 2016 [62], Thiemann and Yamada gave an independent formalisation of algebraic numbers in Isabelle/HOL [83]. They have formalised a theory of resultants and carried out algebraic arithmetic in a verified way (while I was using untrusted external programs to compute resultants). Their formalisation is extensively optimised and heavyweight: the formalisation contains more than 80000 LOC including prerequisite entries in the Archive of Formal Proofs[7], while my formalisation is merely about 6000 LOC. As a result, they do provide a faster exact algebraic arithmetic than my current set-up, due to the inefficiency of MetiTarski as discussed in §4.3. On the other hand, my formalisation still has the advantages of being modular and providing univariate/bivariate sign determination procedures that require only (dyadic) rational arithmetic.

4.5 Remarks

4.5.1 Modularity

A distinguishing feature of this work is modularity, where the dependencies between different parts are shown in Fig. 4.3. In particular, the modularity is reflected in two ways:

- Separation between the abstract type, `alg`, and the finite encoding, *`Alg`*. Switching to another encoding does not affect anything on the abstract level or further theories based on the abstraction.

- Use of untrusted code. Untrusted code is outside the logic of Isabelle/HOL (which is why I have used a dashed arrow in Fig. 4.3 to indicate the detached relation), hence we do not need to modify our formalisation as we revise the untrusted code, or substitute new code.

This modularity should make the formalisation easy to maintain.

[7] www.isa-afp.org

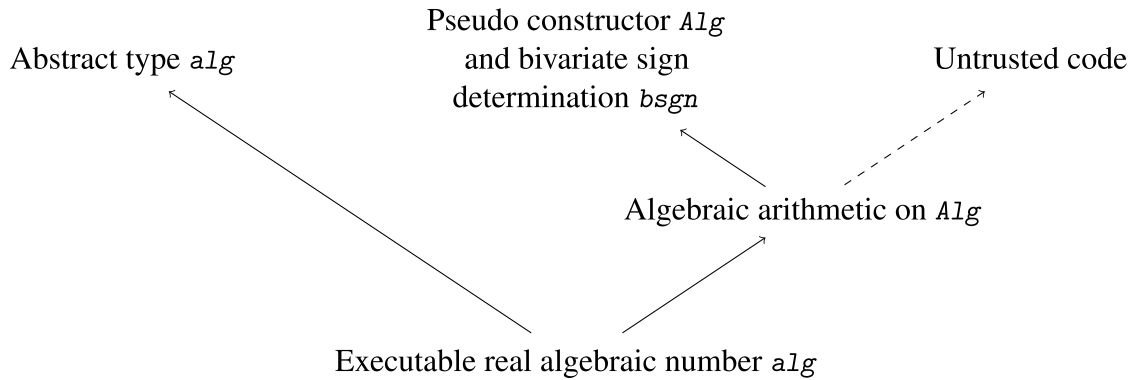

Fig. 4.3 Dependence tree of my formalisation of real algebraic numbers

4.5.2 A potential problem

There is one potential drawback with my formalisation, and it is related to the use of untrusted code. Recall that when interfacing with untrusted code, I declared a constant in the logic without specifying it and linked it to a constant in the target language. In this case the logic constant can be executed but no lemmas are associated with it. However, this method may undermine proofs through reflection unless referential transparency[8] is guaranteed in the target language constant. For example, consider the ML function `serial`, which maintains a counter and returns the number of times it is called. Linking an Isabelle constant, say `time`, to the target language constant `serial` breaks referential transparency:

```
consts time :: "unit ⇒ integer"

code_printing constant time ⇀ (SML) "serial"
```

we have

```
value "time () = time ()"
```

which returns `false` and breaks reflexivity. This example is due to Lochbihler and Züst [63]. In general, I believe this is an inevitable risk whenever we want to reason with results brought back by **code_printing**.

Fortunately, exact algebraic arithmetic is the only part in my thesis that inevitably depends on **code_printing**. All other verified procedures (e.g., `bsgn_at`) and proof tactics, which I will present later in this thesis, do not require this code-printing technique hence can be free of this risk – their execution can be boosted through compilation into native code, but such compilation can be switched off and the evaluation will then be treated as a process of

[8] Programs always return the same value and have the same effect if they are given the same input.

rewritings, which provides the highest trustability. Interested readers can consult the code generation tutorial [37, §5] for various evaluation techniques.

4.5.3 Intended applications

Although I have managed to build exact algebraic arithmetic in this chapter, the main objective is, actually, the univariate/bivariate sign determination procedure at real algebraic points that uses only (dyadic) rational arithmetic. This is due to my long-term goal of building practical decision procedures, and exact real algebraic arithmetic is rarely used in modern computer algebra systems due to its extreme inefficiency. For example, in the lifting phase of a CAD procedure, we may need to isolate the real roots of a polynomial with real algebraic coefficients. Modern approaches usually use sophisticated techniques to soundly approximate those coefficients to a certain precision rather than carrying out exact algebraic arithmetic [79, 18, 82], relying on exact symbolic procedures as a fall-back in degenerate cases.

Following these efficient modern approaches, my sign determination procedures can be improved in at least the following ways:

- Sophisticated interval arithmetic can be used to decide the sign before resorting to a remainder sequence, as has been done in Z3 [29]. This approach should help when the sign is non-zero.
- Pseudo-division, which I am currently using for building remainder sequences, is not good for controlling coefficients growth. More sophisticated approaches, such as subresultant sequences and modular methods, can be used to optimise the calculation of remainder sequences.

Chapter 5

Deciding univariate polynomial problems using untrusted certificates

In this chapter, I present a formally verified procedure based on CAD for univariate polynomial problems with rational coefficients. Goals such as

$$\forall x. \, (x^2 > 2 \wedge x^{10} - 2x^5 + 1 \geq 0) \vee x < 2$$

$$\exists x. \, (x^2 = 2 \wedge (x > 1 \vee x < 0))$$

can be discharged by my tactic automatically.

A key feature of the procedure is its certificate-based design in which an external untrusted (but ideally highly efficient) program is used to find certificates, and those certificates are then checked by verified internal procedures. Overall, the soundness of the procedure depends solely on the soundness of Isabelle's logic (and code generation[1]) rather than trusted external oracles. This is much like Isabelle's sledgehammer tactic, which sceptically incorporates various external tools.

Chapter outline. This chapter continues at follows: A motivating example (§5.1) and a description of the overall design (§5.2) sketch the general idea of our procedure. The main proof is described in (§5.3), which is followed by a discussion of interaction with external solvers (§5.4). Finally, experiments and related work (§5.5) are described.

The material in this chapter is adapted from my joint publication [60] with Paulson and Passmore, where they contributed to the design of the procedure and I carried out all the implementations.

[1]As my tactic is computationally intense, the procedure makes use of the proof by reflection technique [39].

5.1 A motivating example

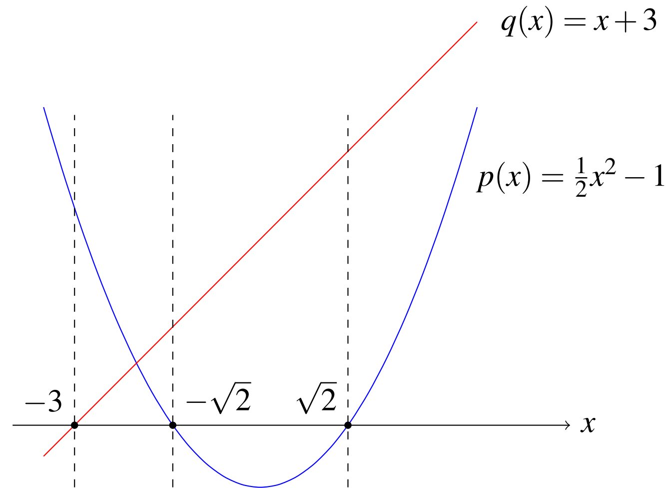

Fig. 5.1 The plot of $p(x) = \frac{1}{2}x^2 - 1$ and $q(x) = x+3$

Unlike the general case of $\mathbb{R}^n$, the restriction of CAD to univariate problems (i.e., to $\mathbb{R}^1$) is relatively straight-forward. Suppose we wish to prove

$$\forall x.\, p(x) > 0 \vee q(x) \geq 0$$

where

$$p(x) = \frac{1}{2}x^2 - 1$$
$$q(x) = x + 3.$$

To do so, we can *decompose* $\mathbb{R}$ into disjoint connected components induced by the roots of p and q. This is illustrated in Fig. 5.1:

$$\mathfrak{D} = \{(-\infty,-3),\{\ -3\ \},(-3,-\sqrt{2}),\{\ -\sqrt{2}\ \},(-\sqrt{2},\sqrt{2}),\{\ \sqrt{2}\ \},(\sqrt{2},+\infty)\}$$

root of q roots of p

and it can be observed that both p and q have invariant signs over each of these components. For example, as can be seen from Fig. 5.1, $p(x) < 0$ and $q(x) > 0$ hold for all $x \in (-\sqrt{2},\sqrt{2})$. To decide the conjecture, we can pick sample points from each of these components and evaluate $\lambda x.\, p(x) > 0 \vee q(x) \geq 0$ at these points. That is,

$$
\begin{aligned}
&\forall x.\, p(x) > 0 \lor q(x) \geq 0 \\
&= \forall D \in \mathfrak{D}.\, \forall x \in D.\, p(x) > 0 \lor q(x) \geq 0 \\
&= \forall x \in \{-4, -3, -2, -\sqrt{2}, 0, \sqrt{2}, 2\}.\, p(x) > 0 \lor q(x) \geq 0 \\
&= (p(-4) > 0 \lor q(-4) \geq 0) \land (p(-3) > 0 \lor q(-3) \geq 0) \land \cdots \\
&\quad\ \land (p(2) > 0 \lor q(2) \geq 0) \\
&= \text{True}
\end{aligned}
\tag{5.1}
$$

since

$$
\begin{aligned}
-4 &\in (-\infty, -3) \\
-3 &\in \{-3\} \\
-2 &\in (-3, -\sqrt{2}) \\
-\sqrt{2} &\in \{-\sqrt{2}\} \\
0 &\in (-\sqrt{2}, \sqrt{2}) \\
\sqrt{2} &\in \{\sqrt{2}\} \\
2 &\in (\sqrt{2}, +\infty).
\end{aligned}
$$

Analogously, to decide an existential formula

$$\exists x.\, p(x) = 0 \land q(x) > 0,$$

we have

$$
\begin{aligned}
&\exists x.\, p(x) = 0 \land q(x) > 0 \\
&= \exists D \in \mathfrak{D}.\, \exists x \in D.\, p(x) = 0 \land q(x) > 0 \\
&= \exists x \in \{-4, -3, -2, -\sqrt{2}, 0, \sqrt{2}, 2\}.\, P(x) = 0 \land q(x) > 0 \\
&= (p(-4) = 0 \land q(-4) > 0) \lor (p(-3) = 0 \land q(-3) > 0) \lor \cdots \\
&\quad\ \lor (p(2) = 0 \land q(2) > 0) \\
&= \text{True}.
\end{aligned}
\tag{5.2}
$$

In performing these arguments, there were a few “obvious” subtleties:

- The decomposition of $\mathbb{R}$ into the seven regions given *covered* the entire real line. That is,

$$(-\infty,-3) \cup \{-3\} \cup (-3,-\sqrt{2}) \cup \{-\sqrt{2}\} \cup (-\sqrt{2},\sqrt{2}) \cup \{\sqrt{2}\} \cup (\sqrt{2},+\infty) = \mathbb{R}.$$

- The "sign-invariance" of p and q over each region was exploited to allow only a single sample point to be selected from each region. This property holds as by the intermediate value theorem, p and q can only change sign by passing through a root.

- The signs of univariate polynomials were evaluated at irrational real algebraic points like $\sqrt{2}$ to determine the truth values of atomic formulas.

In creating our automatic proof procedure, all of this routine reasoning must, of course, be formalised. Moreover, the isolation of polynomial roots (and thus sign-invariant regions) and the sign determination for polynomials at real algebraic points are computationally expensive operations. Computer algebra systems like Mathematica have decades of tuning in their implementations of these core algebraic algorithms. To have a practical proof procedure, we wish to take advantage of these highly tuned external tools as much as possible. Let me next describe how this can be done.

5.2 A sketch of the certificate-based design

There is a rich history of certificate-based, sceptical integrations between proof assistants and external solvers. Examples include John Harrison's sums-of-squares method [42] and the Sledgehammer [76] command in Isabelle.

Certificate-based approaches are motivated by many observations, including:

- External solvers are often highly tuned and run much faster than verified ones.

- Verification of certificates from external solvers is usually much easier than finding them. Such verification ensures the soundness of the overall tactic.

- Switching between different external solvers does not require changes in formal proofs.

Algorithm 2 sketches my idea for univariate universal formulas. In particular, in line 3, I use external programs to return real roots of polynomials (i.e., P) from the quantifier-free part of the formula (i.e., $F(x)$). Those roots (i.e., *roots*) correspond to a decomposition such that each polynomial from P has a constant sign over each component of this decomposition. Since the roots are returned by untrusted programs, in line 5, I not only check

Algorithm 2 Prove univariate universal formulas over reals

Require: $F(x)$ is a quantifier-free formula over reals
Ensure: Return true if $\forall x. F(x)$ holds
1: **procedure** UNIVERSAL($\forall x. F(x)$)
2: $P \leftarrow$ extract polynomials from $F(x)$ ▷ $P \subseteq \mathbb{Z}[X]$
3: *roots* $\leftarrow$ real roots of P ▷ Roots returned by external programs
4: *samples* $\leftarrow$ construct sample points from *roots*
5: **if** $(\forall x \in samples. F(x)) \wedge$ (*roots* are indeed all real roots of P) **then**
6: **return** true
7: **end if**
8: **end procedure**

$\forall x \in samples. F(x)$ as in Equation (5.1) but also certify that these roots are indeed all real roots of P.

The step in line 3 in Algorithm 2 is more commonly referred as *(real) root isolation*, which is a classic and well-studied topic in symbolic computing. Although I can in principle formalise a root isolation procedure (e.g., using the Sturm-Tarski theorem), it is utterly unlikely that my implementation will be competitive with state-of-the-art ones, especially for polynomials of high degree, large bit-width, or whose roots are very close together. Therefore, I delegate this computationally expensive step to external tools.

Algorithm 3 Prove univariate existential formulas over reals

Require: $F(x)$ is a quantifier-free formula over reals
Ensure: Return true if $\exists x. F(x)$ holds
1: **procedure** EXISTENTIAL($\exists x. F(x)$)
2: $r \leftarrow$ solution to $F(x)$ ▷ Solution returned by external programs
3: **if** $F(r)$ **then**
4: **return** true
5: **end if**
6: **end procedure**

With existential formulas, the situation is even simpler as illustrated in Algorithm 3, since we do not need to deal with the decomposition internally. Rather, all we need is a real algebraic witness that satisfies $\lambda x. F(x)$ to certify $\exists x. F(x)$. What is more interesting is that the satisfaction problem for $\lambda x. F(x)$ can be not only solved by a CAD procedure, which is complete but not very fast due to its symbolic nature, but also be complemented by highly efficient incomplete numerical methods. Thus it is natural to externalise the step in line 2 in Algorithm 3.

It is worth noting that both Algorithm 2 and 3 serve as proof tactics rather than decision procedures – if our target goal is actually false, the tactic will simply fail (instead of returning false). On the other hand, if the tactic fails, there could be a false goal, a bug in the tactic or a bug in the external program, but none of these will affect the soundness of the tactic (provided we trust Isabelle's kernel). This is a common practice in interactive theorem proving, where soundness is valued over completeness. Finally, to check that Algorithm 2 or 3 fails due to false goals, we can always try to invoke the tactics on the negation of the previous goals.

5.3 The formal development of the proof procedure

In this section, I describe the formal development of the tactics sketched by Algorithms 2 and 3.

5.3.1 Parsing formulas

The first step of the tactic is to parse the target formula into a structured form. This process is usually referred as reification [16] in Isabelle/HOL. More specifically, given an Isabelle/HOL term e of type τ, we can define a (more structured) datatype δ and an interpretation function *interp* of type $\delta \Rightarrow \tau\ list \Rightarrow \tau$, such that for some e‘ of type δ

$$e = interp\ e`\ xs$$

where xs is a list of free variables in e. Subsequently, instead of directly dealing with e, we now convert it into a more pleasant form *interp e‘ xs* where e‘ is in fact a formal language that captures the structure of e.

The datatypes I defined to capture the structure of target univariate formulas are as follows:

datatype `num = C real` — Constant
  `| Var nat` — Variable index
  `| Add num num | Minus num | Mul num num | Power num nat`

datatype `norm_num2 =`
  `Pol "int poly" nat` — an integer polynomial and its variable index
  `| Const real` — constant
  `| Abnorm num` — in case of anomalies (e.g., bivariate)

datatype `qf_form2 =`

```
Pos norm_num2 — is positive | Zero norm_num2 — is zero
| Neg qf_form2 — negation
| Conj qf_form2 qf_form2 — conjunction
| Disj qf_form2 qf_form2 — disjunction
| T — true   | F — false
```

datatype `norm_form2 =`

```
QF qf_form2 — quantifier free
| ExQ norm_form2 — existential
| AllQ norm_form2 — universal
```

and the interpretation functions:

```
fun num_interp:: "num ⇒ real list ⇒ real" where
  "num_interp (C i) vs = i"|
  "num_interp (Var v) vs = vs!v"|
  "num_interp (Add num1 num2) vs = num_interp num1 vs + num_interp num2 vs "|
  "num_interp (Minus num) vs = - num_interp num vs "|
  "num_interp (Mul num1 num2) vs = num_interp num1 vs * num_interp num2 vs "|
  "num_interp (Power num n) vs = (num_interp num vs)^n"

fun norm_num2_interp :: "norm_num2 ⇒ real list ⇒ real" where
  "norm_num2_interp (Pol p v) vs = poly (of_int_poly p) (vs!v)"|
  "norm_num2_interp (Const c) vs = c"|
  "norm_num2_interp (Abnorm num) vs = num_interp num vs" — anomaly

fun qf_form2_interp:: "qf_form2 ⇒ real list ⇒ bool" where
  "qf_form2_interp (Pos norm_num) vs = (norm_num2_interp norm_num vs > 0)"|
  "qf_form2_interp (Zero norm_num) vs = (norm_num2_interp norm_num vs = 0)"|
  "qf_form2_interp (Neg qf_form) vs = (¬ qf_form2_interp qf_form vs)" |
  "qf_form2_interp (Conj qf_form1 norm_form2) vs
    = (qf_form2_interp qf_form1 vs ∧ qf_form2_interp norm_form2 vs)"|
  "qf_form2_interp (Disj qf_form1 qf_form2) vs
    = (qf_form2_interp qf_form1 vs ∨ qf_form2_interp qf_form2 vs)"|
  "qf_form2_interp T vs = True"|
  "qf_form2_interp F vs = False"

fun norm_form2_interp:: "norm_form2 ⇒real list ⇒ bool" where
  "norm_form2_interp (QF qf) vs = qf_form2_interp qf vs"|
  "norm_form2_interp (ExQ norm_form) vs
      = (∃x. norm_form2_interp norm_form (x#vs))"|
```

```
  "norm_form2_interp (AllQ norm_form) vs
      = (∀x. norm_form2_interp norm_form (x#vs))"
```

Given the definition of a (structured) datatype *norm_form2* and the corresponding interpretation function *norm_form2_interp*, target formulas can now be parsed. For example, we can convert a univariate formula

```
"∀x::real. x > 1/2 ∨ x < 1"
```

into an equivalent form

```
"norm_form2_interp
   (AllQ (QF (Disj (Pos (Pol [:- 1, 2:] 0))
                   (Pos (Pol [:1, - 1:] 0))
            )))
   []"
```

In particular, note

```
qf_form2_interp (Pos (Pol [:- 1, 2:] 0)) [x]
= (poly [:- 1, 2:] x > 0)
= (x > 1/2)
```

in which inequalities have been parsed into a polynomial sign determination problem.

On the contrary, a bivariate non-closed formula such as

```
"∃x::real. x + y >0"
```

will be converted into

```
"norm_form2_interp
  (ExQ (QF (Pos
        (Abnorm
          (Add (Add (Add (C 0) (Mul (Var 0) (Add (C 1) (Mul (Var 0) (C 0)))))
                    (Add (C 0) (Mul (Var 1) (Add (C 1) (Mul (Var 1) (C 0))))))
               (C 0))))))
     [y]"
```

where the *Abnorm* constructor indicates that such formula is not supported by the current tactic.

5.3.2 Existential case

To discharge a univariate existential formula is easy: we can computationally check if a certificate (i.e., a real algebraic number) returned by an external solver satisfies the quantifier-free part of the formula:

Lemma 5.1 (*ExQ_intro*).

```
fixes x::"alg_float" and qf_form::qf_form2
assumes "qf_form2_interp qf_form [of_alg_float x]"
shows "norm_form2_interp (ExQ (QF qf_form)) []"
```

where *x* of type `alg_float`

```
datatype alg_float =
    Arep "int poly" float float — representation of a real algebraic number
  | Flt float — a small optimization in case the number is dyadic rational
```

is a certificate that is supposed to be instantiated by an external solver. The function *of_alg_float* converts *x* from `alg_float` to *real*. In other words, to prove an existential formula:

```
"norm_form2_interp (ExQ (QF qf_form)) []"
```

we can computationally check the truth value of the quantifier-free part of the formula at *x*:

```
"qf_form2_interp qf_form [of_alg_float x]"
```

which is possible due to the univariate sign determination procedure described in §4.2.2 of Chapter 4.

5.3.3 Universal case

For the universal case, the core lemma is as follows:

Lemma 5.2 (*utilize_samples*).

```
fixes P::"real ⇒ bool" and decomps::"real set set"
  and samples::"real set" and f::"real set ⇒ real"
assumes "⋃decomps = ℝ"
    and "∀d∈decomps. ∀x1∈d.∀x2∈d. P x1 = P x2"
    and "∀d∈decomps. f d∈d" and "bij_betw f decomps samples"
shows "(∀x. P x) = (∀pt∈samples. P pt)"
```

where `bij_betw f decomps samples` states that `f::real set ⇒ real` is a bijective function between the decomposition `decomps::real set set` and the sample points `samples::real set`. Essentially, what Lemma 5.2 shows is that given a predicate `P::real ⇒ bool`, an unbounded universal formula `∀x. P x` is equivalent to a bounded one `∀pt∈samples. P pt`, if the truth value of `P` is constant over each component of the decomposition: `∀d∈decomps. ∀x1∈d. ∀x2∈d. P x1 = P x2`.

On top of Lemma 5.2, I similarly convert an unbounded univariate real formula into a bounded one:

Lemma 5.3 (`allQ_subst`).

```
fixes root_reps::"alg_float list" and pols::"float poly set"
  and qf_form::qf_form2
defines "samples ≡ map of_alg_float (mk_samples root_reps)"
assumes "Some pols = extractPols qf_form"
    and "ordered_reps root_reps"
    and "contain_all_roots root_reps pols"
    and "valid_list root_reps"
shows "norm_form2_interp (AllQ (QF qf_form)) vs
  = (∀x ∈ (set samples). norm_form2_interp (QF qf_form) (x#vs))"
```

where

- `root_reps::alg_float list` is a certificate that should be instantiated by an external solver. More specifically, `root_reps` should be the representation of a list of real roots (in ascending order) of polynomials from the quantifier-free part of the target formula,
- `map of_alg_float (mk_samples root_reps)` constructs sample points from the representation of a list of roots,
- `extractPols qf_form` extracts polynomials from the quantifier-free part `qf_form`,
- `ordered_reps root_reps` and `valid_list root_reps` together ensure that the representation of roots are valid and those roots are in ascending order,
- `contain_all_roots roots_reps pols` checks if `root_reps` is a representation of all real roots of the polynomials `pols`. Specifically, by Sturm's theorem, the number of total distinct real roots of each `p ∈ pols` can be computed, which can be then compared with the number of `r ∈ root_reps` that `p(r)=0`.

Most importantly, all assumptions of Lemma 5.3 and its right-hand side

```
(∀x ∈ (set samples). norm_form2_interp (QF qf_form) (x#vs))
```

can be computationally checked, through which we can prove an unbounded univariate universal formula: `norm_form2_interp (AllQ (QF qf_form)) vs`.

5.4 Linking to an external solver

Certificates for both existential and universal cases can be produced by any program performing univariate CAD. For now, I implement the program on top of Mathematica. More specifically, the universal certificates are constructed by the Mathematica command *SemialgebraicComponentInstances*, which gives sample points in each connected component of a semialgebraic set. The existential certificates are constructed by the command *FindInstance*, which incorporates powerful numerical methods to accelerate the search for real algebraic sample points.

Also, it may be worth mentioning that after a certificate has been found, the tactic will record it (as a string) so that repeating the proof no longer requires the external solver. This is much like the sums-of-squares tactic [42].

In general, the certificate-based design grants us much flexibility: We can easily switch to a more efficient external solver without modifying existing formal proofs. In fact, I was first using an implementation of univariate CAD built within MetiTarski, which turned out to be not very efficient, and I painlessly switched to the current one based on Mathematica. In the future, I plan to experiment with other open-source CAD implementations such as Z3 and QEPCAD to provide more options with external solvers.

5.5 Experiments and related work

The most relevant work is the recent `tarski` strategy by Narkawicz et al. [71] in PVS. Both their work and mine rely on a formal proof of the Sturm-Tarski theorem (which they call Tarski's theorem) and handle roughly the same class of problems[2] (i.e., first-order univariate formulas over reals). There are two main differences between their work and mine:

- Their procedure resembles Tarski's original quantifier elimination [8, Chapter 2] and Mahboubi and Cohen's quantifier elimination procedure in Coq [65] by making use of both the Sturm-Tarski theorem and matrices. In contrast, my tactic is based on CAD and real algebraic numbers (instead of matrices).

[2] In fact, their tactic does not handle arbitrary boolean expressions like ours, but I believe this should not be too hard to overcome.

$$
\begin{array}{ll}
\text{ex1}: & \forall x. \neg(x \geq -9 \wedge x < 10 \wedge x^4 > 0) \vee x^{12} > 0 \\
\text{ex2}: & \forall x. \neg((x-2)^2(-x+4) > 0 \wedge x^2(x-3)^2 \geq 0 \\
 & \wedge x-1 \geq 0 \wedge -(x-2)^2+1 > 0) \vee (-(x-\frac{11}{12}))^3(x-\frac{41}{10})^3 \geq 0 \\
\text{ex3}: & \exists x. x^5-x-1=0 \wedge x^{12}+\frac{425}{23}x^{11}-\frac{228}{23}x^{10}-2x^8-\frac{896}{23}x^7-\frac{394}{23}x^6+ \\
 & \frac{456}{23}x^5+x^4+\frac{471}{23}x^3+\frac{645}{23}x^2-\frac{31}{23}x-\frac{228}{23}=0 \wedge x^3+22x^2-31 \geq 0 \\
 & \wedge x^{22}-\frac{234}{567}x^{20}-419x^{10}+1948 > 0 \\
\text{ex4}: & \forall x. x > 0 \vee \frac{20}{9}x^3+\frac{5}{9}x^2-\frac{61}{9}x > -4 \vee 1 \leq x \vee x \leq 0 \vee \frac{10}{9}x^2-\frac{19}{9}x \leq -1 \\
 & \vee \frac{1}{18}x^3+\frac{31}{45}x^2-\frac{13}{9}x \leq -\frac{7}{10} \vee \frac{20}{9}x^3+\frac{5}{9}x^2-\frac{61}{9}x \leq -4 \\
\text{ex5}: & \forall x. -\frac{x^3}{3}-\frac{10}{3}x^2-\frac{5}{6}x > 0 \vee \frac{1}{3}x^3+\frac{10}{3}x^2+\frac{5}{6}x > 0 \vee 1 \leq x \vee x \leq 0 \\
 & \vee \frac{10}{9}x^2-\frac{19}{9}x \leq -1 \vee \frac{1}{18}x^3+\frac{31}{45}x^2-\frac{13}{9}x \leq -\frac{7}{10} \\
 & \vee \frac{14}{15}x^3-\frac{64}{15}x^2-\frac{101}{30}x \leq -\frac{11}{5} \vee \frac{20}{9}x^3+\frac{5}{9}x^2-\frac{61}{9}x \leq -4 \\
\text{ex6}: & \exists x. -70x^6-\frac{2052}{5}x^5-\frac{4329}{5}x^4-\frac{5409}{10}x^3-\frac{267}{2}x^2-\frac{51}{10}x > -\frac{7}{10} \wedge \frac{49}{162}x^9+\frac{49}{3}x^8 \\
 & +\frac{175}{18}x^7+\frac{115774}{405}x^6+\frac{77743}{135}x^5-\frac{57328}{135}x^4-\frac{135853}{810}x^3-\frac{71681}{270}x^2-\frac{10327}{270}x > -\frac{721}{90} \\
 & \wedge \frac{7}{27}x^8+\frac{280}{27}x^7-\frac{595}{54}x^6+\frac{18964}{135}x^5+\frac{2698}{135}x^4-\frac{24217}{270}x^3-\frac{251}{6}x^2-\frac{2981}{90}x > -\frac{206}{45} \\
 & \wedge \frac{7}{54}x^7+\frac{112}{27}x^6+\frac{329}{90}x^5+\frac{2672}{135}x^4-\frac{7933}{270}x^3+\frac{169}{18}x^2-\frac{799}{90}x > -\frac{103}{90} \wedge \frac{7}{27}x^8+\frac{280}{27}x^7 \\
 & +\frac{935}{54}x^6+\frac{7264}{135}x^5+\frac{11323}{135}x^4-\frac{12217}{270}x^3-\frac{701}{6}x^2-\frac{781}{90}x > -\frac{77}{15} \wedge \frac{2}{9}x^7+\frac{52}{9}x^6-\frac{17}{6}x^5 \\
 & +\frac{2353}{90}x^4+\frac{307}{45}x^3-\frac{811}{30}x^2-\frac{361}{30}x > -\frac{44}{15} \wedge \frac{1}{9}x^6+2x^5+\frac{2}{15}x^4+\frac{41}{90}x^3-\frac{2}{15}x^2 \\
 & -\frac{33}{10}x > -\frac{11}{15} \wedge \frac{49}{162}x^8+\frac{1540}{81}x^7+\frac{1109}{27}x^6+\frac{23483}{810}x^5+\frac{65378}{405}x^4-\frac{11549}{270}x^3-\frac{70225}{324}x^2 \\
 & -\frac{1339}{405}x > -\frac{721}{60} \wedge \frac{7}{27}x^7+\frac{203}{18}x^6-\frac{52}{9}x^5+\frac{7753}{270}x^4+\frac{5191}{180}x^3-\frac{2263}{45}x^2-\frac{10741}{540}x > -\frac{103}{15} \\
 & \wedge \frac{2}{9}x^6+\frac{59}{9}x^5-\frac{493}{36}x^4+\frac{2113}{90}x^3-\frac{811}{180}x^2-\frac{1481}{90}x > -\frac{22}{5} \wedge \frac{1}{9}x^5+\frac{17}{9}x^4-\frac{257}{60}x^3+\frac{563}{90}x^2 \\
 & -\frac{913}{180}x > -\frac{11}{10} \wedge \frac{20}{9}x^4-\frac{5}{2}x^3+\frac{10}{3}x^2-\frac{91}{18}x > -2 \wedge \frac{10}{9}x^3-\frac{25}{18}x^2-\frac{2}{9}x > -\frac{1}{2} \wedge \frac{20}{9}x^3 \\
 & +\frac{5}{9}x^2-\frac{61}{9}x > -4 \wedge 1 > x \wedge x > 0 \wedge \frac{10}{9}x^2-\frac{19}{9}x > -1 \wedge \frac{1}{18}x^3+\frac{31}{45}x^2-\frac{13}{9}x > -\frac{7}{10} \wedge \frac{1}{9}x^4 \\
 & +\frac{34}{15}x^3-\frac{53}{30}x^2-\frac{253}{90}x > -\frac{11}{5} \wedge \frac{2}{9}x^5+\frac{82}{9}x^4+\frac{86}{15}x^3-\frac{2051}{90}x^2-\frac{97}{90}x > -\frac{44}{5} \wedge \frac{8}{81}x^8 \\
 & +\frac{931}{81}x^7+\frac{3113}{27}x^6-\frac{289811}{1620}x^5+\frac{264373}{810}x^4+\frac{30583}{270}x^3-\frac{298609}{810}x^2-\frac{93307}{1620}x > -\frac{193}{5} \\
 & \wedge \frac{7}{27}x^7+\frac{38}{3}x^6+\frac{28}{9}x^5-\frac{2686}{135}x^4+\frac{6397}{60}x^3-\frac{9151}{90}x^2-\frac{4741}{540}x > -\frac{77}{10}
\end{array}
$$

$$\begin{aligned}
\text{ex7}:\quad & \forall x. x < -1 \lor 0 > x \lor \frac{1}{8}x^7 + \frac{1207}{35}x^6 + \frac{7083}{10}x^5 + 4983x^4 + \frac{64405}{4}x^3 + 26169x^2 \\
& + \frac{41613}{2}x > -6435 \lor 35x^{12} + 22461058620x^2 + 11821609800x \leq 46204x^{11} \\
& + 5263834x^{10} + 144537452x^9 + 1758662439x^8 + 10317027768x^7 + 31842714428x^6 \\
& + 54212099480x^5 + 45938678170x^4 + 4171407240x^3 \lor x \leq 0 \lor 753x^{10} + 58568x^9 \\
& + 938908x^8 + 6857016x^7 + 27930066x^6 + 68338600x^5 + 102560612x^4 + 92372280x^3 \\
& + 45805760x^2 + 9609600x \leq 0 \lor 10x^{11} + 1101329460x^2 + 788107320x \leq 9179x^{10} \\
& + 1061504x^9 + 24397102x^8 + 240283734x^7 + 1063536663x^6 + 2362290448x^5 \\
& + 2625491260x^4 + 782617220x^3 \lor 5x^{10} + 81290790x^2 + 90935460x \leq 2828x^9 \\
& + 356071x^8 + 6846880x^7 + 51834563x^6 + 161529144x^5 + 237512625x^4 \\
& + 125595120x^3 \lor 207x^9 + 11237x^8 + 138652x^7 + 794964x^6 + 2505504x^5 + 4581220x^4 \\
& + 4837448x^3 + 2735040x^2 + 640640x \leq 0 \lor 5x^8 \leq 608x^7 + 10261x^6 + 63520x^5 \\
& + 192458x^4 + 303324x^3 + 238560x^2 + 73920x \lor 98x^8 + 3514x^7 + 32711x^6 + 142928x^5 \\
& + 332962x^4 + 424284x^3 + 278880x^2 + 73920x \leq 0 \lor x \leq -1
\end{aligned}$$

	Time (s)		
Formula	univ_rcf (Isabelle)	univ_rcf_cert (Isabelle)	tarski (PVS)
ex1	0.9	0.3	2.0
ex2	1.4	0.6	6.8
ex3	1.6	0.7	13.0
ex4	1.3	0.5	20.1
ex5	1.6	0.6	315.7
ex6	5.6	3.9	timeout
ex7	38.4	34.9	timeout

Note: timeout indicates failure to terminate within 24 hours

Fig. 5.2 Comparison between my tactic in Isabelle and the tarski strategy in PVS: univ_rcf includes certificate searching and checking, while univ_rcf_cert includes only checking

- Their procedure is entirely built within PVS, while mine sceptically makes use of efficient external programs to generate certificates.

To compare both tactics empirically, I have conducted experiments on several typical examples from their paper[3] and the MetiTarski project[4] [74]. The experiments are run on a desktop with an Intel Core 2 Quad Q9400 (quad core, 2.66 GHz) CPU and 8 gigabytes RAM. Results of the experiments are illustrated in Fig. 5.2, where my `univ_rcf` tactic includes both certificate searching and checking process, while the `univ_rcf_cert` does the checking part only (when repeating a proof with certificates already recorded as a string).

In general, the experiments indicate that my tactic outperforms the `tarski` strategy in PVS. Particularly, the advantage of my tactic becomes greater as the problems become more complex, which can be attributed to the fact that my tactic has much better worst-case computational complexity (polynomial vs. exponential in the number of polynomials).

In the case of general multivariate problems, the CAD procedure is doubly exponential while Tarski's quantifier elimination procedure is non-elementary in the number of variables [8, Chapter 11]. When limited to univariate problems, the CAD procedure degenerates to root isolation and sign determination on a set of univariate polynomials, which is of polynomial complexity in the number of polynomials and their degree bound [8, Chapter 10]. In comparison, Tarski's quantifier elimination procedure, even when limited to univariate problems, is still exponential in the number of polynomials [65].

In addition, it is worth noting that as the problems become more complex (e.g., ex6 and ex7 in Fig. 5.2), certificate checking becomes the bottleneck factor of my tactic (especially for universal problems). This indicates that, despite the fact that certificate searching is much harder than certificate checking, the Mathematica implementation is still much more efficient than my verified certificate-checking procedure. This leaves much room for future optimisations.

My work has also been greatly inspired by Mahboubi and Cohen's quantifier elimination procedure in Coq [65], which is also similar to Tarski's procedure based on the Strum-Tarski theorem. The fundamental difference between their work and mine is that they aim for theoretical results (i.e., the decidability of real closed fields), while I intend to build practical proof procedures.

Decision procedures based on Sturm's theorem have been implemented in Isabelle and PVS before [33, 72]. Their core idea is to count the number of real roots within a certain (bounded or unbounded) interval. Generally, they can only handle formulas involving a

[3] http://shemesh.larc.nasa.gov/people/cam/Tarski/
[4] http://www.cl.cam.ac.uk/~gp351/cicm2012/

single polynomial, so they are not complete for first-order formulas (unlike my tactic and the `tarski` strategy in PVS).

Assia Mahboubi [64] has implemented the executable part of a general CAD procedure in Coq, but as far as I know, the correctness proof for her implementation is still ongoing. This is also one of the reasons for me to choose the certificate-based approach rather than directly verifying an implementation.

There are other methods to handle non-linear polynomial problems in theorem provers, such as sums of squares [42], which is good for multivariate universal problems but is not applicable when the existential quantifier arises, and interval arithmetic [47, 80], which is very efficient for some cases but is not complete. These methods and mine should be used in a complementary way.

Chapter 6

A formal proof of Cauchy's residue theorem

With the previous Chapter, we are able to certify *univariate* CAD through a certificate-based approach. To proceed to the multivariate case, we need formalised results in complex analysis (e.g. to justify Theorem 2.6 as later discussed in Chapter 8). Therefore, this chapter is devoted to relevant analytical results.

In particular, I will formalise Cauchy's residue theorem along with its immediate consequences: the argument principle and Rouché's theorem. All of them are important results for reasoning about isolated singularities and zeros of holomorphic functions in complex analysis. For example, other than justifying CAD, Cauchy's residue theorem can also be used to evaluate improper integrals like

$$\int_{-\infty}^{+\infty} \frac{e^{itz}}{z^2+1} dz = \pi e^{-|t|}.$$

Proofs in this chapter mainly follow standard textbooks [7, 55, 81], with minor improvements as discussed in the chapter later.

Chapter outline. The chapter begins with some background on complex analysis (§6.1), followed by a proof of the residue theorem, then the argument principle and Rouché's theorem (§6.2–6.4). Then there is a brief discussion of related work (§6.5).

The content of this chapter is adapted from the joint publication with Paulson [61], where he ported the material described in the background section (§6.1) from HOL Light to Isabelle while I finished the remaining proofs.

6.1 Background

I briefly introduce some basic complex analysis from Isabelle/HOL's Analysis library. Most of the material in this section was first formalised in HOL Light by John Harrison [41] and later ported to Isabelle by Larry Paulson.

6.1.1 Contour integrals

Given a path γ, a map from the real interval $[0,1]$ to $\mathbb{C}$, the contour integral of a complex-valued function f on γ can be defined as

$$\oint_\gamma f = \int_0^1 f(\gamma(t))\gamma'(t)dt.$$

Because integrals do not always exist, this notion is formalised as a relation:

```
definition has_contour_integral ::
    "(complex ⇒ complex) ⇒ complex ⇒ (real ⇒ complex) ⇒ bool"
        (infixr "has'_contour'_integral" 50)
  where "(f has_contour_integral i) g ≡
        ((λx. f(g x) * vector_derivative g (at x within {0..1}))
         has_integral i) {0..1}"
```

We can introduce an operator for the integral to use in situations when we know that the integral exists. This is analogous to the treatment of ordinary integrals, derivatives, etc., in HOL Light [41] as well as Isabelle/HOL.

6.1.2 Valid paths

In order to guarantee the existence of the contour integral, we need to place some restrictions on paths. A *valid path* is a piecewise continuously differentiable function on `[0..1]`. In plain English, the function must have a derivative on all but finitely many points, and this derivative must also be continuous.

```
definition piecewise_C1_differentiable_on
         :: "(real ⇒ 'a :: real_normed_vector) ⇒real set ⇒ bool"
        (infixr "piecewise'_C1'_differentiable'_on" 50)
  where "f piecewise_C1_differentiable_on i ≡
        continuous_on i f ∧
        (∃s. finite s ∧ (f C1_differentiable_on (i - s)))"
```

```
definition valid_path :: "(real ⇒ 'a :: real_normed_vector) ⇒ bool"
  where "valid_path f ≡ f piecewise_C1_differentiable_on {0..1::real}"
```

6.1.3 Winding number

The winding number of the path γ at the point w is defined (following textbook definitions) as

$$n(\gamma,z) = \frac{1}{2\pi i}\oint_\gamma \frac{dw}{w-z}.$$

A lemma to illustrate this definition is as follows:

Lemma 6.1 (`winding_number_valid_path`).

```
fixes γ::"real ⇒ complex" and z::complex
assumes "valid_path γ" and "z ∉ path_image γ"
shows "winding_number γ z
        = 1/(2*pi*i) * contour_integral γ (λw. 1/(w - z))"
```

6.1.4 Holomorphic functions and Cauchy's integral theorem

A function is *holomorphic* if it is complex differentiable in a neighbourhood of every point in its domain. The Isabelle/HOL version follows that of HOL Light:

```
definition holomorphic_on::"(complex ⇒ complex) ⇒ complex set ⇒ bool"
    (infixl "(holomorphic'_on)" 50)
  where "f holomorphic_on s ≡ ∀x∈s. f field_differentiable (at x within s)"
```

As a starting point to reason about holomorphic functions, it is fortunate that John Harrison has made the effort to prove Cauchy's integral theorem in a rather general form:

Theorem 6.2 (`Cauchy_theorem_global`).

```
fixes s::"complex set" and f::"complex ⇒ complex"
assumes "open s" and "f holomorphic_on s"
  and "valid_path γ" and "pathfinish γ = pathstart γ"
  and "path_image γ ⊆ s"
  and "⋀w. w ∉ s ⟹ winding_number γ w = 0"
shows "(f has_contour_integral 0) γ"
```

Note, a more common statement of Cauchy's integral theorem requires the open set s to be simply connected (connected and without holes). Here, the simply connectedness is encoded by a homologous assumption

```
"⋀w. w ∉ s ⟹ winding_number γ w = 0".
```

The reason behind this homologous assumption is that a non-simply-connected set s should contain a cycle γ and a point a within one of its holes, such that `winding_number γ a` is non-zero. Statements of such homologous version of Cauchy's integral theorem can be found in standard texts[1, 55].

6.2 Cauchy's residue theorem

As a result of Cauchy's integral theorem, if f is a holomorphic function on a simply connected open set s which contains a closed path γ, then

$$\oint_\gamma f(w) = 0.$$

However, if the set s does have a hole, then Cauchy's integral theorem will not apply. For example, consider $f(w) = \frac{1}{w}$ so that f has a pole at $w = 0$, and γ is the circular path $\gamma(t) = e^{2\pi i t}$:

$$\oint_\gamma \frac{dw}{w} = \int_0^1 \frac{1}{e^{2\pi i t}} \left(\frac{d}{dt} e^{2\pi i t}\right) dt = \int_0^1 2\pi i dt = 2\pi i \neq 0.$$

Cauchy's residue theorem applies when a function is holomorphic on an open set except for a finite number of points (i.e., isolated singularities):

Theorem 6.3 (*Residue_theorem*).

```
fixes s pts::"complex set" and f::"complex ⇒ complex"
  and γ::"real ⇒ complex"
assumes "open s" and "connected s" and "finite pts" and
        "f holomorphic_on s - pts" and
        "valid_path γ" and
        "γ 0 = γ 1" and
        "path_image γ ⊆ s - pts" and
        "∀z. (z ∉ s) ⟶ winding_number γ z  = 0"
shows "contour_integral γ f
      = 2 * pi * i *(∑p∈pts. winding_number γ p * residue f p)"
```

where `residue f p` denotes the residue of `f` at `p`, which I will describe in details in the next subsection.

6.2.1 Residue

A complex function f is defined to have an *isolated singularity* at point z, if f is holomorphic on an open disc centered at z but not at z.

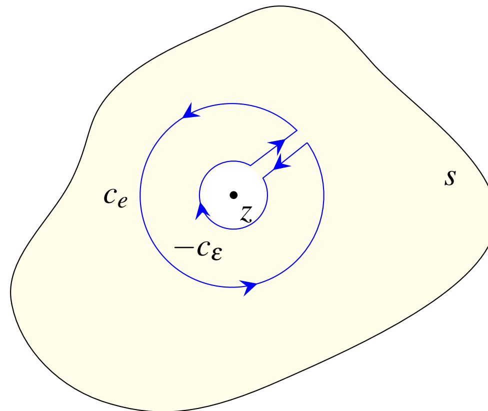

Fig. 6.1 Circlepath c_e and c_ε around an isolated singularity z

We can now define `residue f z` to be the path integral of f (divided by a constant $2\pi i$) along a small circular path around z:

```
definition residue::"(complex ⇒ complex) ⇒ complex ⇒ complex" where
  "residue f z = (SOME int. ∃e>0. ∀ε>0. ε<e
    ⟶ (f has_contour_integral 2 * pi * i * int) (circlepath z ε))"
```

To actually utilise our definition, we need not only to show the existence of such integral but also its invariance when the radius of the circular path becomes sufficiently small.

Lemma 6.4 (`base_residue`).

```
  fixes s::"complex set" and f::"complex ⇒ complex"
    and e::real and z::complex
  assumes "open s" and "z ∈ s" and "e > 0"
    and "f holomorphic_on (s - {z})" and "cball z e ⊆ s"
  shows "(f has_contour_integral 2 * pi * i * residue f z) (circlepath z e)"
```

Here `cball` denotes the familiar concept of a closed ball:

```
definition cball :: "'a::metric_space ⇒ real ⇒ 'a set"
  where "cball x e = {y. dist x y ≤ e}"
```

Proof. Given two small circular path c_ε and c_e around z with radius ε and e respectively, we want to show that

$$\oint_{c_\varepsilon} f = \oint_{c_e} f.$$

Let γ is a line path from the end of c_e to the start of $-c_\varepsilon$. As illustrated in Fig. 6.1, consider the path

$$\Gamma = c_e + \gamma + (-c_\varepsilon) + (-\gamma),$$

where $+$ is path concatenation, and $-c_\varepsilon$ and $-\gamma$ are reverse paths of c_ε and γ respectively. As Γ is a valid closed path and f is holomorphic on the interior of Γ, we have

$$\oint_\Gamma f = \oint_{c_e} f + \oint_\gamma f + (-\oint_{c_\varepsilon} f) + (-\oint_\gamma f) = \oint_{c_e} f - \oint_{c_\varepsilon} f = 0,$$

hence

$$\oint_{c_\varepsilon} f = \oint_{c_e} f,$$

and the proof is completed. □

6.2.2 Generalisation to a finite number of singularities

Lemma 6.4 can be viewed as a special case of Lemma 6.3 where there is only one singularity point and γ is a circular path. In this section, I will describe the proofs of generalising Lemma 6.4 to a plane with finite number of singularities.

It is fortunate that Isabelle has already had results about open connected sets being valid path connected (recall that valid paths are piecewise continuous differentiable functions on the closed interval $[0,1]$):

Lemma 6.5 (*connected_open_polynomial_connected*).

```
fixes s::"'a::euclidean_space set" and x y::'a
assumes "open s" and "connected s" and "x ∈ s" and "y ∈ s"
shows "∃g. polynomial_function g ∧ path_image g ⊆ s ∧
           g 0 = x ∧ g 1 = y"
```

Lemma 6.6 (*valid_path_polynomial_function*).

```
fixes p::"real ⇒ 'a::euclidean_space"
shows "polynomial_function p ⟹ valid_path p"
```

Following this, I manage to derive a valid path γ on some connected punctured set such that a holomorphic function has an integral along γ:

Lemma 6.7 (*get_integrable_path*).

```
fixes s pts::"complex set" and a b::complex and f::"complex ⇒ complex"
assumes "open s" and "connected (s - pts)" and "finite pts"
  and "f holomorphic_on (s - pts) "
  and "a ∈ s - pts" and "b ∈ s - pts"
obtains γ where
  "valid_path γ" and "pathstart γ = a" and "pathfinish γ = b"
  and "path_image γ ⊆ s - pts" and "f contour_integrable_on γ"
```

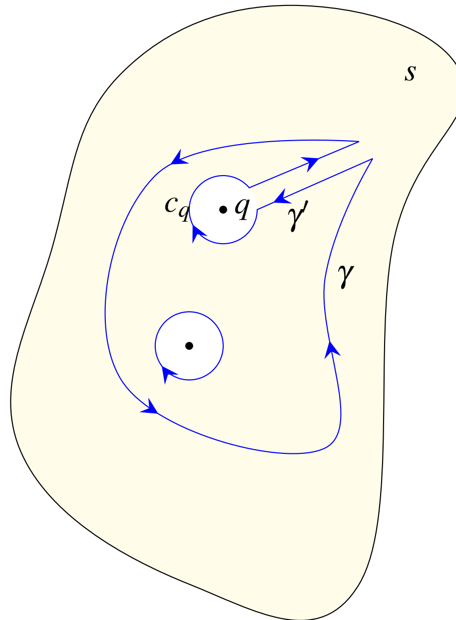

Fig. 6.2 Induction on the number of singularities

Finally, I obtain a lemma that reduces the integral along γ to a sum of integrals over small circles around singularities:

Lemma 6.8 (*Cauchy_theorem_singularities*).

```
fixes s pts::"complex set" and f::"complex ⇒ complex"
    and γ::"real ⇒ complex" and h::"complex ⇒ real"
assumes "open s" and "connected (s - pts)" and "finite pts"
    and "f holomorphic_on s - pts" and "valid_path γ"
    and "γ 1 = γ 0" and "path_image γ ⊆ s - pts"
    and "∀z. (z ∉ s) ⟶ winding_number γ z  = 0"
    and "∀p∈s. h p>0 ∧ (∀w∈cball p (h p). w∈s ∧ (w≠p ⟶ w ∉ pts))"
shows "contour_integral γ f = (∑p ∈ pts. winding_number γ p
        * contour_integral (circlepath p (h p)) f)"
```

Proof. Since the number of singularities *pts* is finite, we do induction on them. Assuming the lemma holds when there are *pts* singularities, we aim to show the lemma for $\{q\} \cup pts$.

As illustrated in Fig. 6.2, suppose c_q is a (small) circular path around q, by Lemma 6.7, we can obtain a valid path γ' from the end of γ to the start of c_q such that f has an integral along γ'.

Consider the path

$$\Gamma = \gamma + \gamma' + \underbrace{(-c_q) + \cdots + (-c_q)}_{n(\gamma,q)} + (-\gamma'),$$

where $+$ is path concatenation, $n(\gamma, q)$ is the winding number of the path γ around q, and $-\gamma'$ and $-c_q$ are the reverse path of γ' and c_q respectively. We can show that Γ is a valid cycle path and the induction hypothesis applies to Γ, that is,

$$\oint_{\Gamma} f = \sum_{p \in pts} n(\Gamma, p) \oint_{c_p} f,$$

hence

$$\oint_{\gamma} f + \oint_{\gamma'} f - n(\gamma, q) \oint_{c_q} f - \oint_{\gamma'} f = \sum_{p \in pts} n(\gamma, p) \oint_{c_p} f,$$

and finally

$$\oint_{\gamma} f = \sum_{p \in \{q\} \cup pts} n(\gamma, p) \oint_{c_p} f$$

which concludes the proof. □

By combining Lemma 6.8 and 6.4, we can finish the proof of Cauchy's residue theorem (i.e., Theorem 6.3).

6.2.3 Applications

Besides corollaries like the argument principle and Rouché's theorem, which I will describe later, Cauchy's residue theorem is useful when evaluating improper integrals.

Example 6.9. Evaluating an improper integral:

$$\int_{-\infty}^{+\infty} \frac{dx}{x^2 + 1} = \pi$$

corresponds to the following lemma in Isabelle/HOL:

lemma `"Lim at_top (λr. integral {- r..r} (λx. 1 / (x^2 + 1))) = pi"`

Proof. Let

$$f(z) = \frac{1}{z^2 + 1}.$$

Now $f(z)$ is holomorphic on $\mathbb{C}$ except for two poles when $z = i$ or $z = -i$. We can then construct a semicircular path $\gamma_r + C_r$, where γ_r is a line path from $-r$ to r and C_r is an arc

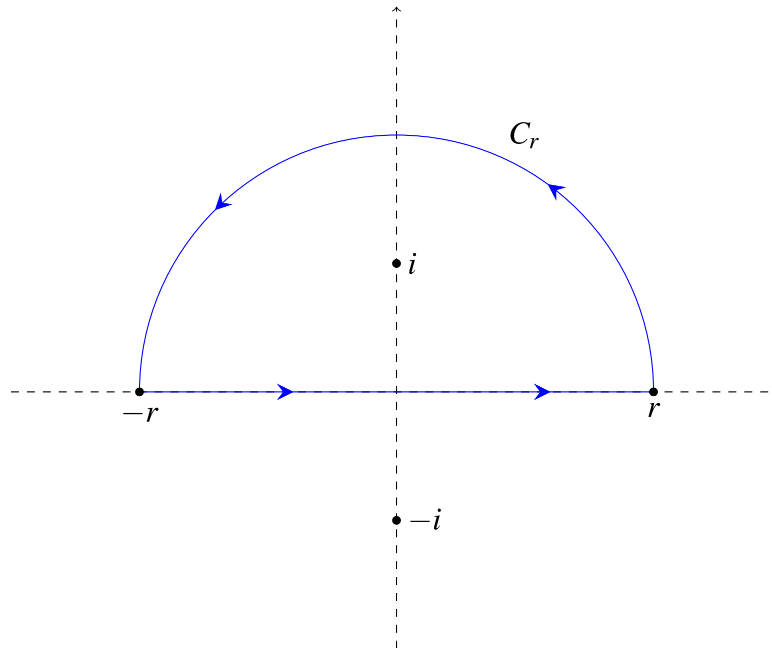

Fig. 6.3 A semicircular path centred at 0 with radius $R > 1$

from r to $-r$, as illustrated in Fig. 6.3. From Cauchy's residue theorem, we obtain

$$\oint_{\gamma_r + C_r} f = \mathrm{Res}(f, i) = \pi$$

where $\mathrm{Res}(f, i)$ is the residue of f at i. Moreover, we have

$$\left| \oint_{C_r} f \right| \leq \frac{1}{r^2 - 1} \pi r$$

as $|f(z)|$ is bounded by $1/(r^2 - 1)$ when z is on C_r and r is large enough. Hence,

$$\oint_{C_r} f \to 0 \quad \text{when} \quad r \to +\infty$$

and therefore

$$\int_{-\infty}^{+\infty} \frac{dx}{x^2 + 1} = \oint_{\gamma_r} f = \oint_{\gamma_r + C_r} f = \pi \quad \text{when} \quad r \to +\infty$$

which concludes the proof. □

Evaluating such improper integrals was difficult for Avigad et al. [6] in their formalisation of the central limit theorem. I hope this development could facilitate such proofs in the future, though it may not be immediate as their proof is based on a different integration operator.

6.2.4 Remarks on the formalisation

It is surprising that I encountered difficulties when generalising Lemma 6.4 to the case of a finite number of poles. Several complex analysis textbooks [25, 81] omit proofs for this part (giving the impression that the proof is trivial). My statement of Lemma 6.8 follows the statement of Theorem 2.4, Chapter IV of Lang [55], but I was reluctant to follow his proof of generalising paths to chains for fear of complicating existing theories. In the end, I devised proofs for this lemma on my own with inspiration from Stein and Shakarchi's concept of a *keyhole* [81].

Another tricky part I have encountered is in the formal proof of Example 6.9. When showing

$$\oint_{\gamma_r + C_r} f = \mathrm{Res}(f, i) = \pi,$$

it is necessary to show i ($-i$) is inside (outside) the semicircle path $\gamma_r + C_r$, that is,

$$n(i, \gamma_r + C_r) = 1 \wedge n(-i, \gamma_r + C_r) = 0,$$

where n is the winding number operation. Such proof is straightforward for humans when looking at Fig. 6.3. However, to formally prove it in Isabelle/HOL, I ended up manually constructing some ad-hoc counter examples and employed proof by contradiction several times. I will discuss this problem in detail in Chapter 7.

6.3 The argument principle

In complex analysis, the *argument principle* is a theorem to describe the difference between the number of zeros and poles of a meromorphic[1] function.

Theorem 6.10 (`argument_principle`).

```
fixes f h::"complex ⇒ complex" and poles s:: "complex set"
defines "zeros ≡ {p. f p = 0} - poles"
assumes "open s" and "connected s" and
        "f holomorphic_on (s - poles)" and
        "h holomorphic_on s" and
        "valid_path γ" and
        "γ 1= γ 0" and
        "path_image γ ⊆ s - (zeros ∪ poles)" and
        "∀z. (z ∉ s) ⟶ winding_number γ z = 0" and
```

[1] holomorphic except for isolated poles

```
        "finite (zeros ∪ poles)" and
        "∀p ∈ poles. is_pole f p"
  shows "contour_integral γ (λx. deriv f x * h x / f x) = 2 * pi * i *
        ((∑p ∈ zeros. winding_number γ p * h p * zorder f p)
         - (∑p ∈ poles. winding_number γ p * h p * porder f p))"
```

where

```
definition is_pole :: "('a::topological_space ⇒ 'b::real_normed_vector)
    ⇒ 'a ⇒ bool" where
  "is_pole f a =  (LIM x (at a). f x :> at_infinity)"
```

encodes the usual definition of poles (i.e., f approaches infinity as x approaches a). `zorder` and `porder` are the order of zeros and poles, which I will define in detail in the next subsection.

6.3.1 Zeros and poles

A complex number z is referred as a *zero* of a holomorphic function f if $f(z) = 0$. And there is a local factorisation property about $f(z)$:

Lemma 6.11 (`holomorphic_factor_zero_Ex1`).

```
  fixes s::"complex set" and f::"complex ⇒ complex" and z::complex
  assumes "open s" and "connected s" and "z ∈ s" and "f(z) = 0"
      and "f holomorphic_on s" and "∃w∈s. f w ≠ 0"
  shows "∃!n. ∃g r. 0 < n ∧ 0 < r ∧ ball z r ⊆ s ∧
                g holomorphic_on ball z r
                ∧ (∀w∈ball z r. f w = (w-z)^n * g w ∧ g w≠0)"
```

Here a *ball*, as usual, is an open neighborhood centred on a given point:

```
definition ball :: "'a::metric_space ⇒ real ⇒ 'a set"
  where "ball x e = {y. dist x y < e}"
```

Proof. [2] As f is holomorphic, f has a power expansion locally around z:

$$f(w) = \sum_{k=0}^{+\infty} a_k (w-z)^k$$

[2]The existence proof of such `n, g and r` is ported from HOL Light, while I have shown the uniqueness of `n` on my own.

and since f does not vanish identically, there exists a smallest n such that $a_n \neq 0$. Therefore,

$$f(w) = \sum_{k=n}^{+\infty} a_k (w-z)^k = (w-z)^n \sum_{k=0}^{+\infty} a_{k+n} (w-z)^k = (w-z)^n g(w),$$

and the function $g(w)$ is holomorphic and non-vanishing near z due to $a_n \neq 0$.

Also, we can show that this n is unique, by assuming there exist m and another locally holomorphic function $h(w)$ such that

$$f(w) = (w-z)^n g(w) = (w-z)^m h(w),$$

and $h(w) \neq 0$. If $m > n$, then

$$g(w) = (w-z)^{m-n} h(w),$$

and this yields $g(w) \to 0$ when $w \to z$, which contradicts the fact that $g(w) \neq 0$. If $n > m$, then similarly $h(w) \to 0$ when $w \to z$, which contradicts $h(w) \neq 0$. Hence, $n = m$, and the proof is completed. □

The unique `n` in Lemma 6.11 is usually referred as the *order/multiplicity of the zero* of `f` at `z`:

```
definition zorder::"(complex ⇒ complex) ⇒ complex ⇒ nat" where
  "zorder f z = (THE n. n > 0 ∧ (∃g r. r > 0 ∧ g holomorphic_on cball z r
                 ∧ (∀w∈cball z r. f w = g w * (w-z) ^ n ∧ g w ≠0)))"
```

We can also refer the complex function g in Lemma 6.11 using Hilbert's epsilon operator in Isabelle/HOL:

```
definition zer_poly::"[complex ⇒ complex, complex]⇒ complex ⇒ complex"
   where
  "zer_poly f z = (SOME g. ∃r. r > 0 ∧ g holomorphic_on cball z r
           ∧ (∀w ∈ cball z r. f w = g w * (w-z) ^ (zorder f z) ∧ g w ≠0))"
```

Given a complex function f has a pole at z while also holomorphic (but not at) z, we know the function

$$\lambda x.\quad \text{if} \quad x = z \quad \text{then} \quad 0 \quad \text{else} \quad 1/\text{f(x)}$$

has a zero at z and is holomorphic near (and at) z. On the top of the definition of the order of zeros, we can define the *order/multiplicity of the pole* of f at z:

```
definition porder::"(complex ⇒ complex) ⇒ complex ⇒ nat" where
  "porder f z = (let f' = (λx. if x = z then 0 else inverse (f x))
                in zorder f' z)"
```

```
definition pol_poly::"[complex ⇒ complex,complex]⇒complex ⇒ complex" where
  "pol_poly f z = (let f' = (λ x. if x=z then 0 else inverse (f x))
                  in inverse o zer_poly f' z)"
```

and a lemma to describe a similar relationship among `f`, `porder` and `pol_poly`:

Lemma 6.12 (`porder_exist`).

```
fixes f::"complex ⇒ complex" and s::"complex set"
  and z::complex
defines "n ≡ porder f z" and "h ≡ pol_poly f z"
assumes "open s" and "connected s" and "z ∈ s"
  and "(λx. if x = z then 0 else inverse (f x)) holomorphic_on s"
  and "∃w∈s. w ≠ z ∧ f w ≠ 0"
shows "∃r. n > 0 ∧ r > 0 ∧ cball z r ⊆ s ∧ h holomorphic_on cball z r
      ∧ (∀w∈cball z r. (w ≠ z ⟶ f w  = h w / (w-z)^n) ∧ h w ≠ 0)"
```

Proof. With Lemma 6.11, we can derive that there exist n and g such that

$$1/f(w) = (w-z)^n g(w),$$

and $g(w) \neq 0$ for w near z. Hence,

$$f(w) = \frac{\frac{1}{g(w)}}{(w-z)^n} = \frac{h(w)}{(w-z)^n}$$

and $h(w) \neq 0$ due to $g(w) \neq 0$. This concludes the proof. □

Moreover, `porder` and `pol_poly` can be used to construct an alternative definition of residue when the singularity is a pole.

Lemma 6.13 (`residue_porder`).

```
fixes f::"complex ⇒ complex" and s::"complex set"
  and z::complex
defines "n≡porder f z" and "h≡pol_poly f z"
assumes "open s" and "connected s" and "z ∈ s"
  and "(λx. if x=z then 0 else inverse (f x)) holomorphic_on s"
  and "∃w∈s. w≠z ∧ f w≠0"
shows "residue f z = ((deriv ^^ (n - 1)) h z / fact (n-1))"
```

Proof. The idea behind Lemma 6.13 is to view $f(w)$ as $h(w)/(w-z)^n$, hence the conclusion becomes

$$\frac{1}{2\pi i}\oint_{c_\varepsilon}\frac{h(w)}{(w-z)^n}dw = \frac{1}{(n-1)!}\frac{d^{n-1}}{dw^{n-1}}h(z),$$

which can be then solved by Cauchy's integral formula. □

6.3.2 The main proof

The main idea behind the proof of Theorem 6.10 is to exploit the local factorisation properties at zeros and poles, and then apply the residue theorem.

Proof of the argument principle. Suppose f has a zero of order m when $w = z$. Then $f(w) = (w-z)^m g(w)$ and $g(w) \neq 0$. Hence,

$$\frac{f'(w)}{f(w)} = \frac{m}{w-z} + \frac{g'(w)}{g(w)},$$

which leads to

$$\oint_\gamma \frac{f'(w)h(w)}{f(w)} = \oint_\gamma \frac{mh(w)}{w-z} = mh(z), \tag{6.1}$$

since

$$\lambda w.\frac{g'(w)h(w)}{g(w)}$$

is holomorphic near z (i.e., g, g' and h are holomorphic and $g(w) \neq 0$).

Similarly, if f has a pole of order m when $w = z$, then $f(w) = g(w)/(w-z)^m$ and $g(w) \neq 0$. Hence,

$$\oint_\gamma \frac{f'(w)h(w)}{f(w)} = \oint_\gamma \frac{-mh(w)}{w-z} = -mh(z). \tag{6.2}$$

By combining Equations (6.1), (6.2) and Lemma 6.8[3], we can show

$$\oint_\gamma \frac{f'(w)h(w)}{f(w)} = 2\pi i\left(\sum_{p\in\text{zeros}} n(\gamma,p)h(p)\mathrm{zo}(\mathrm{f},\mathrm{p}) - \sum_{\mathrm{p}\in\text{poles}} \mathrm{n}(\gamma,\mathrm{p})\mathrm{h}(\mathrm{p})\mathrm{po}(\mathrm{f},\mathrm{p})\right),$$

where $\mathrm{zo}(\mathrm{f},\mathrm{p})$ (or $\mathrm{po}(\mathrm{f},\mathrm{p})$) is the order of zero (or pole) of f at p, and the proof is now complete. □

[3] Either Lemma 6.8 or Theorem 6.3 suffices in this place.

6.3.3 Remarks

My definitions and lemmas in §6.3.1 roughly follow Stein and Shakarchi [81], with one major exception. When f has a pole of order n at z, Stein and Shakarchi define residue as

$$\mathrm{Res}(f,z) = \lim_{w \to z} \frac{1}{(n-1)!} \frac{d^{n-1}}{dw^{n-1}} [(w-z)^n f(w)]$$

while my Lemma 6.13 states

$$\mathrm{Res}(f,z) = \frac{1}{(n-1)!} \frac{d^{n-1}}{dw^{n-1}} h(z),$$

where $f(w) = h(w)/(w-z)^n$ and $h(w)$ is holomorphic and non-vanishing near z. Note, $h(w) = (w-z)^n f(w)$ only when $w \neq z$, since $f(w)$ is a pole (i.e. undefined) when $w = z$. Introducing the function h eliminates the limit from the definition, and the associated technical difficulties of reasoning about limits formally.

6.4 Rouché's theorem

Given two functions f and g holomorphic on an open set containing a path γ, if

$$|f(w)| > |g(w)|$$

for all $w \in \gamma$, then Rouché's theorem states that f and $f+g$ have the same number of zeros counted with multiplicity and weighted with winding number:

Theorem 6.14 (*Rouche_theorem*).

```
fixes f g::"complex ⇒ complex" and s::"complex set"
defines "fg ≡ (λp. f p + g p)"
defines "zeros_fg ≡ {p. fg p = 0}" and "zeros_f ≡ {p. f p = 0}"
assumes "open s" and "connected s" and
  "finite zeros_fg" and "finite zeros_f" and
  "f holomorphic_on s" and "g holomorphic_on s" and
  "valid_path γ" and "γ 1 = γ 0" and
  "path_image γ ⊆ s " and
  "∀z ∈ path_image γ. cmod(f z) > cmod(g z)" and
  "∀z. (z ∉ s) ⟶ winding_number γ z = 0"
shows "(∑p ∈ zeros_fg. winding_number γ p * zorder fg p)
        = (∑p ∈ zeros_f. winding_number γ p * zorder f p)"
```

Proof. Let $\mathbb{Z}(f+g)$ and $\mathbb{Z}(f)$ be the number of zeros that $f+g$ and f has respectively (counted with multiplicity and weighted with winding number). By the argument principle, we have

$$\mathbb{Z}(f+g) = \frac{1}{2\pi i}\oint_\gamma \frac{(f+g)'}{f+g} = \frac{1}{2\pi i}\oint_\gamma \frac{f'}{f} + \frac{1}{2\pi i}\oint_\gamma \frac{(1+\frac{g}{f})'}{1+\frac{g}{f}}$$

and

$$\mathbb{Z}(f) = \frac{1}{2\pi i}\oint_\gamma \frac{f'}{f.}$$

Hence, $\mathbb{Z}(f+g) = \mathbb{Z}(f)$ holds if we manage to show

$$\frac{1}{2\pi i}\oint_\gamma \frac{(1+\frac{g}{f})'}{1+\frac{g}{f}} = 0.$$

As illustrated in Fig. 6.4, let

$$h(w) = 1 + \frac{g(w)}{f(w).}$$

Then the image of $h \circ \gamma$ is located within the disc of radius 1 centred at 1, since $|f(w)| > |g(w)|$ for all w on the image of γ. In this case, it can be observed that 0 lies outside $h \circ \gamma$, which leads to

$$\oint_{h\circ\gamma} \frac{dw}{w} = n(h\circ\gamma, 0) = 0,$$

where $n(h\circ\gamma,0)$ is the winding number of $h\circ\gamma$ at 0. Hence, we have

$$\oint_\gamma \frac{(1+\frac{g}{f})'}{1+\frac{g}{f}} = \int_0^1 \frac{h'(\gamma(t))}{h(\gamma(t))}\gamma'(t)dt = \int_0^1 \frac{(h\circ\gamma)'(t)}{(h\circ\gamma)(t)}dt = \oint_{h\circ\gamma}\frac{dw}{w} = 0,$$

and this concludes the proof.

□

My proof of Theorem 6.14 follows informal textbook proofs [7, 55], but the formulation here is slightly more general: we do not require γ to be a regular closed path (i.e. where $n(\gamma,w) = 0 \vee n(\gamma,w) = 1$ for every complex number w that does not lie on the image of γ).

6.5 Related work

HOL Light has a comprehensive library of complex analysis, on top of which the prime number theorem, the Kepler conjecture and other impressive results have been formalised

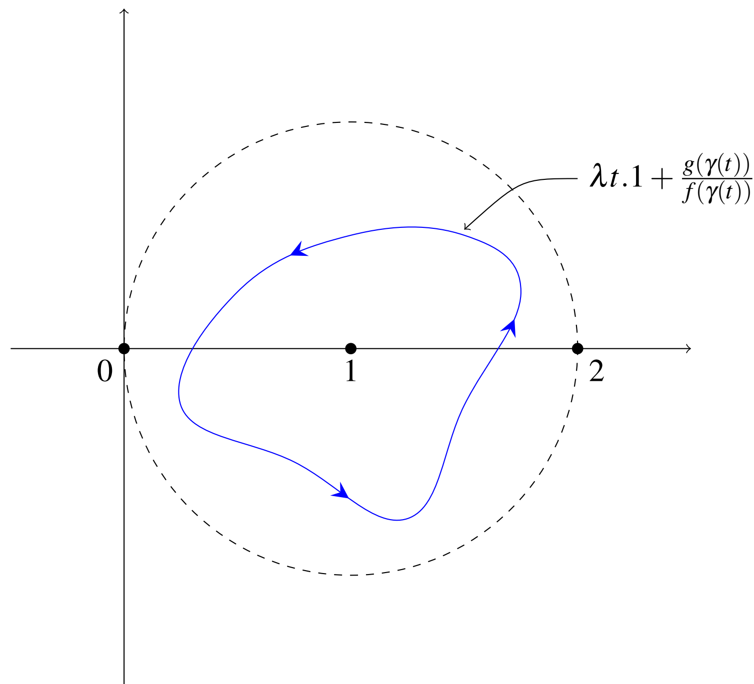

Fig. 6.4 The path image of $\lambda t.1 + \frac{g(\gamma(t))}{f(\gamma(t))}$ when $|f(w)| > |g(w)|$ for all w on the image of γ

[40, 41, 43, 45]. A substantial portion of this library has been ported to Isabelle/HOL. It should be not hard to port my results to HOL Light.

Brunel [14] has described some non-constructive complex analysis in Coq, including a formalisation of winding numbers. Also, there are other Coq libraries (mainly about real analysis), such as Coquelicot [10] and C-Corn [26]. However, as far as I know, Cauchy's integral theorem (which is the starting point of proving Cauchy's residue theorem) is not available in Coq yet.

Chapter 7

Cauchy indices on the complex plane

As mentioned in §6.2.4 of the previous chapter, I found it was hard to formally evaluate winding numbers with current theories of Isabelle. This has motivated me to develop a theory of Cauchy indices on the complex plane, through which we can systematically evaluate winding numbers. On the top of this, by combining this theory with the argument principle, we can further have effective procedures to count the number of complex roots in some domains such as a rectangle and a half-plane.

Formulations in this chapter, such as the definition of the Cauchy index and statements of some key lemmas, mainly follow Rahman and Schmeisser's book [78, Chapter 11] and Eisermann's paper [34]. Nevertheless, I was still obliged to devise some proofs on my own as discussed later in the chapter.

Chapter outline. This chapter continues as follows: I start with a motivating example (§7.1) followed by an intuitive description of the link between winding numbers and Cauchy indices (§7.2). Formal development of the previous intuition is presented in (§7.3). Next, verified procedures that count the number of complex roots in a domain are presented in (§7.4), following which some limitations are discussed (§7.5). Finally, I bring up some general remarks and potential applications (§7.6).

7.1 A motivating example

When evaluating some improper integrals using the residue theorem in Example 6.9 in the previous chapter, the most troublesome part of that proof was to evaluate the following winding numbers:

$$n(L_r + C_r, i) = 1 \tag{7.1}$$

$$n(L_r + C_r, -i) = 0 \tag{7.2}$$

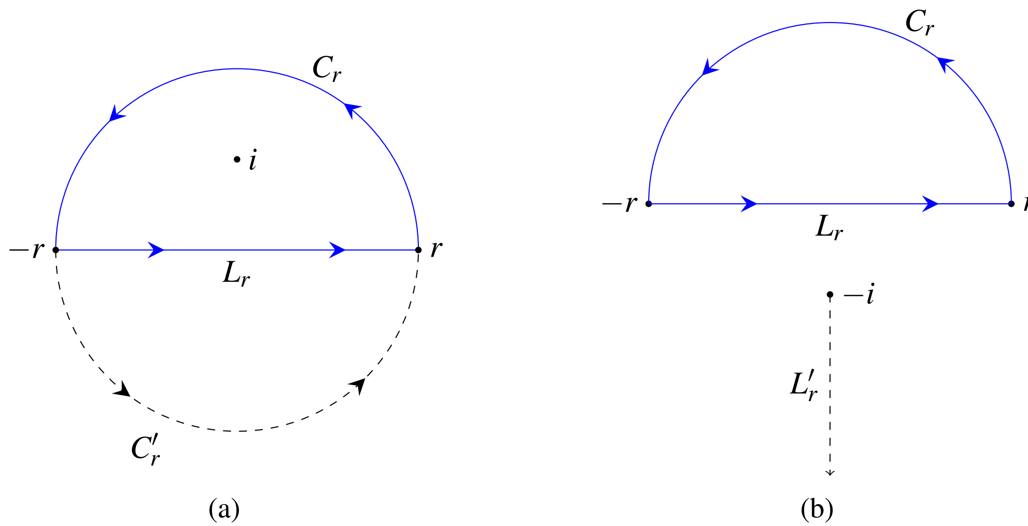

Fig. 7.1 Complex points $(0,-i)$ and $(0,i)$, and a closed path $L_r + C_r$

where C_r

$$C_r(t) = re^{i\pi t} \qquad \text{for} \quad t \in [0,1]$$

is a semicircular path centered at 0 with radius $r > 1$, and L_r

$$L_r(t) = (1-t)(-r) + tr \qquad \text{for} \quad t \in [0,1]$$

is a linear path from $-r$ to r on the complex plane. The closed path $L_r + C_r$ is formed by appending C_r to the end of L_r.

Equations (7.1) and (7.2) are straightforward to humans, as it can be seen from Fig. 7.1 that $L_r + C_r$ passes counterclockwise around the point i exactly one time, and about $-i$ zero time. However, formally deriving these facts was non-trivial.

Example 7.1 (Proof of $n(L_r + C_r, i) = 1$). I defined an auxiliary semi-circular path C'_r where

$$C'_r(t) = re^{i\pi(t+1)} \qquad \text{for} \quad t \in [0,1]$$

as can be seen in Fig. 7.1a. As $C_r + C'_r$ forms a (full) circular path with i lying inside the circle, I had

$$n(C_r + C'_r, i) = 1. \tag{7.3}$$

In addition, I further proved that $C_r + C'_r$ and $L_r + C_r$ are homotopic on the space of the complex plane except for the point i (i.e., on $\mathbb{C} - \{i\}$), and hence

$$n(L_r + C_r, i) = n(C_r + C'_r, i) \tag{7.4}$$

by using the following Isabelle lemma:

Lemma 7.2 (`winding_number_homotopic_paths`).

```
fixes z::complex and γ1 γ2::"real ⇒ complex"
assumes "homotopic_paths (-{z}) γ1 γ2"
shows "winding_number γ1 z = winding_number γ2 z"
```

where `winding_number γ1 z` encodes the winding number of γ_1 around z: $n(\gamma_1, z)$, and `homotopic_paths` encodes the homotopic proposition between two paths. Putting (7.3) and (7.4) together yields $n(L_r + C_r, i) = 1$, which concludes the whole proof.

Example 7.3 (Proof of $n(L_r + C_r, -i) = 0$). I started by defining a ray L'_r starting from $-i$ and pointing towards the negative infinity of the imaginary axis:

$$L'_r(t) = (-i) - ti \qquad \text{for} \quad t \in [0, \infty)$$

as illustrated in Fig. 7.1b. Subsequently, I showed that

$$L'_R \text{ does not intersect with } L_r + C_r, \tag{7.5}$$

and then applied the following lemma in Isabelle

Lemma 7.4 (`winding_number_less_1`).

```
fixes z w::complex and γ::"real ⇒ complex"
assumes "valid_path γ" and "z ∉ path_image γ" and "w ≠ z"
  and not_intersection:"⋀a::real. 0 < a ⟹ z + a*(w - z) ∉ path_image γ"
shows "|Re(winding_number γ z)| < 1"
```

where

- `valid_path γ` assumes that γ is piecewise continuously differentiable on $[0, 1]$,
- `z ∉ path_image γ` asserts that z is not on the path γ,
- the assumption `not_intersection` asserts that the ray starting at $z \in \mathbb{C}$ and through $w \in \mathbb{C}$ ($\{z + a(w - z) \mid a > 0\}$) does not intersect with γ—for all $a > 0$, $z + a(w - z)$ does not lie on γ.

Note that the real part of a winding number $\text{Re}(n(\gamma, z))$ measures the degree of the winding: in case of γ winding around z counterclockwise for exactly one turn, we have $n(\gamma, z) = \text{Re}(n(\gamma, z)) = 1$. Essentially, Lemma 7.4 claims that a path γ can only wind around z for

less than one turn, $|\mathrm{Re}(n(\gamma,z))| < 1$, if there is a ray starting at z and not intersecting with γ. Joining Lemma 7.4 with (7.5) leads to

$$|\mathrm{Re}(n(L_r + C_r, -i))| < 1. \tag{7.6}$$

Moreover, as $L_r + C_r$ is a closed path,

$$n(L_r + C_r, -i) \in \mathbb{Z} \tag{7.7}$$

By combining (7.6) and (7.7) , I managed to derive $n(L_r + C_r, -i) = 0$.

As can be observed in Examples 7.1 and 7.3, my proofs of $n(L_r + C_r, i) = 1$ and $n(L_r + C_r, -i) = 0$ are ad hoc, and involve manual construction of auxiliary paths/rays (e.g. C'_R and L'_R). Similar difficulty has also been mentioned by John Harrison when formalising the prime number theorem [43]. In the next section, I will introduce an idea to systematically evaluate winding numbers.

7.2 The intuition

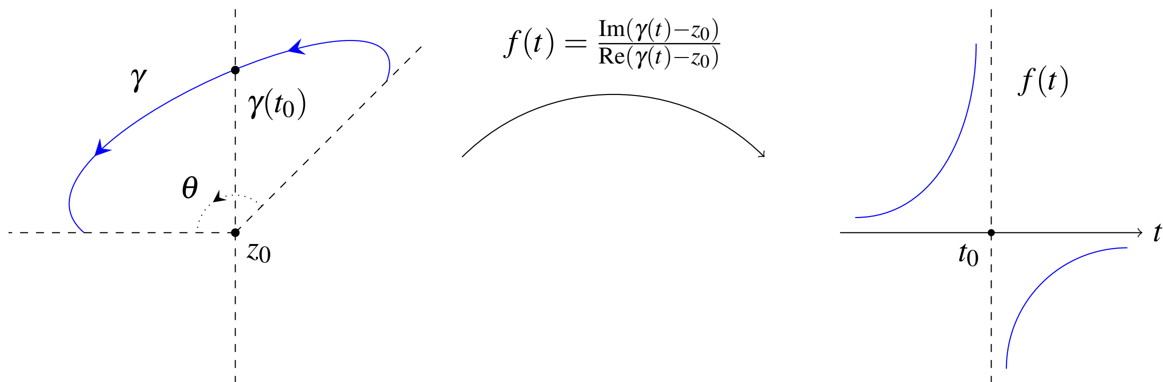

Fig. 7.2 Left: a path γ crosses the line $\{z \mid \mathrm{Re}(z) = \mathrm{Re}(z_0)\}$ at $\gamma(t_0)$ such that $\mathrm{Re}(\gamma(t_0)) > \mathrm{Re}(z_0)$. Right: the image of f as a point travels through γ

The fundamental idea of evaluating a winding number $n(\gamma, z_0)$ in this chapter is to reduce the evaluation to *classifications* of how paths cross the line $\{z \mid \mathrm{Re}(z) = \mathrm{Re}(z_0)\}$.

In a simple case, suppose a path γ crosses the line $\{z \mid \mathrm{Re}(z) = \mathrm{Re}(z_0)\}$ exactly once at the point $\gamma(t_0)$ such that $\mathrm{Im}(\gamma(t_0)) > \mathrm{Im}(z_0)$ (see Fig. 7.2 (left)), and let θ be the change in

the argument of a complex point travelling through γ. It should not be hard to observe that

$$0 < \theta < 2\pi,$$

and by considering $\mathrm{Re}(n(\gamma, z_0)) = \theta/(2\pi)$ we can have

$$0 < \mathrm{Re}(n(\gamma, z_0)) < 1,$$

which is an approximation of $\mathrm{Re}(n(\gamma, z_0))$. That is, we have approximated $\mathrm{Re}(n(\gamma, z_0))$ by the way that γ crosses the line $\{z \mid \mathrm{Re}(z) = \mathrm{Re}(z_0)\}$.

To make this idea more precise, let

$$f(t) = \frac{\mathrm{Im}(\gamma(t) - z_0)}{\mathrm{Re}(\gamma(t) - z_0)}.$$

The image of f as a point travels through γ is as illustrated in Fig. 7.2 (right), where f jumps from $+\infty$ to $-\infty$ across t_0. We can then formally characterise those jumps.

Definition 7.5 (Jump)**.** For $f : \mathbb{R} \to \mathbb{R}$ and $x \in \mathbb{R}$, we define

$$\mathrm{jump}_+(f, x) = \begin{cases} \frac{1}{2} & \text{if } \lim_{u \to x^+} f(u) = +\infty, \\ -\frac{1}{2} & \text{if } \lim_{u \to x^+} f(u) = -\infty, \\ 0 & \text{otherwise}, \end{cases}$$

$$\mathrm{jump}_-(f, x) = \begin{cases} \frac{1}{2} & \text{if } \lim_{u \to x^-} f(u) = +\infty, \\ -\frac{1}{2} & \text{if } \lim_{u \to x^-} f(u) = -\infty, \\ 0 & \text{otherwise}. \end{cases}$$

Specifically, we can conjecture that $\mathrm{jump}_+(f, t_0) - \mathrm{jump}_-(f, t_0)$ captures the way that γ crosses the line $\{z \mid \mathrm{Re}(z) = \mathrm{Re}(z_0)\}$ in Fig. 7.2, hence $\mathrm{Re}(n(\gamma, z_0))$ can be approximated using jump_+ and jump_-:

$$\left| \mathrm{Re}(n(\gamma, z_0)) + \frac{\mathrm{jump}_+(f, t_0) - \mathrm{jump}_-(f, t_0)}{2} \right| < \frac{1}{2}.$$

In more general cases, we can define Cauchy indices by summing up these jumps over an interval and along a path.

Definition 7.6 (Cauchy index)**.** For $f : \mathbb{R} \to \mathbb{R}$ and $a, b \in \mathbb{R}$, the Cauchy index of f over a closed interval $[a,b]$ is defined as

$$\mathrm{Ind}_a^b(f) = \sum_{x \in [a,b)} \mathrm{jump}_+(f,x) - \sum_{x \in (a,b]} \mathrm{jump}_-(f,x).$$

Definition 7.7 (Cauchy index along a path)**.** Given a path $\gamma : [0,1] \to \mathbb{C}$ and a point $z_0 \in \mathbb{C}$, the Cauchy index along γ about z_0 is defined as

$$\mathrm{Indp}(\gamma, z_0) = \mathrm{Ind}_0^1(f)$$

where

$$f(t) = \frac{\mathrm{Im}(\gamma(t) - z_0)}{\mathrm{Re}(\gamma(t) - z_0)}.$$

In particular, it can be checked that the Cauchy index $\mathrm{Indp}(\gamma, z_0)$ captures the way that γ crosses the line $\{z \mid \mathrm{Re}(z) = \mathrm{Re}(z_0)\}$, hence leads to an approximation of $\mathrm{Re}(n(\gamma, z_0))$:

$$\left| \mathrm{Re}(n(\gamma, z_0)) + \frac{\mathrm{Indp}(\gamma, z_0)}{2} \right| < \frac{1}{2}.$$

More interestingly, by further knowing that γ is a loop we can derive $\mathrm{Re}(n(\gamma, z_0)) = n(\gamma, z_0) \in \mathbb{Z}$ and $\mathrm{Indp}(\gamma, z_0)/2 \in \mathbb{Z}$, following which we come to the core proposition of this chapter:

Proposition 7.8. *Given a valid path $\gamma : [0,1] \to \mathbb{C}$ and a point $z_0 \in \mathbb{C}$, such that γ is a loop and z_0 is not on the image of γ, we have*

$$n(\gamma, z_0) = -\frac{\mathrm{Indp}(\gamma, z_0)}{2}.$$

That is, under some assumptions, we can evaluate a winding number through Cauchy indices!

A formal proof of Proposition 7.8 will be introduced later in the chapter (§7.3). Here, given the statement of the proposition we can have alternative proofs for $n(L_r + C_r, i) = 1$ and $n(L_r + C_r, -i) = 0$.

Example 7.9 (Alternative proof of $n(L_r + C_r, i) = 1$). As $L_r + C_r$ is a loop, applying Proposition 7.8 yields

$$n(L_r + C_r, i) = -\frac{\mathrm{Indp}(L_r + C_r, i)}{2} = -\frac{1}{2}(\mathrm{Indp}(L_r, i) + \mathrm{Indp}(C_r, i)),$$

which reduces $n(L_r + C_r, i)$ to the evaluations of $\mathrm{Indp}(L_r, i)$ and $\mathrm{Indp}(C_r, i)$. In this case, by definition we can easily decide $\mathrm{Indp}(L_r, i) = -1$ and $\mathrm{Indp}(C_r, i) = -1$ as illustrated in

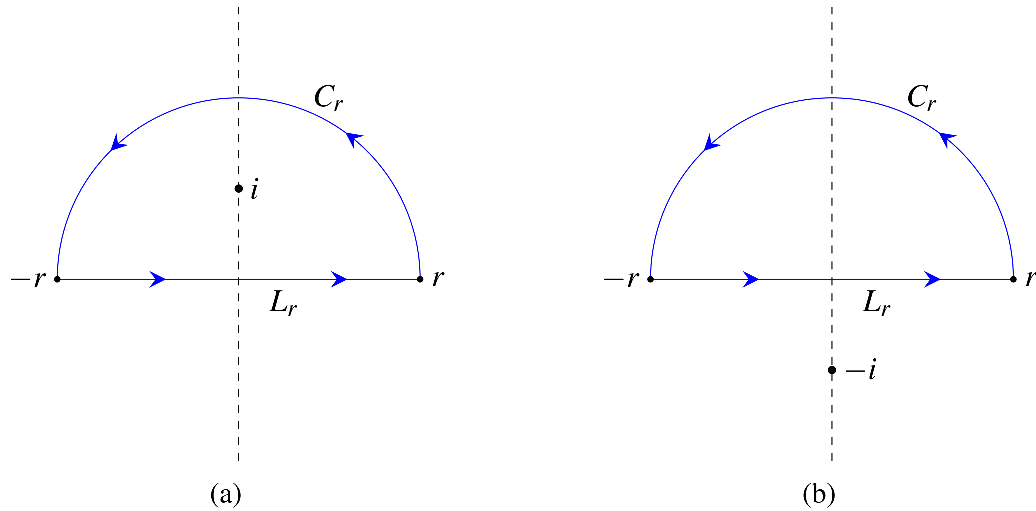

Fig. 7.3 Evaluating $n(L_r + C_r, i)$ and $n(L_r + C_r, -i)$ through the way that the path $L_r + C_r$ crosses the imaginary axis

Fig. 7.3a. Hence, we have

$$n(L_r + C_r, i) = -\frac{1}{2}((-1) + (-1)) = 1$$

and conclude the proof.

Example 7.10 (Alternative proof of $n(L_r + C_r, -i) = 0$). As shown in Fig. 7.3b, we can similarly have

$$\begin{aligned} n(L_R + C_R, -i) &= -\frac{\text{Indp}(L_r + C_r, -i)}{2} \\ &= -\frac{1}{2}(\text{Indp}(L_r, -i) + \text{Indp}(C_r, -i)) \\ &= -\frac{1}{2}(1 + (-1)) = 0 \end{aligned}$$

by which the proof is completed.

Compared to the previous proofs presented in Examples 7.1 and 7.3, the alternative proofs in Examples 7.9 and 7.10 are systematic and less demanding to devise once we have a formalisation of Proposition 7.8, which is what I will introduce in the next section.

7.3 Evaluate winding numbers

The previous section presented an informal intuition to systematically evaluate winding numbers; in this section, I will report the formal development of this intuition. We will

first present a mechanised proof of Proposition 7.8 (§7.3.1), which includes mechanised definitions of jumps and Cauchy indices (i.e., Definition 7.5, 7.6 and 7.7) and several related properties of these objects. After that, I build a tactic in Isabelle/HOL that is used to mechanise proofs presented in Example 7.9 and 7.10 (§7.3.2). Finally, I discuss some subtleties encountered during the formalisation (§7.3.3).

7.3.1 A formal proof of Proposition 7.8

For jump_- and jump_+ (see Definition 7.5), I have used the filter mechanism [48] to define a function `jumpF`:

```
definition  jumpF::"(real ⇒ real) ⇒ real filter ⇒ real" where
  "jumpF f F ≡ (if (LIM x F. f x :> at_top) then 1/2 else
                 if (LIM x F. f x :> at_bot) then -1/2 else 0)"
```

and encoded $\text{jump}_-(f,x)$ and $\text{jump}_+(f,x)$ as

`jumpF f (at_left x)` and `jumpF f (at_right x)`,

respectively. Here, `at_left x jumpF f (at_right x) at_top at_bot` are all filters, where a filter is a predicate on predicates that satisfies certain properties. Filters are extensively used in the analysis library of Isabelle to encode varieties of logical quantification: for example, `at_left x` encodes the statement "for a variable that is sufficiently close to x from the left", and `at_top` represents "for a sufficiently large variable". Furthermore, `LIM x (at_left x). f x :> at_top` encoded the proposition

$$\lim_{u \to x^-} f(u) = +\infty, \tag{7.8}$$

and this encoding can be justified by the following equality in Isabelle:

```
(LIM x (at_left x). f x :> at_top) = (∀z. ∃b<x. ∀y>b. y < x ⟶ z ≤ f y)
```

where `∀z. ∃b<x. ∀y>b. y < x ⟶ z ≤ f y` matches the usual definition of (7.8) in textbooks.

We can then encode $\text{Ind}_a^b(f)$ and $\text{Indp}(\gamma, z_0)$ (see Definitions 7.6 and 7.7) as `cindexE` and `cindex_pathE` respectively:

```
definition cindexE::"real ⇒ real ⇒ (real ⇒ real) ⇒ real" where
  "cindexE a b f =
      (∑x∈{x. jumpF f (at_right x) ≠0 ∧ a≤x ∧ x<b}. jumpF f (at_right x))
      - (∑x∈{x. jumpF f (at_left x) ≠0 ∧ a<x ∧ x≤b}. jumpF f (at_left x))"
```

```
definition cindex_pathE::"(real ⇒ complex) ⇒ complex ⇒ real" where
  "cindex_pathE γ z0 = cindexE 0 1 (λt. Im (γ t - z0) / Re (γ t - z0))"
```

Note, in the definition of $\mathrm{Ind}_a^b(f)$ we have a term

$$\sum_{x\in[a,b)} \mathrm{jump}_+(f,x)$$

which actually hides an assumption that only a finite number of points within the interval $[a,b)$ contribute to the sum. This assumption is made explicit when defining *cindexE*, where the sum is over the set

```
{x. jumpF f (at_right x) ≠ 0 ∧ a ≤ x ∧ x < b}.
```

In the case that the set above is infinite (i.e., the sum $\sum_{x\in[a,b)} \mathrm{jump}_+(f,x)$ is not mathematically well-defined) we have

```
(∑x∈{x. jumpF f (at_right x)≠0 ∧ a≤x ∧ x<b}. jumpF f (at_right x)) = 0.
```

In other words, a default value (i.e., 0) is used in Isabelle/HOL when summing over an infinite set.

Due to the issue of well-defined sums, many of our lemmas related *cindexE* will have an assumption *finite_jumpFs*:

```
definition finite_jumpFs::"(real ⇒ real) ⇒ real ⇒ real ⇒ bool" where
  "finite_jumpFs f a b = finite {x. (jumpF f (at_left x) ≠ 0
                                  ∨ jumpF f (at_right x) ≠ 0) ∧ a ≤ x ∧ x ≤ b}"
```

which guarantees the well-definedness of *cindexE*.

Now, suppose that we know that Indp is well-defined (i.e., there are only finite number of jumps over the path). What is the strategy we can employ to formally prove Proposition 7.8? Naturally, we may want to divide the path into a finite number of segments (subpaths) induced by those jumps, and then perform inductions on these segments. To formalise the finiteness of such segments, we can have:

```
inductive finite_Psegments::"(real ⇒ bool) ⇒ real ⇒ real ⇒ bool"
   for P where
  emptyI: "a≥b ⟹ finite_Psegments P a b"|
  insertI_1: "⟦s∈{a..<b}; s=a ∨ P s; ∀t∈{s<..<b}. P t;
     finite_Psegments P a s⟧ ⟹ finite_Psegments P a b"|
  insertI_2: "⟦s∈{a..<b}; s=a ∨ P s; ∀t∈{s<..<b}. ¬P t;
     finite_Psegments P a s⟧ ⟹ finite_Psegments P a b"
```

```
definition finite_ReZ_segments::"(real ⇒ complex) ⇒ complex ⇒ bool" where
  "finite_ReZ_segments γ z0 = finite_Psegments (λt. Re (γ t - z0) = 0) 0 1"
```

The idea behind `finite_ReZ_segments` is that a jump of

$$f(t) = \frac{\mathrm{Im}(\gamma(t) - z_0)}{\mathrm{Re}(\gamma(t) - z_0)}$$

takes place only if $\lambda t.\ \mathrm{Re}(\gamma(t) - z_0)$ changes from 0 to $\neq 0$ (or vice versa). Hence, each of the segments of the path γ separated by those jumps has either $\lambda t.\ \mathrm{Re}(\gamma(t) - z_0) = 0$ or $\lambda t.\ \mathrm{Re}(\gamma(t) - z_0) \neq 0$.

As can be expected, the finiteness of jumps over a path can be derived by the finiteness of segments:

Lemma 7.11 (`finite_ReZ_segments_imp_jumpFs`).

```
fixes γ::"real ⇒ complex" and z0::complex
assumes "finite_ReZ_segments γ z0" and "path γ"
shows "finite_jumpFs (λt. Im (γ t - z0)/Re (γ t - z0)) 0 1"
```

where `path γ` asserts that γ is a continuous function on $[0..1]$ (so that it is a path). Roughly speaking, Lemma 7.11 claims that a path will have a finite number of jumps if it can be divided into a finite number of segments.

By assuming such a finite number of segments we have well-defined `cindex_pathE`, and can then derive some useful related properties:

Lemma 7.12 (`cindex_pathE_subpath_combine`).

```
fixes γ::"real ⇒ complex" and z0::complex
assumes "finite_ReZ_segments γ z0"and "path γ"
  and "0≤a" and "a≤b" and "b≤c" and "c≤1"
shows "cindex_pathE (subpath a b γ) z0 + cindex_pathE (subpath b c γ) z0
        = cindex_pathE (subpath a c γ) z0"
```

where `subpath a b γ` gives a sub-path of γ based on parameters a and b:

```
definition subpath :: "real ⇒ real ⇒ (real ⇒ 'a) ⇒ real
    ⇒ 'a::real_normed_vector"
  where "subpath a b γ ≡ (λt. γ((b - a) * t + a))"
```

Essentially, Lemma 7.12 indicates that we can combine Cauchy indices along consecutive parts of a path: given a path γ and three parameters a, b, c with $0 \leq a \leq b \leq c \leq 1$, we have

$$\mathrm{Indp}(\gamma_1, z_0) + \mathrm{Indp}(\gamma_2, z_0) = \mathrm{Indp}(\gamma_3, z_0).$$

where $\gamma_1 = \lambda t.\, \gamma((b-a)t+a)$, $\gamma_2 = \lambda t.\, \gamma((c-b)t+b)$ and $\gamma_3 = \lambda t.\, \gamma((c-a)t+a)$.

More importantly, we now have an induction rule for a path with a finite number of segments:

Lemma 7.13 (*finite_ReZ_segments_induct*).

```
fixes γ::"real ⇒ complex" and z₀::complex
  and P::"(real ⇒ complex) ⇒ complex ⇒ bool"
assumes "finite_ReZ_segments γ z₀"
  and sub0:"⋀g z. (P (subpath 0 0 g) z)"
  and subEq:"(⋀s g z. ⟦s ∈ {0..<1}; s=0 ∨ Re (g s) = Re z;
        ∀t ∈ {s<..<1}. Re (g t) = Re z;
        finite_ReZ_segments (subpath 0 s g) z;
        P (subpath 0 s g) z⟧ ⟹ P g z)"
  and subNEq:"(⋀s g z. ⟦s ∈ {0..<1}; s=0 ∨ Re (g s) = Re z;
        ∀t ∈ {s<..<1}. Re (g t) ≠ Re z;
        finite_ReZ_segments (subpath 0 s g) z;
        P (subpath 0 s g) z⟧ ⟹ P g z)"
shows "P γ z₀"
```

where `P` is a predicate that takes a path γ and a complex point z_0, and

- *sub0* is the base case that `P` holds for a constant path;
- *subEq* is the inductive case when the last segment is right on the line $\{x \mid \mathrm{Re}(x) = \mathrm{Re}(z)\}$: $\forall t \in (s,1).\, \mathrm{Re}(g(t)) = \mathrm{Re}(z)$;
- *subNEq* is the inductive case when the last segment does not cross the line $\{x \mid \mathrm{Re}(x) = \mathrm{Re}(z)\}$: $\forall t \in (s,1).\, \mathrm{Re}(g(t)) \neq \mathrm{Re}(z)$.

Given a path γ with a finite number of segments, a complex point z_0 and a predicate P that takes a path and a complex number and returns a boolean, Lemma 7.13 provides us with an inductive rule to derive $P(\gamma, z_0)$ by recursively examining the last segment.

Before attacking Proposition 7.8, we can show an auxiliary lemma about $\mathrm{Re}(n(\gamma, z_0))$ and $\mathrm{Indp}(\gamma, z_0)$ when the end points of γ are on the line $\{z \mid \mathrm{Re}(z) = \mathrm{Re}(z_0)\}$:

Lemma 7.14 (*winding_number_cindex_pathE_aux*).

```
fixes γ::"real ⇒ complex" and z₀ :: complex
assumes "finite_ReZ_segments γ z₀" and "valid_path γ"
  and "z₀ ∉ path_image γ" and "Re (γ 1) = Re z₀"
  and "Re (γ 0) = Re z₀"
shows "2 * Re(winding_number γ z₀) = - cindex_pathE γ z₀"
```

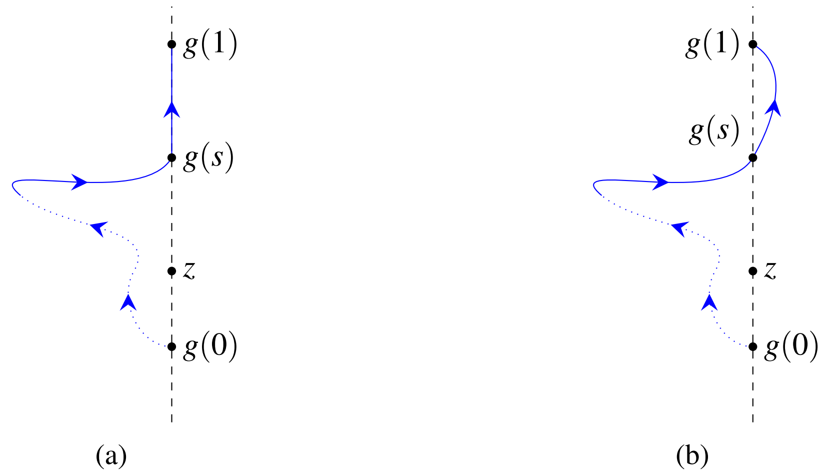

Fig. 7.4 Inductive cases when applying Lemma 7.13

Here, Lemma 7.14 is almost equivalent to Proposition 7.8 except for that more restrictions haven been placed on the end points of γ.

Proof of Lemma 7.14. As there is a finite number of segments along γ (i.e., `finite_ReZ_segments γ z0`), by inducting on these segments with Lemma 7.13 we end up with three cases. The base case is straightforward: given a constant path $g : [0,1] \to \mathbb{C}$ and a complex point $z \in \mathbb{C}$, we have $\text{Re}(n(g,z)) = 0$ and $\text{Indp}(g,z) = 0$, hence $2\,\text{Re}(n(g,z)) = -\text{Indp}(g,z)$.

For the inductive case when the last segment is right on the line $\{x \mid \text{Re}(x) = \text{Re}(z)\}$, there is $\forall t \in (s,1).\ \text{Re}(g(t)) = \text{Re}(z)$ as illustrated in Fig. 7.4a. Let

$$g_1(t) = g(st)$$

$$g_2(t) = g((1-s)t).$$

We have

$$n(g,z) = n(g_1,z) + n(g_2,z), \tag{7.9}$$

and, by the induction hypothesis,

$$2\,\text{Re}(n(g_1,z)) = -\text{Indp}(g_1,z). \tag{7.10}$$

Moreover, it is possible to derive

$$2\,\text{Re}(n(g_2,z)) = -\text{Indp}(g_2,z), \tag{7.11}$$

since $n(g_2,z) = 0$ and $\mathrm{Indp}(g_2,z) = 0$. Furthermore, by Lemma 7.12 we can sum up the Cauchy index along g_1 and g_2:

$$\mathrm{Indp}(g_1,z) + \mathrm{Indp}(g_2,z) = \mathrm{Indp}(g,z) \tag{7.12}$$

Combining Equations (7.9), (7.10), (7.11) and (7.12) yields

$$\begin{aligned} 2\,\mathrm{Re}(n(g,z)) &= 2(\mathrm{Re}(n(g_1,z)) + \mathrm{Re}(n(g_2,z))) \\ &= -\,\mathrm{Indp}(g_1,z) - \mathrm{Indp}(g_2,z) \\ &= -\,\mathrm{Indp}(g,z) \end{aligned} \tag{7.13}$$

which concludes the case.

For the other inductive case when the last segment does not cross the line $\{x \mid \mathrm{Re}(x) = \mathrm{Re}(z)\}$, without loss of generality, we assume

$$\forall t \in (s,1).\ \mathrm{Re}(g(t)) > \mathrm{Re}(z), \tag{7.14}$$

and the shape of g is as illustrated in Fig. 7.4b. Similar to the previous case, by letting $g_1(t) = g(st)$ and $g_2(t) = g((1-s)t)$, we have $n(g,z) = n(g_1,z) + n(g_2,z)$ and, by the induction hypothesis, $2\,\mathrm{Re}(n(g_1,z)) = -\,\mathrm{Indp}(g_1,z)$. Moreover, by observing the shape of g_2 we have

$$2\,\mathrm{Re}(n(g_2,z)) = \mathrm{jump}_-(f,1) - \mathrm{jump}_+(f,0) \tag{7.15}$$

$$\mathrm{Indp}(g_2,z) = \mathrm{jump}_+(f,0) - \mathrm{jump}_-(f,1) \tag{7.16}$$

where $f(t) = \mathrm{Im}(g_2(t) - z)/\mathrm{Re}(g_2(t) - z)$. Combining (7.15) with (7.16) leads to $2\,\mathrm{Re}(n(g_2,z)) = -\,\mathrm{Indp}(g_2,z)$, following which we finish the case by deriving $2\,\mathrm{Re}(n(g,z)) = -\,\mathrm{Indp}(g,z)$ in a way analogous to (7.13). □

Finally, we are ready to formally derive Proposition 7.8 in Isabelle/HOL:

Theorem 7.15 (*winding_number_cindex_pathE*).

```
fixes γ::"real ⇒ complex" and z₀::complex
assumes "finite_ReZ_segments γ z₀" and "valid_path γ"
    and "z₀ ∉ path_image γ" and "γ 0 = γ 1"
shows "winding_number γ z₀ = - cindex_pathE γ z₀ / 2"
```

Proof. By assumption, we know that γ is a loop, and the point $\gamma(0) = \gamma(1)$ can be away from the line $\{z \mid \mathrm{Re}(z) = \mathrm{Re}(z_0)\}$ which makes Lemma 7.14 inapplicable. To resolve this problem, we look for a point $\gamma(s)$ on γ such that $0 \le s \le 1$ and $\mathrm{Re}(\gamma(s)) = \mathrm{Re}(z_0)$, and we can either fail or succeed.

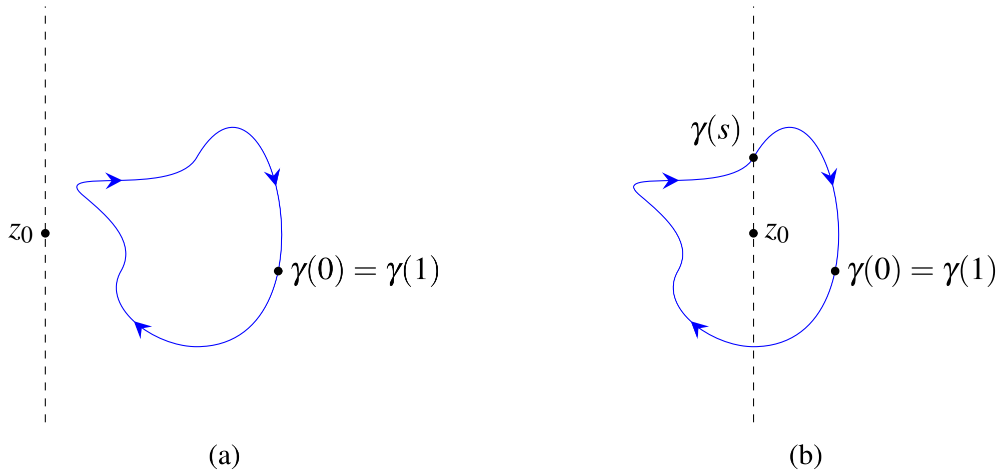

Fig. 7.5 To derive $n(\gamma, z_0) = -\frac{\mathrm{Indp}(\gamma, z_0)}{2}$ when γ is a loop

In the case of failure, without loss of generality, we can assume $\mathrm{Re}(\gamma(t)) > \mathrm{Re}(z_0)$ for all $0 \le t \le 1$, and the shape of γ is as illustrated in Fig. 7.5a. As the path γ does not cross the line $\{z \mid \mathrm{Re}(z) = \mathrm{Re}(z_0)\}$, we can evaluate

$$\mathrm{Indp}(\gamma, z_0) = 0$$

$$n(\gamma, z_0) = \mathrm{Re}(n(\gamma, z_0)) = \frac{\mathrm{Im}(\mathrm{Ln}(\gamma(1) - z_0)) - \mathrm{Im}(\mathrm{Ln}(\gamma(0) - z_0))}{2\pi} = 0$$

where Ln is a complex logarithm function. Hence, $n(\gamma, z_0) = -\mathrm{Indp}(\gamma, z_0)/2$ which concludes the case.

In the case of success, as illustrated in Fig. 7.5b, we have $\mathrm{Re}(\gamma(s)) = \mathrm{Re}(z_0)$. We then define a shifted path γ_s:

$$\gamma_s(t) = \begin{cases} \gamma(t+s) & \text{if } s + t \le 1, \\ \gamma(t+s-1) & \text{otherwise,} \end{cases}$$

such that $\mathrm{Re}(\gamma_s(0)) = \mathrm{Re}(\gamma_s(1)) = \mathrm{Re}(z_0)$. By applying Lemma 7.14, we obtain a relationship between $\mathrm{Re}(n(\gamma_s, z_0))$ and $\mathrm{Indp}(\gamma_s, z_0)$:

$$2\,\mathrm{Re}(n(\gamma_s, z_0)) = -\mathrm{Indp}(\gamma_s, z_0),$$

following which we have $n(\gamma, z_0) = -\mathrm{Indp}(\gamma, z_0)/2$, since $n(\gamma_s, z_0) = n(\gamma, z_0)$ and $\mathrm{Indp}(\gamma_s, z_0) = \mathrm{Indp}(\gamma, z_0)$. $\square$

7.3.2 A tactic for evaluating winding numbers

With Proposition 7.8 formalised, we are now able to build a tactic to evaluate winding numbers through Cauchy indices. The idea has already been sketched in Examples 7.9 and 7.10. In general, I have built a tactic named *eval_winding* to convert goals of the form

$$n(\gamma_1 + \gamma_2 + \cdots + \gamma_n, z_0) = k, \tag{7.17}$$

where k is an integer and γ_j ($1 \leq j \leq n$) is either a linear path:

$$\gamma_j(t) = (1-t)a + tb \quad \text{where } a, b \in \mathbb{C}$$

or a part of a circular path:

$$\gamma_j(t) = z + re^{i((1-t)a+tb)} \quad \text{where } a, b, r \in \mathbb{R} \text{ and } z \in \mathbb{C}.$$

The tactic *eval_winding* will transform (7.17) into

$$\gamma_j(1) = \gamma_{j+1}(0) \text{ for all } 1 \leq j \leq n-1, \text{ and } \gamma_n(1) = \gamma_1(0), \tag{7.18}$$

$$z_0 \notin \{\gamma_j(t) \mid 0 \leq t \leq 1\} \text{ for all } 1 \leq j \leq n, \tag{7.19}$$

$$\text{Indp}(\gamma_1, z_0) + \text{Indp}(\gamma_2, z_0) + \cdots + \text{Indp}(\gamma_n, z_0) = -2k, \tag{7.20}$$

where (7.18) ensures that the path $\gamma_1 + \gamma_2 + \cdots + \gamma_n$ is a loop; (7.19) certifies that z_0 is not on the image of $\gamma_1 + \gamma_2 + \cdots + \gamma_n$.

To achieve this transformation, *eval_winding* will first perform a substitution step on the left-hand side of Equation (7.17) using Theorem 7.15. As the substitution is conditional, we will need to resolve four extra subgoals (i.e., (7.21), (7.22), (7.23) and (7.24) as follows) and Equation (7.17) is transformed into (7.25):

$$\textit{finite_ReZ_segments } (\gamma_1 \text{ +++ } \gamma_2 \text{ +++ } \ldots \text{ +++}\gamma_n)\ z_0, \tag{7.21}$$

$$\textit{valid_path } (\gamma_1 \text{ +++ } \gamma_2 \text{ +++ } \ldots \text{ +++}\gamma_n), \tag{7.22}$$

$$z_0 \notin \textit{path_image } (\gamma_1 \text{ +++ } \gamma_2 \text{ +++ } \ldots \text{ +++}\gamma_n), \tag{7.23}$$

$$(\gamma_1 \text{ +++ } \gamma_2 \text{ +++ } \ldots \text{ +++}\gamma_n)\ 0 = (\gamma_1 \text{ +++ } \gamma_2 \text{ +++ } \ldots \text{ +++}\gamma_n)\ 1, \tag{7.24}$$

$$-\ \textit{cindex_pathE } (\gamma_1 \text{ +++ } \gamma_2 \text{ +++ } \ldots \text{ +++}\gamma_n)\ z_0\ /\ 2 = k. \tag{7.25}$$

To simplify (7.21), the tactic will keep applying the following introduction rule:[1]

Lemma 7.16 (*finite_ReZ_segments_joinpaths*).
fixes γ_1 γ_2 *:: "real ⇒ complex"* **and** z_0 *:: complex*
assumes *"finite_ReZ_segments* γ_1 z_0*"* **and** *"finite_ReZ_segments* γ_2 z_0*"*
and *"path* γ_1*"* **and** *"path* γ_2*"* **and** *"*γ_1 *1 =* γ_2 *0"*
shows *"finite_ReZ_segments (*γ_1*+++*γ_2*)* z_0*"*

to eliminate the path join operations (*+++*) until the predicate *finite_ReZ_segments* is only applied to a linear path or a part of a circular path, and either of these two cases can be directly discharged because these two kinds of paths are proved to be divisible into a finite number of segments by the imaginary axis:

Lemma 7.17 (*finite_ReZ_segments_linepath*).
"finite_ReZ_segments (linepath a b) z"

Lemma 7.18 (*finite_ReZ_segments_part_circlepath*).
"finite_ReZ_segments (part_circlepath z0 r st tt) z"

In terms of other subgoals introduced when applying Lemma 7.16, such as *path* γ_1, *path* γ_2 and γ_1 *1 =* γ_2 *0*, we can discharge them by the following introduction and simplification rules (all of which have been formally proved):

- ⟦*path* γ_1*; path* γ_2*;* γ_1 *1 =* γ_2 *0*⟧ $\Longrightarrow$ *path(*γ_1 *+++* γ_2*)*,
- *path (part_circlepath* z_0 *r st tt)*,
- *path (linepath a b)*,
- *(*γ_1 *+++* γ_2*) 1 =* γ_2 *1*,
- *(*γ_1 *+++* γ_2*) 0 =* γ_1 *0*.

As a result, *eval_winding* will eventually simplify the subgoal (7.21) to (7.18).

Similar to the process of simplifying (7.21) to (7.18), the tactic *eval_winding* will also simplify

- (7.22) to (7.18),
- (7.23) to (7.19),
- and (7.24) to (7.18).

[1] Applying an introduction rule will replace a goal by a set of subgoals derived from the premises of the rule, provided the goal can be unified with the conclusion of the rule.

Finally, with respect to (7.25), we can similarly rewrite with a rule between the Cauchy index (`cindex_pathE`) and the path join operation (`+++`):

Lemma 7.19 (`cindex_pathE_joinpaths`)**.**

```
fixes γ1 γ2 :: "real ⇒ complex" and z0 :: complex
assumes "finite_ReZ_segments γ1 z0" and "finite_ReZ_segments γ2 z0"
  and "path γ1" and "path γ2" and "γ1 1 = γ2 0"
shows "cindex_pathE (γ1 +++ γ2) z0 = cindex_pathE γ1 z0 + cindex_pathE γ2 z0"
```

to convert the subgoal (7.25) to (7.18) and (7.20).

After building the tactic `eval_winding`, we are now able to convert a goal like Equation (7.17) to (7.18), (7.19) and (7.20). In most cases, discharging (7.18) and (7.19) is straightforward. To derive (7.20), we will need to formally evaluate each $\mathrm{Indp}(\gamma_j, z_0)$ $(1 \leq j \leq n)$ when γ_j is either a linear path or a part of a circular path.

When γ_j is a linear path, the following lemma grants us a way to evaluate $\mathrm{Indp}(\gamma_j, z_0)$ through its right-hand side:

Lemma 7.20 (`cindex_pathE_linepath`)**.**

```
fixes a b z0 :: complex
assumes "z0∉path_image (linepath a b)"
shows "cindex_pathE (linepath a b) z0 = (
  let c1 = Re a - Re z0;
      c2 = Re b - Re z0;
      c3 = Im a * Re b + Re z0 * Im b + Im z0 * Re a - Im z0 * Re b
            - Im b * Re a - Re z0 * Im a;
      d1 = Im a - Im z0;
      d2 = Im b - Im z0
  in if (c1>0 ∧ c2<0) ∨ (c1<0 ∧ c2>0) then
        (if c3>0 then 1 else -1)
     else
        (if (c1=0 ⟷ c2≠0) ∧ (c1=0 ⟶d1≠0) ∧ (c2=0 ⟶ d2≠0) then
            if (c1=0 ∧ (c2 >0 ⟷ d1>0)) ∨ (c2=0 ∧ (c1 >0 ⟷ d2<0))
            then 1/2 else -1/2
        else 0))"
```

Although Lemma 7.20 may appear terrifying, evaluating its right-hand side is usually automatic when the number of free variables is small. For example, in a formal proof of Example 7.9 in Isabelle/HOL, we can have the following fragment:

lemma

```
  fixes R::real
  assumes "R>1"
  shows "winding_number (part_circlepath 0 R 0 pi +++ linepath (-R) R) i = 1"
proof (winding_eval, simp_all)
  ...
  have "i ∉ path_image (linepath (- R) (R::complex))" by ...
  from cindex_pathE_linepath[OF this] ‹R>1›
  have "cindex_pathE (linepath (-R) (R::complex)) i = -1" by  auto
  ...
qed
```

where `winding_eval` is first applied to convert the goal into (7.18), (7.19) and (7.20), and `simp_all` subsequently simplifies those newly generated subgoals. In the middle of the proof, we show that the complex point i is not on the image of the linear path L_r (i.e., `linepath (-R) (R::complex))` in Isabelle/HOL), following which we apply Lemma 7.20 to derive $\text{Indp}(L_r, i) = -1$: the evaluation process is automatic through the command `auto`, given the assumption `R>1`.

When γ_j is a part of a circular path, a similar lemma has been provided to facilitate the evaluation of $\text{Indp}(\gamma_j, z_0)$.

7.3.3 Subtleties

The first subtlety I have encountered during the formalisation of Proposition 7.8 is about the definition of jumps and Cauchy indices, for which my first attempt followed the standard definitions in textbooks [67, 78, 8].

Definition 7.21 (Jump)**.** For $f : \mathbb{R} \to \mathbb{R}$ and $x \in \mathbb{R}$, we define

$$\text{jump}(f, x) = \begin{cases} 1 & \text{if } \lim_{u \to x^-} f(u) = -\infty \text{ and } \lim_{u \to x^+} f(u) = +\infty, \\ -1 & \text{if } \lim_{u \to x^-} f(u) = +\infty \text{ and } \lim_{u \to x^+} f(u) = -\infty, \\ 0 & \text{otherwise.} \end{cases}$$

Definition 7.22 (Cauchy index)**.** For $f : \mathbb{R} \to \mathbb{R}$ and $a, b \in \mathbb{R}$, the Cauchy index of f over an open interval (a, b) is defined as

$$\text{Ind}_a^b(f) = \sum_{x \in (a,b)} \text{jump}(f, x).$$

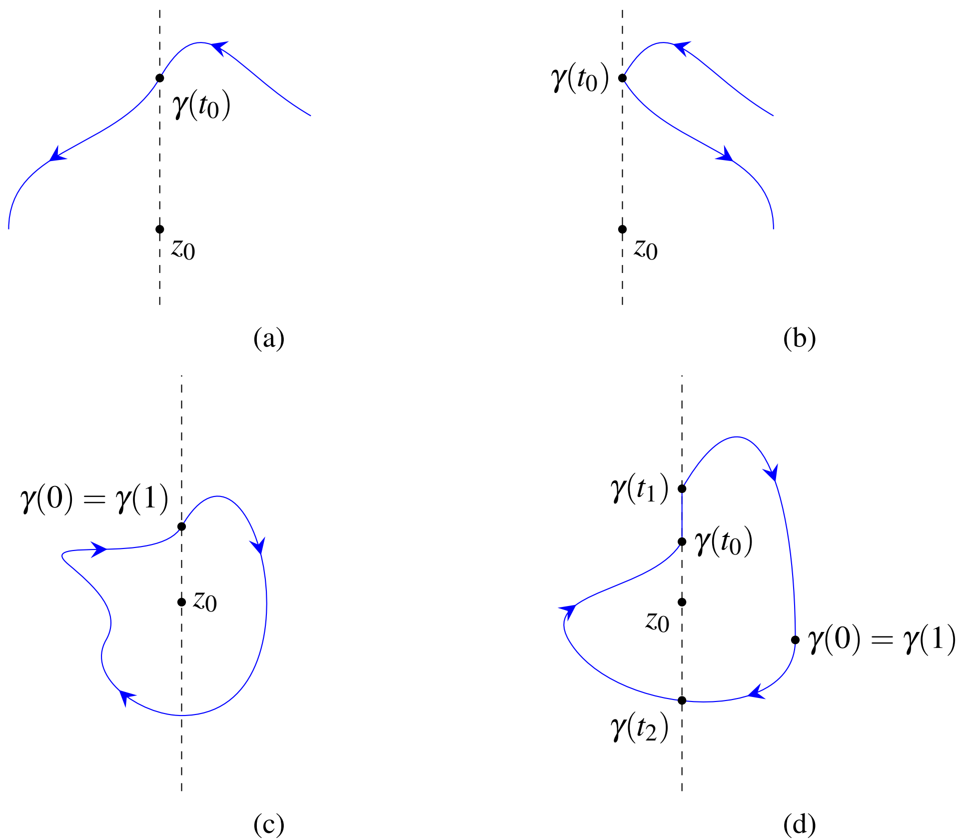

Fig. 7.6 Different ways a path γ can intersect with the line $\{z \mid \mathrm{Re}(z) = \mathrm{Re}(z_0)\}$

The impact of the difference between the current definition of the Cauchy index (i.e., Definition 7.6) and the classic one (i.e., Definition 7.22) is small when formalising the Sturm-Tarski theorem, where f is a rational function. In this case, the path γ intersects with the line $\{z \mid \mathrm{Re}(z) = \mathrm{Re}(z_0)\}$ a finite number of times, and for each intersection point (see Fig. 7.6a and b), by letting $f(t) = \mathrm{Im}(\gamma(t) - z_0)/\mathrm{Re}(\gamma(t) - z_0)$, we have

$$\mathrm{jump}(f,t) = \mathrm{jump}_+(f,t) - \mathrm{jump}_-(f,t),$$

hence

$$\sum_{x \in (a,b)} \mathrm{jump}(f,x) = \sum_{x \in [a,b)} \mathrm{jump}_+(f,x) - \sum_{x \in (a,b]} \mathrm{jump}_-(f,x),$$

provided $\mathrm{jump}_+(f,a) = 0$ and $\mathrm{jump}_-(f,b) = 0$. That is, the classic Cauchy index and the current one are equal when f is a rational function and does not jump at both ends of the target interval.

Naturally, the disadvantages of Definition 7.22 are twofold:

- The function $\lambda t.\, \mathrm{Re}(\gamma(0) - z_0)$ cannot vanish at either end of the interval. That is, we need to additionally assume $\mathrm{Re}(\gamma(0) - z_0) \neq 0$ as in Rahman and Schmeisser's formulation [78, Lemma 11.1.1 and Theorem 11.1.3], and Proposition 7.8 will be inapplicable in the case of Fig. 7.6c where $\mathrm{Re}(\gamma(0)) = \mathrm{Re}(\gamma(1)) = \mathrm{Re}(z_0)$.

- The function $\lambda t. \mathrm{Im}(\gamma(t) - z_0)/\mathrm{Re}(\gamma(t) - z_0)$ has to be rational, which makes Proposition 7.8 inapplicable for cases like in Fig. 7.6d (if we follow Definition 7.22). To elaborate, it can be observed in Fig. 7.6d that $n(\gamma, z_0) = -1$, while we will only get a wrong answer by following Definition 7.22 and evaluating through Proposition 7.8:

$$-\frac{1}{2}\left(\sum_{x\in(0,1)} \mathrm{jump}(f,x)\right) = -\frac{\mathrm{jump}(f,t_2)}{2} = -\frac{1}{2},$$

  where $f(t) = \mathrm{Im}(\gamma(t) - z_0)/\mathrm{Re}(\gamma(t) - z_0)$. In comparison, Definition 7.6 leads to the correct answer:

$$\begin{aligned} n(\gamma, z_0) &= -\frac{1}{2}\left(\sum_{x\in[0,1)} \mathrm{jump}_+(f,x) - \sum_{x\in(0,1]} \mathrm{jump}_-(f,x)\right) \\ &= -\frac{1}{2}\left(\mathrm{jump}_+(f,t_2) + \mathrm{jump}_+(f,t_1) - \mathrm{jump}_-(f,t_2) - \mathrm{jump}_-(f,t_0)\right) \\ &= -\frac{1}{2}\left(\frac{1}{2} + \frac{1}{2} - (-\frac{1}{2}) - (-\frac{1}{2})\right) \\ &= -1. \end{aligned}$$

Fortunately, Michael Eisermann [34] recently proposed a new formulation of the Cauchy index that overcomes those two disadvantages, and this new formulation is what I have followed (in Definitions 7.5 and 7.6).

Another subtlety I ran into was the well-definedness of the Cauchy index. Such well-definedness is usually not an issue and left implicit in the literature, because, in most cases, the Cauchy index is only defined on rational functions, where only finitely many points can contribute to the sum. When attempting to formally derive Proposition 7.8, I realised that this assumption needed to be made explicit, since the path γ can be flexible enough to allow the function $f(t) = \mathrm{Im}(\gamma(t) - z_0)/\mathrm{Re}(\gamma(t) - z_0)$ to be non-rational (e.g. Fig. 7.6d). In my first attempt of following Definition 7.22, the Cauchy index was formally defined as follows:

```
definition cindex::"real ⇒ real ⇒ (real ⇒ real) ⇒ int" where
  "cindex a b f = (∑x∈{x. jump f x≠0 ∧ a<x ∧ x<b}. jump f x)"
```

and its well-definedness was ensured by the finite number of times that γ crosses the line $\{z \mid \mathrm{Re}(z) = \mathrm{Re}(z_0)\}$:

```
definition finite_axes_cross::"(real ⇒ complex) ⇒ complex ⇒ bool" where
  "finite_axes_cross γ z0 =
     finite {t. (Re (γ t - z0) = 0 ∨ Im (γ t - z0) = 0) ∧ 0 ≤ t ∧ t ≤ 1}"
```

where the part `Re (γ t - z0) = 0` ensures that `jump f t` is non-zero only at a finite number of points over the interval $[0,1]$. When constrained by `finite_axes_cross`, the function $f(t) = \mathrm{Im}(\gamma(t) - z_0)/\mathrm{Re}(\gamma(t) - z_0)$ behaves like a rational function. More importantly, the path γ, in this case, can be divided into a finite number of ordered segments delimited by those points over $[0,1]$, which makes an inductive proof of Proposition 7.8 possible. However, after abandoning my first attempt and switching to Definition 7.6, the well-definedness of the Cauchy index is assured by the finite number of jump_+ and jump_- of f (i.e., Definition `finite_jumpFs` in §7.3.1), with which I did not know how to divide the path γ into segments and carry out an inductive proof. It took me quite some time to properly define the assumption of finite segments (i.e., Definition `finite_ReZ_segments`) that implied the well-definedness through Lemma 7.11 and provided a lemma for inductive proofs (i.e., Lemma 7.13).

7.4 Counting the number of complex roots

In the previous section, I have described a way to evaluate winding numbers through Cauchy indices. In this section, I will further explore this idea and propose verified procedures to count the number of complex roots of a polynomial in some domain, such as a rectangle and a half-plane.

Does a winding number have anything to do with the number of roots of a polynomial? The answer is yes. Thanks to the argument principle in the previous chapter, we can calculate the number of roots by evaluating a contour integral:

$$\frac{1}{2\pi i} \oint_\gamma \frac{p'(x)}{p(x)} dx = N \tag{7.26}$$

where $p \in \mathbb{C}[x]$, $p'(x)$ is the first derivative of p and N is the number of complex roots of p (counted with multiplicity) inside the loop γ. Also, by the definition of winding numbers, we have

$$n(p \circ \gamma, 0) = \frac{1}{2\pi i} \oint_\gamma \frac{p'(x)}{p(x)} dx. \tag{7.27}$$

Combining Equations (7.26) and (7.27) gives us the relationship between a winding number and the number of roots of a polynomial:

$$n(p \circ \gamma, 0) = N. \tag{7.28}$$

And the question becomes: can we evaluate $n(p \circ \gamma, 0)$ through Cauchy indices?

7.4.1 Roots in a rectangle

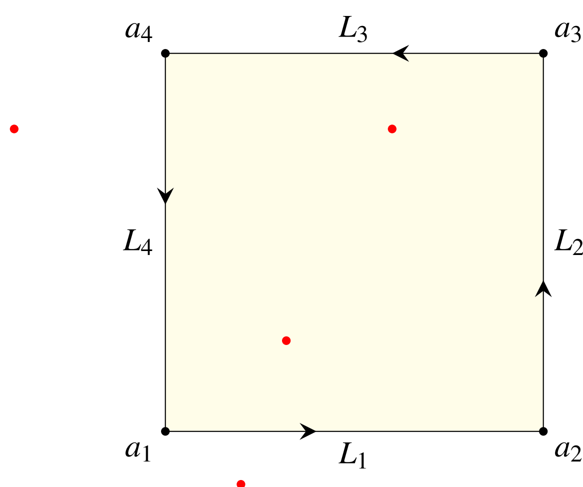

Fig. 7.7 Complex roots of a polynomial (red dots) and a rectangular path $(L_1 + L_2 + L_3 + L_4)$ on the complex plane

Let N be the number of complex roots of a polynomial p inside the rectangle defined by its lower left corner a_1 and upper right corner a_3. As illustrated in Fig. 7.7, we can define four linear paths along the edge of the rectangle:

$$\begin{aligned}
L_1(t) &= (1-t)a_1 + ta_2 \\
L_2(t) &= (1-t)a_2 + ta_3 \\
L_3(t) &= (1-t)a_3 + ta_4 \\
L_4(t) &= (1-t)a_4 + ta_1
\end{aligned}$$

where $a_2 = \mathrm{Re}(a_3) + i\,\mathrm{Im}(a_1)$ and $a_4 = \mathrm{Re}(a_1) + i\,\mathrm{Im}(a_3)$. Combining Proposition 7.8 with Equation (7.28) yields

$$\begin{aligned} N &= n(p \circ (L_1 + L_2 + L_3 + L_4), 0) \\ &= -\frac{1}{2}\,\mathrm{Indp}(p \circ (L_1 + L_2 + L_3 + L_4), 0) \\ &= -\frac{1}{2}\left(\mathrm{Indp}(p \circ L_1, 0) + \mathrm{Indp}(p \circ L_2, 0) + \mathrm{Indp}(p \circ L_3, 0) + \mathrm{Indp}(p \circ L_4, 0)\right). \end{aligned} \tag{7.29}$$

Here, the path $p \circ L_j : [0,1] \to \mathbb{C}$ $(1 \leq j \leq 4)$ is (mostly) neither a linear path nor a part of a circular path, which indicates our evaluation strategies in §7.3.2, such as Lemma 7.20, will no longer apply. Thankfully, the Sturm-Tarski theorem, which I have already formalised in Chapter 3, come to our rescue.

As a side product of the Sturm-Tarski theorem, we can evaluate the Cauchy index of a rational function f through some remainder sequence: let $q, p \in \mathbb{R}[x]$ be, respectively, the numerator and denominator polynomial of f, such that $f(t) = q(t)/p(t)$. We have

$$\mathrm{Ind}_a^b(f) = \mathrm{Var}(\mathrm{SRemS}(p,q); a, b) \tag{7.30}$$

where $a, b \in \overline{\mathbb{R}}$ $a < b$ and are not roots of p. $\mathrm{SRemS}(p,q)$ and $\mathrm{Var}(-; a, b)$ are, respectively, the signed remainder sequence and the difference in the number of sign variations as stated in Theorem 3.2 in Chapter 3.

Back to the case of $\mathrm{Indp}(p \circ L_j, 0)$, we have

$$\mathrm{Indp}(p \circ L_j, 0) = \mathrm{Ind}_0^1\left(\lambda t.\ \frac{\mathrm{Im}(p(L_j(t)))}{\mathrm{Re}(p(L_j(t)))}\right)$$

and the function $\lambda t.\,\mathrm{Im}(p(L_j(t)))/\mathrm{Re}(p(L_j(t)))$ is, amazingly, a rational function! Therefore, by combining Equations (7.29) and (7.30) we have an idea of how to count the number of roots inside a rectangle.

While proceeding to the formal development, the first problem I encountered is that the Cauchy index in Equation (7.30) actually follows the classic definition (i.e., Definition 7.22), and is different from the one in Equation (7.29) (i.e., Definitions 7.6 and 7.7). Subtle differences between these two formulations have already been discussed in §7.3.3. Luckily, Eisermann [34] has also described an alternative sign variation operator so that our current definition of the Cauchy index (i.e., Definition 7.6) can be computationally evaluated:

Lemma 7.23 (*cindex_polyE_changes_alt_itv_mods*).

```
fixes a b::real and p q::"real poly"
assumes "a < b" and "coprime p q"
```

```
shows "cindex_polyE a b q p = changes_alt_itv_smods a b p q / 2"
```

Here, `cindex_polyE` is the Cauchy index of a function f when f is known to be rational (i.e., $f(t) = q(t)/p(t)$):

Lemma 7.24 (`cindexE_eq_cindex_polyE`).

```
fixes a b::real and p q::"real poly"
assumes "a < b"
shows "cindexE a b (λx. poly q x / poly p x) = cindex_polyE a b q p"
```

where the alternative sign variation operator $\widehat{\mathrm{Var}}$ is defined as follows:

$$
\begin{aligned}
\widehat{\mathrm{Var}}([p_1, p_2, ..., p_3]; a, b) &= \widehat{\mathrm{Var}}([p_1, p_2, ..., p_3]; a) - \widehat{\mathrm{Var}}([p_1, p_2, ..., p_3]; b), \\
\widehat{\mathrm{Var}}([p_1, p_2, ..., p_3]; a) &= \widehat{\mathrm{Var}}([p_1(a), p_2(a), ..., p_3(a)]), \\
\widehat{\mathrm{Var}}([]) &= 0, \\
\widehat{\mathrm{Var}}([x_1]) &= 0, \\
\widehat{\mathrm{Var}}([x_1, x_2, ..., x_n]) &= |\mathrm{sgn}(x_1) - \mathrm{sgn}(x_2)| + \widehat{\mathrm{Var}}([x_2, ..., x_n]).
\end{aligned}
$$

The difference between $\widehat{\mathrm{Var}}$ and Var is that Var discards zeros before calculating variations while $\widehat{\mathrm{Var}}$ takes zeros into consideration. For example, $\mathrm{Var}([1, 0, -2]) = \mathrm{Var}([1, -2]) = 1$, while $\widehat{\mathrm{Var}}([1, 0, -2]) = 2$.

Before implementing Equation (7.29), we need to realise that there is a restriction in our strategy: roots are not allowed on the border (i.e., the image of the path $L_1 + L_2 + L_3 + L_4$). To computationally check this restriction, the following function is defined

```
definition no_proots_line::"complex poly ⇒ complex ⇒ complex ⇒ bool" where
  "no_proots_line p a b = (proots_within p (closed_segment a b) = {})"
```

which will return "true" if there is no root on the closed segment between `a` and `b`, and "false" otherwise. Here, `closed_segment a b` is defined as the set $\{(1-u)a + ub \mid 0 \leq u \leq 1\} \subseteq \mathbb{C}$, and the function `proots_within p s` gives the set of roots of the polynomial `p` within the set `s`:

```
definition proots_within::"'a::comm_semiring_0 poly ⇒ 'a set ⇒ 'a set" where
  "proots_within p s = {x∈s. poly p x=0}"
```

To make `no_proots_line` executable, we can derive the following code equation:

Lemma 7.25 (`no_proots_line_code[code]`).

```
"no_proots_line p a b = (if poly p a ≠ 0 ∧ poly p b ≠ 0 then
                          (let p_c = p ∘_p [:a, b - a:];
```

```
            pR = map_poly Re pc;
            pI = map_poly Im pc;
            g  = gcd pR pI
          in if changes_itv_smods 0 1 g (pderiv g) = 0
            then True else False)
        else False)"
```

where $\circ_p$ is the polynomial composition operation and `map_poly Re` and `map_poly Im`, respectively, extract the real and imaginary parts of the complex polynomial p_c.

Proof of Lemma 7.25. Supposing $L : [0,1] \to \mathbb{C}$ is a linear path from a to b: $L(t) = (1-t)a + tb$, we know that $p \circ L$ is still a polynomial with complex coefficients. Subsequently, we extract the real and imaginary part (p_R and p_I, respectively) of $p \circ L$ such that

$$p(L(t)) = p_R(t) + ip_I(t).$$

If there is a root of p lying right on L, we will be able to obtain some $t_0 \in [0,1]$ such that

$$p_R(t_0) = p_I(t_0) = 0,$$

hence, by letting $g = \gcd(p_R, p_I)$ we have $g(t_0) = 0$. Therefore, the polynomial p has no (complex) root on L if and only if g has no (real) root within the interval $[0,1]$, and the latter can be computationally checked using Sturm's theorem. □

Finally, I define the function `proots_rectangle` that returns the number of complex roots of a polynomial (counted with multiplicity) within a rectangle defined by its lower left and upper right corner:

```
definition proots_rectangle::"complex poly ⇒ complex ⇒ complex ⇒ int" where
  "proots_rectangle p a1 a3 = proots_count p (box a1 a3)"
```

where `proots_count` is defined as follows:

```
definition proots_count::"'a::idom poly ⇒ 'a set ⇒ nat" where
  "proots_count p s = (∑r∈proots_within p s. order r p)"
```

As usual, the executability of the function `proots_rectangle` can be established: The executability of the function `proots_rectangle` can be established with the following code equation:

Lemma 7.26 (`proots_rectangle_code1[code]`).

```
  "proots_rectangle p a1 a3 =
```

```
(if Re a₁ < Re a₃ ∧ Im a₁ < Im a₃ then
    if p≠0 then
    if no_proots_line p a₁ (Complex (Re a₃) (Im a₁))
      ∧ no_proots_line p (Complex (Re a₃) (Im a₁)) a₃
      ∧ no_proots_line p a₃ (Complex (Re a₁) (Im a₃))
      ∧ no_proots_line p (Complex (Re a₁) (Im a₃)) a₁ then
    (
    let p₁ = p ∘ₚ [:a₁, Complex (Re a₃ - Re a₁) 0:];
        pR1 = map_poly Re p₁; pI1 = map_poly Im p₁; g₁ = gcd pR1 pI1;
        p₂ = p ∘ₚ [:Complex (Re a₃) (Im a₁), Complex 0 (Im a₃ - Im a₁):];
        pR2 = map_poly Re p₂; pI2 = map_poly Im p₂; g₂ = gcd pR2 pI2;
        p₃ = p ∘ₚ [:a₃, Complex (Re a₁ - Re a₃) 0:];
        pR3 = map_poly Re p₃; pI3 = map_poly Im p₃; g₃ = gcd pR3 pI3;
        p₄ = p ∘ₚ [:Complex (Re a₁) (Im a₃), Complex 0 (Im a₁ - Im a₃):];
        pR4 = map_poly Re p₄; pI4 = map_poly Im p₄; g₄ = gcd pR4 pI4
    in
      - (changes_alt_itv_smods 0 1 (pR1 div g₁) (pI1 div g₁)
        + changes_alt_itv_smods 0 1 (pR2 div g₂) (pI2 div g₂)
        + changes_alt_itv_smods 0 1 (pR3 div g₃) (pI3 div g₃)
        + changes_alt_itv_smods 0 1 (pR4 div g₄) (pI4 div g₄)) div 4
    )
    else Code.abort (STR ''proots_rectangle fails when there is
            a root on the border.'') (λ_. proots_rectangle p a₁ a₃)
    else Code.abort (STR ''proots_rectangle fails when p=0.'')
            (λ_. proots_rectangle p a₁ a₃)
    else 0
)"
```

The proof of the above code equation roughly follows Equations (7.29) and (7.30), where `no_proots_line` checks if there is a root of p on the rectangle's border. Note that the gcd calculations here, such as g_1 = *gcd* p_{R1} p_{I1}, are due to the coprime assumption in Lemma 7.23.

Example 7.27. Given a rectangle defined by $(-1, 2+2i)$ (as illustrated in Fig. 7.8) and a polynomial p with complex coefficients:

$$p(x) = x^2 - 2ix - 1 = (x-i)^2$$

we can now type the following command to count the number of roots within the rectangle:

value "proots_rectangle [:-1, -2*i, 1:] (-i) (2+2*i)"

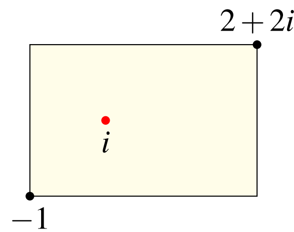

Fig. 7.8 A complex point i and a rectangle defined by its lower left corner -1 and upper right corner $2+2i$

which will return 2 as p has exactly two complex roots (i.e. i with multiplicity 2) in the area.

7.4.2 Roots in a half-plane

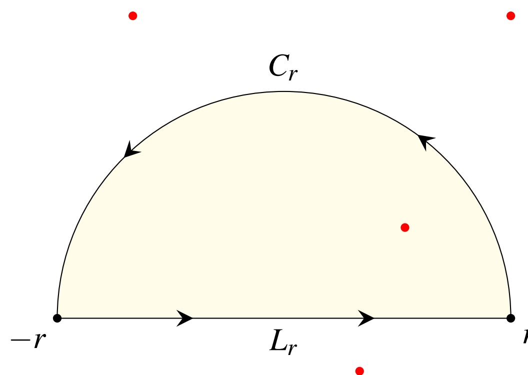

Fig. 7.9 Complex roots of a polynomial (red dots) and a linear path (L_r) concatenated by a semi-circular path (C_r) on the complex plane

For roots in a half-plane, we can start with a simplified case, where we count the number of roots of a polynomial in the upper half-plane of $\mathbb{C}$:

definition *proots_upper::"complex poly ⇒ int"* **where**
"proots_upper p = proots_count p {z. Im z>0}"

As usual, our next step is to set up the executability of *proots_upper*. To achieve that, we first define a linear path $L_r(t) = (1-t)(-r)+tr$ and a semicircular path $C_r(t) = re^{i\pi t}$, as

illustrated in Fig. 7.9. Subsequently, let

$$C_p(r) = p \circ C_r$$
$$L_p(r) = p \circ L_r,$$

and by following Equation (7.28) we have

$$\begin{aligned} N_r &= n(p \circ (L_r + C_r), 0) \\ &= \mathrm{Re}(n(L_p(r), 0)) + \mathrm{Re}(n(C_p(r), 0)) \end{aligned} \tag{7.31}$$

where N_r is the number of roots of p inside the path $L_r + C_r$. Note that as r approaches positive infinity, N_r will be the roots on the upper half-plane (i.e., `proots_upper p`), which is what we are aiming for. For this reason, it is natural for us to examine

$$\lim_{r \to +\infty} \mathrm{Re}(n(L_p(r), 0)) = ?$$

$$\lim_{r \to +\infty} \mathrm{Re}(n(C_p(r), 0)) = ?.$$

As for the case of $\lim_{r \to +\infty} \mathrm{Re}(n(L_p(r), 0))$, we can have

Lemma 7.28 (*Re_winding_number_poly_linepth*).

```
fixes p::"complex poly"
defines "Lp ≡ (λr::real. poly p o linepath (-r) r)"
assumes "lead_coeff p=1" and "∀x∈{x. poly p x=0}. Im x≠0"
shows "((λr. 2*Re (winding_number (Lp r) 0) + cindex_pathE (Lp r) 0)
        ⟶ 0) at_top"
```

which essentially indicates

$$\lim_{r \to +\infty} \mathrm{Re}(n(L_p(r), 0)) = -\frac{1}{2} \lim_{r \to +\infty} \mathrm{Indp}(L_p(r), 0), \tag{7.32}$$

provided that the polynomial p is monic and does not have any root on the real axis.

Next, for $\lim_{r \to +\infty} \mathrm{Re}(n(C_p(r), 0))$, we first derive a lemma about C_r:

Lemma 7.29 (*Re_winding_number_tendsto_part_circlepath*).

```
fixes z z0::complex
shows "((λr. Re (winding_number (part_circlepath z r 0 pi ) z0))
        ⟶ 1/2) at_top"
```

that is, $\lim_{r \to +\infty} \mathrm{Re}(n(C_r, 0)) = 1/2$, following which and by induction we have

Lemma 7.30 (*Re_winding_number_poly_part_circlepath*).

```
fixes z::complex and p::"complex poly"
defines "Cp ≡ (λr::real. poly p o part_circlepath z r 0 pi)"
assumes "degree p>0"
shows "((λr. Re (winding_number (Cp r) 0)) ⟶ degree p/2) at_top"
```

which is equivalent to

$$\lim_{r \to +\infty} \mathrm{Re}(n(C_p(r), 0)) = \frac{\deg(p)}{2}, \tag{7.33}$$

provided $\deg(p) > 0$.

Putting Equations (7.32) and (7.33) together yields the core lemma about `proots_upper` in this section:

Lemma 7.31 (*proots_upper_cindex_eq*).

```
fixes p::"complex poly"
assumes "lead_coeff p=1" and "∀x∈{x. poly p x=0}. Im x≠0"
shows "proots_upper p =
        (degree p - cindex_poly_ubd (map_poly Im p) (map_poly Re p))/2"
```

where `cindex_poly_ubd (map_poly Im p) (map_poly Re p)` is mathematically interpreted as $\mathrm{Ind}_{-\infty}^{+\infty}(\lambda t.\, \mathrm{Im}(p(t))/\mathrm{Re}(p(t)))$, which is derived from $\lim_{r\to+\infty} \mathrm{Indp}(L_p(r), 0)$ in Equation (7.32) since

$$\begin{aligned}
\lim_{r \to +\infty} \mathrm{Indp}(L_p(r), 0) &= \lim_{r \to +\infty} \mathrm{Indp}(L_p(r), 0) \\
&= \lim_{r \to +\infty} \mathrm{Ind}_0^1 \left(\lambda t.\, \frac{\mathrm{Im}(L_p(r,t))}{\mathrm{Re}(L_p(r,t))} \right) \\
&= \lim_{r \to +\infty} \mathrm{Ind}_{-r}^{r} \left(\lambda t.\, \frac{\mathrm{Im}(p(t))}{\mathrm{Re}(p(t))} \right) \\
&= \mathrm{Ind}_{-\infty}^{+\infty} \left(\lambda t.\, \frac{\mathrm{Im}(p(t))}{\mathrm{Re}(p(t))} \right).
\end{aligned}$$

Finally, following Lemma 7.31, the executability of the function `proots_upper` is established:

Lemma 7.32 (*proots_upper_code1[code]*).

```
"proots_upper p =
    (if p ≠ 0 then
      (let pm= smult (inverse (lead_coeff p)) p;
           pI= map_poly Im pm;
           pR= map_poly Re pm;
```

```
          g = gcd p_I p_R
      in
        if changes_R_smods g (pderiv g) = 0
        then
          (degree p - changes_R_smods p_R p_I) div 2
        else
          Code.abort (STR ''proots_upper fails when there is a root
            on the border.'') (λ_. proots_upper p)
    )
  else
    Code.abort (STR ''proots_upper fails when p=0.'')
      (λ_. proots_upper p))"
```

where

- `smult (inverse (lead_coeff p)) p` divides the polynomial `p` by its leading coefficient so that the resulting polynomial p_m is monic. This corresponds to the assumption `lead_coeff p=1` in Lemma 7.31.
- `changes_R_smods g (pderiv g) = 0` checks if `p` has no root lying on the real axis, which is due to the second assumption in Lemma 7.31.
- `changes_R_smods` p_R p_I evaluates

$$\mathrm{Ind}_{-\infty}^{+\infty}\left(\lambda t.\,\frac{\mathrm{Im}(p_I(t))}{\mathrm{Re}(p_R(t))}\right)$$

  by following Equation (7.30).

As for the general case of a half-plane, we can have a definition as follows:

definition `proots_half::"complex poly ⇒ complex ⇒ complex ⇒ int"` **where**
`"proots_half p a b = proots_count p {w. Im ((w - a) / (b - a)) > 0}"`

which encodes the number of roots in the left half-plane of the vector $b-a$. Roots of `p` in this half-plane can be transformed to roots of `p` $\circ_p$ `[:a, b-a:]` in the upper half-plane of $\mathbb{C}$:

Lemma 7.33 (`proots_half_proots_upper`).
fixes `a b::complex` **and** `p::"complex poly"`
assumes `"a≠b"` **and** `"p≠0"`
shows `"proots_half p a b = proots_upper (p` $\circ_p$ `[:a, b-a:])"`

And so we can naturally evaluate `proots_half` through `proots_upper`:

Lemma 7.34 (*proots_half_code1[code]*).

```
"proots_half p a b =
    (if a≠b then
     if p≠0 then
        proots_upper (p ∘p [:a, b - a:])
     else Code.abort (STR ''proots_half fails when p=0.'')
            (λ_. proots_half p a b)
     else 0)"
```

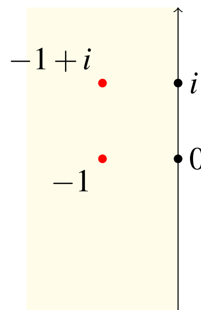

Fig. 7.10 Complex roots of a polynomial (red dots) and a vector $(0,i)$

Example 7.35. We can now use the following command

```
value "proots_half [:1-i, 2-i, 1:] 0 i"
```

to decide that the polynomial

$$p(x) = x^2 + (2-i)x + (1-i) = (x+1)(x+1-i)$$

has exactly two roots within the left half-plane of the vector $(0,i)$, as shown in Fig. 7.10.

Despite our naive implementation, both *proofs_half* and *proots_rectangle* are applicable for small or medium examples. For most polynomials with coefficient bitsize up to 10 and degree up to 30, our complex root counting procedures terminate within minutes.

7.5 Limitations and future work

There are, of course, several improvements that can be made on both the evaluation tactic in §7.3.2 and root counting procedures in §7.4. As the tactic is intended to be applied to winding numbers with variables, full automation with this tactic is unlikely in most cases, but we can always aim for better automation and an enhanced interactive experience for users (e.g., presenting unsolved goals in a more user-friendly way).

Regarding the two root-counting procedures in §7.4, a key limitation is that they do not allow cases where any of the roots is on the border. There are two possible solutions to this problem:

- To generalise the definition of winding numbers. The current formulation of winding numbers in Isabelle/HOL follows the one in complex analysis:

$$n(\gamma,z) = \frac{1}{2\pi i}\oint_\gamma \frac{dw}{w-z}$$

  which becomes undefined when the point z is on the image of the path γ. With other and more relaxed formulations of winding numbers, such as the algebraic version by Eisermann [34], we may be able to derive a more general version of the argument principle that allows zeros on the border.

- To deploy a more sophisticated strategy to count the number of times that the path winds. Recall that the underlying idea in this chapter is to reduce the evaluation of winding numbers to *classifications* of how paths cross some line. The Cauchy index merely provides one classification strategy, which I considered simple and elegant enough for formalisation. In contrast, Collins and Krandick proposed a much more sophisticated strategy for such classifications [24]. Their strategy has, in fact, been widely implemented in modern systems, such as Mathematica and SymPy, to count the number of complex roots.

Neither of these two solutions are straightforward to incorporate, hence I leave them for future investigation.

Besides rectangles and half-planes, it is also possible to similarly count the number of roots in an open disk and even a sector:

$$\text{sector}(z_0,\alpha,\beta) = \{z \mid \alpha < \arg(z-z_0) < \beta\}$$

where $\arg(-)$ returns the argument of a complex number. Informal proofs of root counting in these two domains can be found in Rahman and Schmeisser's book [78, Chapter 11].

7.6 Remarks and potential applications

Rahman and Schmeisser's book [78, Chapter 11] and Eisermann's paper [34] are the two main sources that my development is built upon. Nevertheless, there are still some differences in formulations:

- Rahman and Schmeisser formulated the Cauchy index as in Definitions 7.21 and 7.22, and their formulation was used in my first attempt. However, after I realised the subtleties discussed in §7.3.3, I abandoned this formulation and switched to the one proposed by Eisermann (i.e., Definition 7.6). As a result, the root counting procedures presented in this chapter are more general than the ones in their book due to fewer assumptions.

- Eisermann formulated a winding number $n(\gamma, z_0)$ in a real-algebraical sense where γ is required to be a piecewise polynomial path (i.e., each piece from the path needs to be a polynomial). In comparison, $n(\gamma, z_0)$ in Isabelle/HOL follows the classic definition in complex analysis, and places fewer restrictions on the shape of γ (i.e., piecewise continuously differentiable is less restrictive than being a piecewise polynomial) but does not permit z_0 to be on the image of γ (while Eisermann's formulation does). Consequently, Eisermann's root counting procedure works in more restrictive domains (i.e., he only described the rectangle case in his paper) but does not prevent roots on the border.

Another point that may be worth mentioning is the difference between informal and formal proofs: in this development, I generally treated their lemma statements as guidance and had to devise my own proofs for those statements. For instance, when proving Proposition 7.8, I defined an inductive data type for segments and derived an induction rule for it, which was far away from the informal proof scripts. Such situation also happened when I justified the root counting procedure in a half-plane.

Interestingly, the root-counting procedure in a half-plane is also related to the stability problems in the theory of dynamic systems. For instance, let $A \in \mathbb{R}^{n \times n}$ be a square matrix with real coefficients and $y : [0, +\infty) \to \mathbb{R}^n$ be a function that models the system state over time, a linear dynamic system can be described as an ordinary differential equation:

$$\frac{dy(t)}{dt} = Ay(t) \tag{7.34}$$

with an initial condition $y(0) = y_0$. The system of (7.34) is considered stable if all roots of the characteristic polynomial of A lie within the open left half-plane (i.e., $\{z \mid \mathrm{Re}(z) < 0\}$), and such stability test is usually referred as the Routh–Hurwitz stability criterion [5, Section 23; 67, Chapter 9]. As has already been demonstrated in Example 7.35, counting the number of roots in the left half-plane is within reach of the procedure `proots_half`. For this reason, I believe that the development in this chapter will be beneficial to the future reasoning of dynamic systems in Isabelle/HOL.

It is worth mentioning that root counting in a rectangle is usually coupled with a classic problem in computer algebra, namely, complex root isolation. The basic idea is to keep bisecting a rectangle (vertically or horizontally) into smaller ones until sub-rectangle contains exactly one root or none (provided the target polynomial is square-free). Following this idea it is possible to build a simple and verified complex root isolation procedure similar to Wilf's work [84]: we start with a large rectangle and then repeatedly apply the verified procedure to count roots during the rectangle bisection phase. However, compared to modern complex procedures [24, 85], this simplistic approach suffers from several drawbacks:

- My root counting procedure is based on remainder sequences, which are generally considered much slower than those built upon Descartes' rule of signs.
- Modern isolation procedures are routinely required to deliver isolation boxes whose size are up to some user-specified limit, hence they usually keep *refining* the isolation boxes even after the roots have been successfully isolated. The bisection strategy still works in the root refinement stage, but dedicated numerical approaches such as Newton's iteration are commonly implemented for efficiency reasons.
- Modern isolation procedures sometimes prefer a bit-stream model in which coefficients of the polynomial are approximated as a bit stream. This approach is particularly beneficial when the coefficients are of utterly large bit-width or consist of algebraic numbers.
- Modern implementations usually have numerous accumulated low-level optimisations, such as highly-tuned data structures, which are almost impossible to incorporate into verified procedures in a theorem prover.

Therefore, it is unlikely that my root counting procedures can lead to a verified root isolation program with extremely high efficiency. Nevertheless, they can alternatively serve as internal verified procedures to certify results from untrusted external root isolation programs, like what I did with the previous real-root-counting procedure in §5.2 of Chapter 5.

Chapter 8

Towards certifying multivariate CAD

Due to time limitation, I was not able to certify a full (multivariate) CAD procedure. On the other hand, I did manage to formalise the bivariate case of the projection theorem (i.e., Theorem 2.6 in Chapter 2), which I believe will be crucial for certifying multivariate CAD.

Chapter outline. This chapter begins with a theorem that claims continuous dependence of polynomial roots on the coefficients (§8.1), following which a bivariate case of the projection theorem in CAD has been formally derived (§8.2). Finally, I will briefly discuss my current plan towards certifying multivariate CAD (§8.3).

8.1 Polynomial roots continuously depend on coefficients

Before proceeding to the projection theorem for CAD, a key prerequisite theorem we need is the continuous dependence of polynomial roots on the coefficients.

Theorem 8.1. *Let $p \in \mathbb{C}[x]$ be a non-zero polynomial of degree n:*

$$p(z) = a_n z^n + \cdots + a_1 z + a_0,$$

and suppose p has m ($m \leq n$) distinct roots $z_1, z_2, \ldots, z_m$ with multiplicity $\mu_1, \mu_2, \ldots, \mu_m$ respectively. For any $\varepsilon > 0$ such that ε is small enough to allow the closed balls $\mathrm{cball}(z_j, \varepsilon)$ *($1 \leq j \leq m$) to be disjoint, there exists $\delta > 0$ such that for any non-zero polynomial $q \in \mathbb{C}[x]$ also of degree n:*

$$q(z) = b_n z^n + \cdots + b_1 z + b_0,$$

if $|a_i - b_i| < \delta$ for each $0 \leq i \leq n$, then given any $1 \leq j \leq m$, q has exactly μ_j roots within the open ball $\mathrm{ball}(z_j, \varepsilon)$.

Roughly speaking, given a function $f : \mathbb{C}^{n+1} \to \mathbb{C}$ that maps the coefficients of a polynomial p (of degree n) to one of its roots, Theorems 8.1 claims that f is continuous.

My formulation of Theorem 8.1 is as follows:

Theorem 8.2 (*continuous_dependence*).

```
fixes p::"complex poly" and ε::real
defines "n ≡ degree p"
assumes "ε > 0" and "p ≠ 0"
  and "∀z ∈ proots p. proots_within p (cball z ε) = {z}"
shows "∃δ > 0. ∀q. degree q = n ∧ q ≠ 0 ∧
          (∀i ≤ n. cmod(coeff q i - coeff p i) < δ)
            ⟶ (∀z ∈ proots p. proots_count q (ball z ε) = order z p)"
```

where `coeff p i` is the coefficient of the term x^i of `p` (i.e., a_i in Theorems 8.1), `proots p` is the set of (distinct) roots of `p`, and `proots_count q (ball z ε)` is the number of roots (counting multiplicity) of `q` within the open ball `ball z ε`.

Theorems 8.1 is a classic result in algebra, and has many proofs including the ones based on the implicit function theorem [8, Theorem 5.12] and those based on Rouché's theorem [67, Theorem 1.4]. Taking into account the status of Isabelle's library, I decided to follow an elegantly formulated proof by Alen Alexanderian [3], which utilised Rouché's theorem:[1]

Proof of Theorem 8.1. Let $C(-)$ be a circle of radius ε:

$$C(z) = \{w \mid |w - z| = \varepsilon\}$$

and $V(-)$ and $U(-)$ be two auxiliary functions on a circle:

$$V(z) = \inf_{w \in C(z)} |p(w)|$$

$$U(z) = \sup_{w \in C(z)} \left(\sum_{i=0}^{n} |w|^i \right)$$

where inf and sup are, respectively, the infimum and supremum operations. Since $\lambda w. |p(w)|$ is continuous on a compact set $C(z)$, we know $\{|p(w)| \mid w \in C(z)\} = (\lambda w. |p(w)|)(C(z))$ is a compact set, hence

$$V(z) \in \{|p(w)| \mid w \in C(z)\}. \tag{8.1}$$

Moreover, we have

$$\forall x \in \{|p(w)| \mid w \in C(z)\}. x > 0 \tag{8.2}$$

[1]This was actually what motivated me to mechanise Rouché's theorem in the first place.

as those circles (i.e., $C(z)$) are disjoint and centred at roots of p. Putting (8.1) and (8.2) together yields

$$V(z) > 0, \tag{8.3}$$

provided z is at a root of p. As for $U(z)$, by definition we have

$$U(z) > 0, \tag{8.4}$$

$$\forall x \in \left\{ \sum_{i=0}^{n} |w|^i \mid w \in C(z) \right\}. x \le U(z). \tag{8.5}$$

Then, by combining (8.3) with (8.4) we can obtain $\delta > 0$ such that

$$\forall z \in \mathrm{Zer}(p).\, \delta V(z) \le U(z) \tag{8.6}$$

where $\mathrm{Zer}(p)$ is the set of roots of p.

Now, choose a polynomial $q(z) = b_n z^n + \cdots + b_1 z + b_0$, such that $\deg(q) = n$ and for each $1 \le i \le n$, $|a_i - b_i| < \delta$. Next, for $z \in \mathrm{Zer}(p)$ and $w \in C(z)$, combining (8.5), (8.6), and the definition of $V(-)$ yields

$$\begin{aligned} |q(w) - p(w)| &= |\sum_{i=0}^{n} (b_i - a_i) w^i| \\ &\le \sum_{i=0}^{n} |b_i - a_i||w^i| \\ &< \delta \sum_{i=0}^{n} |w^i| \le \delta U(z) \le V(z) \le |p(w)|. \end{aligned}$$

That is, $|q(w) - p(w)| < |p(w)|$, with which we can invoke Rouché's theorem (in §6.4) to derive that p and q have the same number of roots (counting multiplicity) in the interior of $C(z)$, hence the whole proof can be concluded. □

8.2 Formal development towards the projection theorem of CAD

As mentioned in §2.2 of Chapter 2, Theorem 2.6 is the rationale behind the projection operation in Collins' CAD algorithm: it leads to a stack that is adapted to the target set of polynomials P by relating the invariance of the number of the real roots to that of the complex

roots (of P). Therefore, I believe a formal proof of this theorem should be an ingredient of certifying multivariate CAD.

The following is my formalised bivariate version of Theorem 2.6:

Lemma 8.3 (*bivariate_CAD_proj*).

```
fixes p q::"real bpoly" and S::"real set"
defines "p_y≡λb. map_poly complex_of_real (poly_y p b)"
defines "q_y≡λb. map_poly complex_of_real (poly_y q b)"
assumes "connected S"
  and deg_p_inv:      "(λb. degree (poly_y p b)) constant_on S"
  and pzero_inv:      "(λb. poly_y p b = 0) constant_on S"
  and distinct_p_inv: "(λb. card (proots (p_y b))) constant_on S"
  and deg_q_inv:      "(λb. degree (poly_y q b)) constant_on S"
  and qzero_inv:      "(λb. poly_y q b = 0) constant_on S"
  and distinct_q_inv: "(λb. card (proots (q_y b))) constant_on S"
  and common_pq_inv:  "(λb. degree (gcd (p_y b) (q_y b))) constant_on S"
shows "(λb. card (proots (poly_y (p * q) b))) constant_on S"
```

where

- `poly_y p b` and `poly_y q b` are polynomials with real coefficients by substituting `b` for the variable `y` in $p \in \mathbb{R}[y,x]$ and $q \in \mathbb{R}[y,x]$ respectively.

- p_y `b` and q_y `b` are polynomials with complex coefficients converted from `poly_y p b` and `poly_y q b`, respectively, by embedding their coefficients into $\mathbb{C}$.

- The assumption `deg_p_inv` ensures the number of complex roots (counting multiplicity) of `p` stays constant as we instantiate the variable y of `p` with different points in `S`. `pzero_inv` covers an corner case of `deg_p_inv` where we require that `p` remains either zero or non-zero as y of `p` varies over `S`.

- The assumption `distinct_p_inv` (`distinct_q_inv`) states that the number of distinct complex roots of `p` (`q`) stays constant as y of `p` (`q`) varies over `S`.

- The assumption `common_pq_inv` requires that the number of common complex roots (counting multiplicity) between `p` and `q` stays constant as y of `p` and `q` varies over `S`.

- Finally, the conclusion of Lemma 8.3 indicates that the number of distinct real roots of `p * q` is constant as y varies over `S`.

Proof of Lemma 8.3. We can first apply the following lemma:

Lemma 8.4 (*locally_constant_imp_constant*).

```
fixes S::"'a::topological_space set" and f::"'a ⇒ 'b"
assumes "connected S"
    and opI: "⋀a. a ∈ S ⟹
              ∃T. open T ∧ a ∈ T ∩ S ∧ (∀x ∈ T ∩ S. f x = f a)"
shows "f constant_on S"
```

to transform our goal into some local equality: given $a \in S$, we need to find an open neighbourhood T of a such that for all $b \in T \cap S$, the total number of distinct real roots of $(pq)(b,x)$ is equal to that of $(pq)(a,x)$. Here, $(pq)(b,x)$ and $(pq)(a,x)$ are polynomials with real coefficients instantiated from $pq \in \mathbb{R}[y,x]$ by substituting b and a, respectively, for the variable y.

Let $z_1, z_2, \ldots, z_m$ be the distinct complex roots of $(pq)(a,x)$. Let v_j $(1 \leq j \leq m)$ be the multiplicity of z_j as a root of $p(a,x)$, and u_j $(1 \leq j \leq m)$ be that of z_j as a root of $q(a,x)$. We can first choose a small $\varepsilon > 0$ such that closed balls $\text{cball}(z_j, \varepsilon)$ $(1 \leq j \leq m)$ are disjoint. With Theorem 8.1 and the assumptions *deg_p_inv* and *deg_q_inv*, we can obtain $\delta > 0$ such that for each $b \in \text{ball}(a, \delta) \cap S$ and z_j $(1 \leq j \leq m)$, the numbers of complex roots (counting multiplicity) of $p(b,x)$ and $q(b,x)$ within the open ball $\text{ball}(z_j, \varepsilon)$ are v_j and u_j respectively. By considering the assumptions *distinct_p_inv* and *distinct_q_inv*, we can further derive that $p(b,x)$ $(q(b,x))$ has exactly one distinct root within $\text{ball}(z_j, \varepsilon)$ when $v_j > 0$ $(u_j > 0)$.

Note that $\min(v_j, u_j)$ is the multiplicity of z_j of $\gcd(p(a,x), q(a,x))$, and when $\min(v_j, u_j) > 0$, by following the assumption *common_pq_inv* we have that $\gcd(p(b,x), q(b,x))$ has exactly one distinct root in $\mathbb{C}$ within $\text{ball}(z_j, \varepsilon)$. Then, by considering the fact that within $\text{ball}(z_j, \varepsilon)$ the added number of distinct complex roots of $(pq)(b,x)$ and $\gcd(p(b,x), q(b,x))$ is equal to that of $p(b,x)$ and $q(b,x)$, we end up deriving that for each $1 \leq j \leq m$ the polynomial $(pq)(b,x)$ has exactly one complex root w_j within $\text{ball}(z_j, \varepsilon)$.

Now, let us consider the relationship between w_j and z_j. If $w_j \in \mathbb{R}$, then $z_j \in \mathbb{R}$, otherwise $w_j \in \text{ball}(z_j, \varepsilon)$ and $w_j \in \text{ball}(\overline{z_j}, \varepsilon)$ that contradicts the disjointness between $\text{ball}(z_j, \varepsilon)$ and $\text{ball}(\overline{z_j}, \varepsilon)$. Moreover, if $w_j \notin \mathbb{R}$, then $z_j \notin \mathbb{R}$, otherwise w_j and $\overline{w_j}$ are two distinct complex roots within $\text{ball}(z_j, \varepsilon)$. As a result, for each $1 \leq j \leq m$ we have $w_j \in \mathbb{R}$ if and only if $z_j \in \mathbb{R}$, following which the polynomials $(pq)(b,x)$ and $(pq)(a,x)$ have the same number of distinct real roots, provided $b \in \text{ball}(a, \delta) \cap S$. □

The proof above follows Basu et al.'s formulation [8, Proposition 5.13], which is actually a proof of Theorem 2.6. For this reason, I believe my proof of Lemma 8.3 can be adapted to derive Theorem 2.6 without much effort once we have an appropriate library of multivariate polynomials in Isabelle/HOL.

8.3 Towards certifying multivariate CAD

Regarding the multivariate case of CAD, I will still aim for a certificate-based approach similar to the univariate case in Chapter 5. The main rationale is efficiency: it is unlikely for me to implement an efficient verified CAD procedure that is capable of handling non-toy problems. In addition to that, a certificate-based design allows us to smoothly switch from one implementation of CAD to another, considering CAD is still an active field with new algorithms being proposed regularly [27, 17].

Following the idea of certifying univariate CAD in Chapter 5, given a first-order formula over reals with n variables, we extract a set of polynomials $P \subseteq \mathbb{R}[x_1, x_2, \ldots, x_n]$ from the formula and pass P to external untrusted programs for CAD. With a set of sample points $S_n \subseteq \mathbb{R}^n$ from the external program, we need to check if

- each sample point $x \in S_n$ can indeed represent a cell in a CAD adapted to P,
- and by deciding the sign of each $p \in P$ at these sample points the target formula is indeed true.

The latter part should not be too hard, as we already have a verified sign determination procedure for bivariate polynomials at real algebraic points (as presented in §4.2.3 of Chapter 4), which should be generalised to multivariate cases without much trouble. Therefore, the real problem that I need to deal with is the former (i.e., these sample points are genuine representatives).

Recall that in Algorithm 2 in Chapter 5, we explicitly check roots from external programs are indeed all the roots of the extract polynomials, and because of the intermediate value theorem we know those sample points (constructed from those untrusted roots) are representative. Analogously, my current plan for the multivariate case is to check if S_n contain all the solutions of the triangular system of equations $\prod \mathrm{proj}^{n-1}(P) = 0$, $\prod \mathrm{proj}^{n-2}(P) = 0$, ..., $\prod \mathrm{proj}^{0}(P) = 0$, where $\mathrm{proj}(-)$ is the projection operation in Algorithm 1, proj^0 is the identity function and $\mathrm{proj}^{i+1}(P) = \mathrm{proj}^{i}(\mathrm{proj}(P))$ for all i. The idea behind this is to certify that S_n is indeed the result of the base case and the lifting phase in Algorithm 1.

This is my rough idea at the time of writing this thesis. Subtleties are expected to appear when I undertake the formalisation, and the plan may be revised in the process.

Chapter 9

Conclusion

Aiming at certifying multivariate CAD in Isabelle, this thesis started with a formal proof of the Sturm-Tarski theorem (Chapter 3), which enables us to effectively compute the Tarski query and the Cauchy index through remainder sequences. Subsequently, I proposed a library for real algebraic numbers (Chapter 4), whose notable features include its modular design and sign determination procedures that only require (dyadic) arithmetic. With a formalisation of real algebraic numbers and sign determination procedures, I successfully certified univariate CAD in a certificate-based approach (Chapter 5). To pave the way for multivariate CAD, I formalised some results in complex analysis: Cauchy's residue theorem, the argument principle and Rouché's theorem (Chapter 6), following which I built a tactic to evaluate winding numbers through Cauchy indices and verified procedures to count complex roots inside a rectangle and a half-plane (Chapter 7). Finally, I presented a formal proof towards the projection theorem in CAD (Chapter 8).

The quest towards certifying multivariate CAD is still ongoing.

9.1 On formalised mathematics

Over the last four years, I spent much of my time formalising mathematics (to justify executable procedures). The process was both fun and exhausting. Good things about formalised mathematics (in a proof assistant) include:

- Being extremely precise. I have uncovered numerous ambiguities, hidden assumptions and even small errors when following (informal) proofs in the literature. With my proofs being checked by computers, I feel great confidence in their correctness (and relaxed).

- The ability to archive mathematical theorems and proofs. Every mathematical theorem is usually formulated by various authors in different forms, and its proofs are scattered in the literature. With proof assistants, those theorems (and their proofs) can be archived on a unified system, hence many of the compatibility issues can be resolved and proofs can be compared on a common ground. Especially with structured proofs in Isabelle, proof blocks can be folded to allow proof scripts to be presented in a sufficiently abstract way. In fact, when looking up an unfamiliar theorem, I would rather consult the Isabelle library, where theorems are unambiguously stated and proofs are recorded in a human-understandable way.

- Easy to collaborate. Because of the unified system (and logic) and machine-guaranteed preciseness, formal proofs produced by one mathematician (or computer scientist) can be safely reused by another, as long as they stay in the same system.

The main drawback with formalised mathematics is, unsurprisingly, lack of automation: huge effort is needed to elaborate our informal proofs so that computers can understand. Fortunately, formalised mathematics is generally reusable, so the labour of formalising can be considered as a process of accumulation. Besides, with recent breakthrough in machine learning, I am optimistic about the level of automation (in proof assistants) we can achieve in the future.

Apart from automation, I also wish we had better search, management and refactoring tools, so that managing mechanized mathematics can be similar to maintaining a huge codebase in software companies; in fact, I do not see much difference between them. Luckily, with still improving proof environments (i.e., Prover IDE in Isabelle), I believe we are on the right track.

9.2 Computer algebra in proof assistants

Besides my work, there has been a recent trend of formalising computer algebra algorithms in proofs assistants [32, 66, 4]. Compared to commercial systems like Mathematica and Maple, this formalisation effort has its advantages:

- The verified procedures are usually open-sourced and accompanied by formal proofs, while commercial systems are mostly used as a black box.

- Those verified procedures are generally built within a more expressive logic, hence are more flexible and applicable when formalising mathematics. In comparison, procedures provided by commercial systems are mostly applicable for concrete data types (e.g.,

$\mathbb{Z}$, $\mathbb{R}$ or $\mathbb{C}$). For example, suppose we want to factorise a polynomial over an integral domain for our proof, we may want a procedure expressed in a richer logic.

- Being part of an interactive tactic, the verified procedures might offer a better experience of interactivity in complex situations (e.g., when simplifying an expression with many open variables).

However, the largest issue with verified procedures is efficiency. Modern commercial systems are usually highly tuned, and the sophistication of their implementations can go beyond our current verification capabilities (or at least with reasonable effort). As a result, unverified procedures in commercial systems are often far more efficient than our verified ones. How can we bridge the gap of efficiency? The certificate-based approach I advocated is just one way to alleviate this situation, and this approach also brings a new question – how to design a good certificate? The sums-of-squares tactic represents the best-case scenario, where almost all the hard work is delegated to external programs and certifying the results only requires simple sign-based reasoning and rational arithmetic. As for CAD, even the univariate case requires more mathematics (e.g., real algebraic numbers and the Sturm-Tarski theorem) and more computation (especially for the universal case). In general, I believe that a good certificate design needs to balance the difficulty of the formalisation effort and verified computation required to check the certificates with the efficiency improvements offered by offloading the construction of the certificates to high-performance external tools.

References

[1] Ahlfors, L. V. (1966). *Complex Analysis: An Introduction to the Theory of Analytic Funtions of One Complex Variable*. McGraw-Hill, New York.

[2] Akbarpour, B. and Paulson, L. C. (2010). MetiTarski - An Automatic Theorem Prover for Real-Valued Special Functions. *Journal of Automated Reasoning*.

[3] Alexanderian, A. (2013). On continuous dependence of roots of polynomials on coefficients.

[4] Aransay, J. and Divasón, J. (2015). Formalisation in higher-order logic and code generation to functional languages of the Gauss-Jordan algorithm. *Journal of Functional Programming*, 25:147.

[5] Arnold, V. I. (1992). *Ordinary Differential Equations*. Springer.

[6] Avigad, J., Hölzl, J., and Serafin, L. (2017). A Formally Verified Proof of the Central Limit Theorem. *Journal of Automated Reasoning*, pages 1–35.

[7] Bak, J. and Newman, D. (2010). *Complex Analysis*. Springer.

[8] Basu, S., Pollack, R., and Roy, M.-F. (2006). *Algorithms in Real Algebraic Geometry*, volume 10 of *Algorithms and Computation in Mathematics*. Springer, Berlin, Heidelberg.

[9] Bertot, Y. and Castéran, P. (2013). *Interactive theorem proving and program development: Coq'Art: the calculus of inductive constructions*. Springer.

[10] Boldo, S., Lelay, C., and Melquiond, G. (2014). Coquelicot: A user-friendly library of real analysis for Coq. *Mathematics in Computer Science*, 9(1):41–62.

[11] Brown, C. W. (2001). Improved Projection for Cylindrical Algebraic Decomposition. *Journal of Symbolic Computing*, 32(5):447–465.

[12] Brown, C. W. (2003). QEPCAD B - a program for computing with semi-algebraic sets using CADs. *ACM SIGSAM Bulletin*, 37(4):97.

[13] Brown, C. W. and Davenport, J. H. (2007). The complexity of quantifier elimination and cylindrical algebraic decomposition. In *Proceedings of the 32th International Symposium on Symbolic and Algebraic Computation, ISSAC '07*, pages 54–60, Waterloo, Ontario, Canada. ACM Press.

[14] Brunel, A. (2011). Non-constructive complex analysis in Coq. In *18th International Workshop on Types for Proofs and Programs, TYPES 2011, September 8-11, 2011, Bergen, Norway*, pages 1–15.

[15] Caviness, B. F. and Johnson, J. R. (2012). *Quantifier Elimination and Cylindrical Algebraic Decomposition*. Texts and Monographs in Symbolic Computation. Springer, Vienna.

[16] Chaieb, A. et al. (2008). Automated methods for formal proofs in simple arithmetics and algebra. *Diss., Technische Universität, München*.

[17] Chen, C. and Moreno Maza, M. (2014). Cylindrical Algebraic Decomposition in the RegularChains Library. In Hong, H. and Yap, C., editors, *Proceedings of the th International Congress on Mathematical Software, ICMS '14*, Seoul, South Korea. Springer.

[18] Cheng, J.-S., Gao, X.-S., and Yap, C.-K. (2007). Complete numerical isolation of real zeros in zero-dimensional triangular systems. In *Proceedings of the International Symposium on Symbolic and Algebraic Computation, ISSAC '07*, pages 92–99. ACM.

[19] Clarke, E. M., Grumberg, O., and Peled, D. (1999). *Model Checking*. MIT Press.

[20] Cohen, C. (2012a). Construction of Real Algebraic Numbers in Coq. In Beringer, L. and Felty, A., editors, *Proceedings of the 3rd International Conference on Interactive Theorem Proving, ITP 2012*, pages 67–82, Berlin, Heidelberg. Springer.

[21] Cohen, C. (2012b). *Formalized algebraic numbers: construction and first-order theory*. PhD thesis, École polytechnique, Laboratoire d'informatique de l'École polytechnique - LIX, INRIA Saclay - Ile de France, Microsoft Research - Inria Joint Centre.

[22] Collins, G. E. (1975). Quantifier elimination for real closed fields by cylindrical algebraic decompostion. In *Automata Theory and Formal Languages 2nd GI Conference Kaiserslautern, May 20–23, 1975*, pages 134–183. Springer.

[23] Collins, G. E. and Hong, H. (1991). Partial Cylindrical Algebraic Decomposition for Quantifier Elimination. *Journal of Symbolic Computing*, 12(3):299–328.

[24] Collins, G. E. and Krandick, W. (1992). An efficient algorithm for infallible polynomial complex root isolation. In *Proceedings of International Symposium on Symbolic and Algebraic Computation, ISSAC '92*, pages 189–194, Berkeley, CA, USA. ACM.

[25] Conway, J. B. (1978). *Functions of One Complex Variable*, volume 11. Springer, New York, second edition.

[26] Cruz-Filipe, L., Geuvers, H., and Wiedijk, F. (2004). C-CoRN, the constructive Coq repository at Nijmegen. In *Mathematical Knowledge Management*, pages 88–103. Springer.

[27] Davenport, J. H. and England, M. (2015). Recent Advances in Real Geometric Reasoning. In Botana, F. and Quaresma, P., editors, *Automated Deduction in Geometry: 10th International Workshop, ADG 2014, Coimbra, Portugal, July 9-11, 2014, Revised Selected Papers*, pages 37–52. Springer, Cham.

[28] de Moura, L. and Bjørner, N. (2008). Z3 - An Efficient SMT Solver. *TACAS*, 4963(Chapter 24):337–340.

[29] de Moura, L. and Passmore, G. O. (2013). Computation in Real Closed Infinitesimal and Transcendental Extensions of the Rationals. In *Proceedings of the 24th International Conference on Automated Deduction, CADE '13*, pages 178–192, Berlin, Heidelberg. Springer.

[30] Denman, W. and Muñoz, C. (2014). Automated real proving in PVS via MetiTarski. In *International Symposium on Formal Methods*, pages 194–199. Springer.

[31] Divasón, J. and Aransay, J. (2013). Rank-Nullity Theorem in Linear Algebra. *Archive of Formal Proofs*. http://isa-afp.org/entries/Rank_Nullity_Theorem.html, Formal proof development.

[32] Divasón, J., Joosten, S. J. C., Thiemann, R., and Yamada, A. (2017). A formalization of the Berlekamp-Zassenhaus factorization algorithm. In *Proceedings of the 6th ACM SIGPLAN Conference on Certified Programs and Proofs, CPP 2017*, pages 17–29, Paris, France.

[33] Eberl, M. (2015). A Decision Procedure for Univariate Real Polynomials in Isabelle/HOL. In *Proceedings of the 2015 Conference on Certified Programs and Proofs, CPP 2015*, pages 75–83, Mumbai, India. ACM Press.

[34] Eisermann, M. (2012). The Fundamental Theorem of Algebra Made Effective: An Elementary Real-algebraic Proof via Sturm Chains. *The American Mathematical Monthly*, 119(9):715.

[35] Gao, S., Kong, S., and Clarke, E. M. (2014). Proof generation from delta-decisions. In *Proceedings of the 16th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing, SYNASC 2014*, pages 156–163. IEEE.

[36] Gonthier, G., Asperti, A., Avigad, J., Bertot, Y., Cohen, C., Garillot, F., Le Roux 0001, S., Mahboubi, A., O'Connor, R., Biha, S. O., Pasca, I., Rideau, L., Solovyev, A., Tassi, E., and Théry, L. (2013). A Machine-Checked Proof of the Odd Order Theorem. In Blazy, S., Paulin-Mohring, C., and Pichardie, D., editors, *Proceedings of the 4th International Conference on Interactive Theorem Proving, ITP 2013*, pages 163–179, Rennes, France. Springer.

[37] Haftmann, F. and Bulwahn, L. (2017). Code generation from Isabelle/HOL theories.

[38] Haftmann, F., Lochbihler, A., and Schreiner, W. (2014). Towards abstract and executable multivariate polynomials in Isabelle. In *Isabelle Workshop*, volume 201.

[39] Haftmann, F. and Nipkow, T. (2010). Code generation via higher-order rewrite systems. In *International Symposium on Functional and Logic Programming*, pages 103–117. Springer.

[40] Hales, T., Adams, M., Bauer, G., Dang, D. T., Harrison, J., Le Hoang, T., Kaliszyk, C., Magron, V., McLaughlin, S., Nguyen, T. T., Nguyen, T. Q., Nipkow, T., Obua, S., Pleso, J., Rute, J., Solovyev, A., Ta, A. H. T., Tran, T. N., Trieu, D. T., Urban, J., Vu, K. K., and Zumkeller, R. (2015). A formal proof of the Kepler conjecture. *arXiv.org*.

[41] Harrison, J. (2007a). Formalizing basic complex analysis. In Matuszewski, R. and Zalewska, A., editors, *From Insight to Proof: Festschrift in Honour of Andrzej Trybulec*, volume 10(23), pages 151–165. University of Białystok.

[42] Harrison, J. (2007b). Verifying nonlinear real formulas via sums of squares. In Schneider, K. and Brandt, J., editors, *Proceedings of the 20th International Conference on Theorem Proving in Higher Order Logics, TPHOLs 2007*, volume 4732 of *Lecture Notes in Computer Science*, pages 102–118, Kaiserslautern, Germany. Springer.

[43] Harrison, J. (2009a). Formalizing an analytic proof of the Prime Number Theorem (dedicated to Mike Gordon on the occasion of his 60th birthday). *Journal of Automated Reasoning*, 43:243–261.

[44] Harrison, J. (2009b). HOL Light: An overview. In Berghofer, S., Nipkow, T., Urban, C., and Wenzel, M., editors, *Proceedings of the 22nd International Conference on Theorem Proving in Higher Order Logics, TPHOLs 2009*, volume 5674 of *Lecture Notes in Computer Science*, pages 60–66, Munich, Germany. Springer.

[45] Harrison, J. (2013). The HOL Light theory of Euclidean space. *Journal of Automated Reasoning*, 50:173–190.

[46] Harrison, J. and Théry, L. (1998). A Skeptic's Approach to Combining HOL and Maple. *Journal of Automated Reasoning*, 21:279–294.

[47] Hölzl, J. (2009). Proving inequalities over reals with computation in Isabelle/HOL. In *International Workshop on Programming Languages for Mechanized Mathematics Systems*, pages 38–45.

[48] Hölzl, J., Immler, F., and Huffman, B. (2013). Type classes and filters for mathematical analysis in Isabelle/HOL. 7998:279–294.

[49] Hong, H. (1990). An Improvement of the Projection Operator in Cylindrical Algebraic Decomposition. In *Proceedings of the International Symposium on Symbolic and Algebraic Computation, ISSAC '90*, pages 261–264, Tokyo, Japan. ACM.

[50] Huang, Z., England, M., Wilson, D. J., Davenport, J. H., Paulson, L. C., and Bridge, J. P. (2014). Applying Machine Learning to the Problem of Choosing a Heuristic to Select the Variable Ordering for Cylindrical Algebraic Decomposition. *CICM*, 8543(3):92–107.

[51] Huffman, B. and Kunčar, O. (2013). Lifting and Transfer: A modular design for quotients in Isabelle/HOL. In *Certified Programs and Proofs*, pages 131–146. Springer.

[52] Jirstrand, M. et al. (1995). *Cylindrical algebraic decomposition: an introduction*. Linköpings university.

[53] Klein, G., Andronick, J., Elphinstone, K., Heiser, G., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., Sewell, T., Tuch, H., and Winwood, S. (2010). seL4 - formal verification of an operating-system kernel. *Communications of ACM*, 53(6):107.

[54] Kumar, R., Arthan, R., Myreen, M. O., and Owens, S. (2016). Self-Formalisation of Higher-Order Logic - Semantics, Soundness, and a Verified Implementation. *Journal of Automated Reasoning*, 56(3):221–259.

[55] Lang, S. (1993). *Complex Analysis*. Springer.

[56] Leroy, X. (2009). Formal verification of a realistic compiler. *Communications of ACM*, 52(7):107.

[57] Li, W. (2014). The Sturm-Tarski Theorem. *Archive of Formal Proofs*. http://isa-afp.org/entries/Sturm_Tarski.html, Formal proof development.

[58] Li, W. (2017a). Count the Number of Complex Roots. *Archive of Formal Proofs*. http://isa-afp.org/entries/Count_Complex_Roots.html, Formal proof development.

[59] Li, W. (2017b). Evaluate Winding Numbers through Cauchy Indices. *Archive of Formal Proofs*. http://isa-afp.org/entries/Winding_Number_Eval.html, Formal proof development.

[60] Li, W., Passmore, G. O., and Paulson, L. C. (2017). Deciding Univariate Polynomial Problems Using Untrusted Certificates in Isabelle/HOL. *Journal of Automated Reasoning*, 44(3):175–23.

[61] Li, W. and Paulson, L. C. (2016a). A formal proof of Cauchy's residue theorem. In Blanchette, J. C. and Merz, S., editors, *Proceedings of the 4th International Conference on Interactive Theorem Proving, ITP 2013*, pages 235–251, Nancy, France. Springer.

[62] Li, W. and Paulson, L. C. (2016b). A modular, efficient formalisation of real algebraic numbers. In Avigad, J. and Chlipala, A., editors, *Proceedings of the 5th ACM SIGPLAN Conference on Certified Programs and Proofs, CPP 2016*, pages 66–75, St. Petersburg, FL, USA. ACM.

[63] Lochbihler, A. and Züst, M. (2014). Programming TLS in Isabelle/HOL. In *Isabelle Workshop*, volume 2014.

[64] Mahboubi, A. (2007). Implementing the cylindrical algebraic decomposition within the Coq system. *Mathematical Structures in Computer Science*, 17(1):99–127.

[65] Mahboubi, A. and Cohen, C. (2012). Formal proofs in real algebraic geometry: from ordered fields to quantifier elimination. *Logical Methods in Computer Science*, 8(1).

[66] Mahboubi, A., Melquiond, G., and Sibut-Pinote, T. (2016). Formally Verified Approximations of Definite Integrals. In Blanchette, J. C. and Merz, S., editors, *Proceedings of the 7th International Conference on Interactive Theorem Proving, ITP 2016*, pages 274–289, Nancy, France. Springer.

[67] Marden, M. (1949). *Geometry of Polynomials. Second Edition*. American Mathematical Society, Providence, Rhode Island.

[68] McCallum, S. (1988). An improved projection operation for cylindrical algebraic decomposition of three-dimensional space. *Journal of Symbolic Computation*, 5(1-2):141–161.

[69] McLaughlin, S. and Harrison, J. (2005). A proof-producing decision procedure for real arithmetic. volume 3632 of *Lecture Notes in Computer Science*, pages 295–314, Tallinn, Estonia. Springer.

[70] Mishra, B. (1993). *Algorithmic Algebra*. Springer.

[71] Narkawicz, A., Muñoz, C. A., and Dutle, A. (2015). Formally-Verified Decision Procedures for Univariate Polynomial Computation Based on Sturm's and Tarski's Theorems. *Journal of Automated Reasoning*, 54(4):285–326.

[72] Narkawicz, A. J. and Muñoz, C. A. (2014). A formally-verified decision procedure for univariate polynomial computation based on Sturm's theorem. Technical Memorandum NASA/TM-2014-218548, NASA, Langley Research Center, Hampton VA 23681-2199, USA.

[73] Nipkow, T., Paulson, L. C., and Wenzel, M. (2016). *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*.

[74] Passmore, G. O., Paulson, L. C., and de Moura, L. (2012). Real algebraic strategies for MetiTarski proofs. In *Intelligent Computer Mathematics*, pages 358–370. Springer.

[75] Paulson, L. C. (1994). *Isabelle: A generic theorem prover*, volume 828. Springer.

[76] Paulson, L. C. (2010). Three Years of Experience with Sledgehammer, a Practical Link between Automatic and Interactive Theorem Provers. In *Proceedings of the 2nd Workshop on Practical Aspects of Automated Reasoning, PAAR-2010, Edinburgh, Scotland, UK, July 14, 2010*, pages 1–10.

[77] Platzer, A. (2008). *Differential Dynamic Logics: Automated Theorem Proving for Hybrid Systems*. PhD thesis, Department of Computing Science, University of Oldenburg.

[78] Rahman, Q. I. and Schmeisser, G. (2016). *Analytic Theory of Polynomials(2002)*. Oxford University Press.

[79] Sagraloff, M. (2010). A general approach to isolating roots of a bitstream polynomial. *Mathematics in Computer Science*, 4(4):481–506.

[80] Solovyev, A. and Hales, T. C. (2013). Formal Verification of Nonlinear Inequalities with Taylor Interval Approximations. In *NASA Formal Methods*, pages 383–397. Springer.

[81] Stein, E. M. and Shakarchi, R. (2010). *Complex Analysis*, volume 2. Princeton University Press.

[82] Strzeboński, A. W. (2006). Cylindrical Algebraic Decomposition using validated numerics. *Journal of Symbolic Computation*, 41(9):1021–1038.

[83] Thiemann, R. and Yamada, A. (2016). Algebraic Numbers in Isabelle/HOL. In Blanchette, J. C. and Merz, S., editors, *Proceedings of the 7th International Conference on Interactive Theorem Proving, ITP 2016*, pages 391–408, Nancy, France. Springer.

[84] Wilf, H. S. (1978). A Global Bisection Algorithm for Computing the Zeros of Polynomials in the Complex Plane. *Journal of the ACM (JACM)*, 25(3):415–420.

[85] Yap, C.-K. and Sagraloff, M. (2011). A simple but exact and efficient algorithm for complex root isolation. In *Proceedings of the 36th International Symposium on Symbolic and Algebraic Computation, ISSAC '11*, page 353, San Jose, CA, USA. ACM Press.