Intuition in Formal Proof: A Novel Framework for Combining Mathematical Tools

Laura Isabel Meikle

Doctor of Philosophy

Centre for Intelligent Systems and their Applications

School of Informatics

University of Edinburgh

2014

Abstract

This doctoral thesis addresses one major difficulty in formal proof: removing obstructions to intuition which hamper the proof endeavour. We investigate this in the context of formally verifying geometric algorithms using the theorem prover Isabelle, by first proving the Graham's Scan algorithm for finding convex hulls, then using the challenges we encountered as motivations for the design of a general, modular framework for combining mathematical tools.

We introduce our integration framework — the Prover's Palette, describing in detail the guiding principles from software engineering and the key differentiator of our approach — emphasising the role of the user. Two integrations are described, using the framework to extend Eclipse Proof General so that the computer algebra systems QEPCAD and Maple are directly available in an Isabelle proof context, capable of running either fully automated or with user customisation. The versatility of the approach is illustrated by showing a variety of ways that these tools can be used to streamline the theorem proving process, enriching the user's intuition rather than disrupting it. The usefulness of our approach is then demonstrated through the formal verification of an algorithm for computing Delaunay triangulations in the Prover's Palette.

Acknowledgements

This thesis is the culmination of many years of work and would not have been possible without the encouragement and support of so many. I am indebted most to my supervisor Jacques Fleuriot for the guidance, wisdom, and encouragement he gave throughout. The hours he dedicated to answering my questions and supporting my research are much appreciated, and easily worth the grey hair or two I am now blamed for! Helpful also were the insightful comments of my examiners, David Aspinall and Christoph Lüth, which have given this final version of the thesis a better flow. Thank you.

I am also grateful to my family: to my mum and dad, who not only gave hours of childcare happily, but also the constant reminder to keep life balanced; to Alex whose love gave me strength which sustained me and whose wizardry with Java saved many a sleepless night; and finally to my boys, Lucas and Reuben, who have taught me more about the limits of logic than any theorem prover ever could. This is for you.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Laura Isabel Meikle*)

Table of Contents

Chapter 1

Maths through the Looking Glass

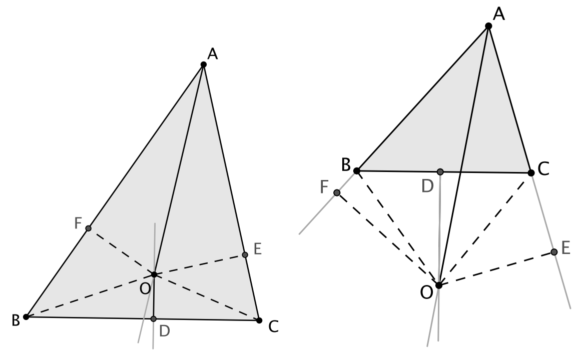
Intuition is a fundamental aspect of human knowledge, but its fit and proper role has long been contentious. On the isles of Ancient Greece, intuitive Platonic ideals did battle with Aristotelian knowledge acquisition, and the argument raged in the abstract, through Descartes and Hobbes, to Kant and countless other philosophers. Meanwhile, farms made food, science made discoveries, and mathematicians made new theories. The real world was largely unaffected — at least until the 19th Century, when real analysis and non-Euclidean geometries gave birth to "monsters" which defied intuition. These monsters in turn bred a new emphasis on rigour, in mathematics initially, but the impact of the logical frameworks so borne has since left no discipline untouched, and the Boolean digital world is ineluctable. At the pinnacle of this formal, deductive methodology stands software capable of verifying any proof possible in logic; but this software stands unremarked outside a small community, and the place for intuition is still uncertain.

In this chapter we will establish two humble claims: that intuition is powerful, and that intuition is fallible; and we will look through these looking glasses to examine the interplay between intuition and the logic of mathematics.

1.1 All Triangles are Isosceles

Let us begin by considering two proofs, where the juxtaposition will provide some illustration of how intuition and logic inform our thought processes.

THEOREM. For any triangle ABC in Euclidean space, either $AB = AC$, $AB = BC$, or $AC = BC$. In other words, the triangle is isosceles.



PROOF. Begin by drawing the bisector of $\angle BAC$, and the bisector of BC through midpoint D.

If these two lines are parallel or identical (not shown) then the angle bisector must meet BC at a 90° angle. Call this intersection D'. Triangles $AD'B$ and $AD'C$ are congruent because the side AD' is in common, $\angle BAD' = \angle CAD'$ (angle bisector), and $\angle AD'B = \angle AD'C$ (right angles). Thus $AB = AC$.

Otherwise call the intersection of the two bisectors O, and drop perpendiculars from O to E and F on rays AC and AB respectively.

Triangles AOF and AOE are congruent since the side AO is common, $\angle OAF = \angle OAE$, and $\angle OFA = \angle OEA$. Hence $AF = AE$ and $OF = OE$. Triangles OBD and OCD are congruent, using common side OD, $\angle BDO = \angle CDO$ (right angles), and $BD = CD$ (midpoint), so $OB = OC$. Lastly, triangles OFB and OEC are congruent, because $OF = OE$, $OB = OC$, and $\angle OFB = \angle OEC$ (right angles), implying $FB = EC$.

For O inside triangle ABC (above left) this gives:

$$AB = AF + FB = AE + EC = AC$$

And for O outside the triangle (above right):

$$AB = AF - BF = AE - CE = AC$$

In both cases, ABC is isosceles. □

THEOREM. For three points $x, y, z \in \mathbb{Q}$ with p-adic distance function d, it must be the case that either $d(x,y) = d(x,z)$, $d(x,y) = d(y,z)$, or $d(x,z) = d(y,z)$.

PRELIMINARIES. p-adic numbers are a way of extending the rationals $\mathbb{Q}$ by focussing on powers of a prime p.

For non-zero x in $\mathbb{Q}$, the ***p*-adic valuation** $v_p(x)$ is defined as the integer n such that $p^n \cdot \frac{a}{b} = x$ where a and b are in $\mathbb{Z}$ and p does not divide a or b. It is easily shown that $v_p(xy) = v_p(x) + v_p(y)$ and that $v_p(-x) = v_p(x)$, and it is not hard to prove that $v_p(x+y) \geq \min\big(v_p(x), v_p(y)\big)$.

The ***p*-adic absolute value** for $x \in \mathbb{Q}$ is then defined as $|x|_p = p^{-v_p(x)}$ with $|0|_p = 0$ by convention. This yields a ***p*-adic distance metric** in the same manner as the usual distance function, namely $d(x,y) = |x-y|_p$.

PROOF. Without loss of generality we can take $d(x,y)$ to be the largest distance, that is:

$$d(x,y) \geq \max(d(x,z), d(y,z))$$

If any two points are identical, the theorem is trivially true. Otherwise, the p-adic valuation is defined for all differences, and by manipulating the p-adic valuation additive inequality, we can write:

$$\begin{aligned} v_p(x-y) &= v_p\big((x-z)+(z-y)\big) \\ &\geq \min\big(v_p(x-z), v_p(z-y)\big) \\ &= \min\big(v_p(x-z), v_p(y-z)\big) \end{aligned}$$

Raising p to the power of both sides gives:

$$\begin{aligned} p^{v_p(x-y)} &\geq p^{\min\left(v_p(x-z), v_p(y-z)\right)} \\ p^{-v_p(x-y)} &\leq p^{-\min\left(v_p(x-z), v_p(y-z)\right)} \\ &= \max\left(p^{-v_p(x-z)}, p^{-v_p(y-z)}\right) \\ d(x,y) &\leq \max\big(d(x,z), d(y,z)\big) \end{aligned}$$

As we began by choosing $d(x,y)$ to be the largest of the three distances, we now have $d(x,y) = \max\big(d(x,z), d(y,z)\big)$. □

FIGURE 1.1: **Two Proofs of the Assertion that All Triangles Are Isosceles.** The left-hand proof is flawed but hard to correct. The other is correct but hard to understand. The Euclidean proof at left is based on one by Rouse Ball [12]. The p-adic proof at right is by us after a comment by Gouvêa [68].

Figure 1.1 shows two "proofs" of the statement that "all triangles are isosceles". The result is intuitively absurd, and so we have an immediate suspicion: particularly in Euclidean space, we expect the reader's intuition to be so powerful that the proof will be rejected from the very start — and rightly so. However, examining this proof to find the error is often quite difficult[1]: each step seems logically correct, and yet since we know the conclusion is false, we have been led astray somewhere by relying on intuition without realising it. This demonstrates well that intuition is fallible.

Turning now to the p-adic proof: few people have the domain experience to make any substantial use of intuition here. The reader with a working knowledge of analysis may symbolically confirm the validity of each step, and the ultimate benefit of logic and rigour is well illustrated, when this reader finishes by confirming the validity of the statement. Without relying on intuition the reader is not misled, for in p-adic space it is true that all triangles are isosceles. However, without intuition, the concepts are unfamiliar or abstract, the steps take more effort to follow, and worst of all the result may seem quite meaningless.

The following sections will delve more deeply into the ways that intuition is powerful, and then the ways that is is fallible. Subsequently we take a look at how logic has developed to keep intuition in check. We conclude with a discussion of the nature of proof: what is the goal, and where do logic and intuition fit in achieving it.

1.2 Intuition is Powerful

We begin with a paean to intuition, the mysterious and powerful force which drives our discovery of new insights going forward, which underpins our understanding of the world around us, and which plays such an important part in sharing that knowledge with others. We will examine these three areas — discovery, understanding, and communication — with special regard for its relevance to mathematics and proof, but let us first give a definition. There is surprising consensus among dictionaries on this point; Oxford's is particularly pleasing: "the ability to understand something immediately, without the need for conscious reasoning" [139].

[1]We presented this proof to an audience of 20 postgraduate students of computer science (many of whom had studied mathematics at undergraduate level), and none could identify the flaw after three minutes, after which they were put out of their misery. The present reader is not given that luxury here but will find good exposition in "The Lewis Carroll Picture Book" [32], to which the inspiration for the chapter title is due. There is also an interesting analysis of this fallacy by Mumma [128].

1.2.1 Discovery: Spontaneous Intuitions

> You enter the first room of the mansion and it's completely dark. You stumble around bumping into the furniture but gradually you learn where each piece of furniture is. Finally, after six months or so, you find the light switch, you turn it on, and suddenly it's all illuminated. You can see exactly where you were. Then you move into the next room . . . [135]

Interviewed on Nova, Andrew Wiles gave the metaphor above to illustrate his experience researching mathematics. The sudden illumination of the room is analogous to the "Eureka!" moment familiar to us all, a "sudden spontaneousness" [72] when deep understanding occurs immediately. Conscious reasoning is bypassed, and precisely how the moment occurs is hard to predict, but the feeling and impact are tremendous: these are new insights behind new discoveries. These are "God's thoughts", in Einstein's words, when his idea of relativity was triggered by a streetcar and a clock tower in Bern, and he described it as "a storm broke loose in my mind" [136].

Much has been written about the process surrounding these insights, from the idea of "restructuring" in Gestalt theory where "the whole is other than the sum of the parts" [103] to Edison's more prosaic formula of "1-percent inspiration and 99-percent perspiration". In mathematical cognition, much of the research stems from lectures by Poincaré [144], including Hadamard's influential four-phase model for mathematical invention [72] — preparation, incubation, illumination, and verification. Pólya includes the tip to have a "bright idea" in his guide *How to Solve It* [146], although his advice on how to go about this is a single word: "patience". It remains a mystery how, but there is widespread agreement that, firstly, these grand intuitions do happen, and secondly, they are fostered by familiarity with a subject.

1.2.2 Understanding: Quotidian Intuition

The consensus that discoveries often rely on domain familiarity, brings us to a second area where intuition plays a major role: in our understanding. Far more frequently than the spontaneous flashes of insight, our intuition helps us make sense of everyday situations, in mathematics, in science, and in the world around us. Logical thought proceeds from it, and is used to evaluate intuitions and thoughts, but the components of our understanding arrive non-logically, as everyday intuitions. One's intuition develops through experiences, drawing on the cultural and scientific heritage, and rendering once-complex thoughts simple. As Isaac Newton put it, "if I have seen a little further than others, it is because I have stood on the shoulders of giants".

With a good intuition in a domain, we can quickly evaluate the plausibility of an idea, whether it is the latest political proposal or the proposition that all triangles are isosceles. This gives the familiarity which permits the first class, grand spontaneous intuitions, but more frequently it allows us to make sense of what we read and to develop new intuitions.

A good illustration of this in mathematics is given by Feferman [53]. Our familiarity with the plane gives us intuition for Euclidean geometry; these geometric and physical intuitions can carry us into the study of analysis in higher dimensional spaces, and from there on into functional analysis . He enumerates many more such examples, from topology to set theory, all with the aim of highlighting how the cultivation of intuition "is essential for motivation of notions and results and to guide one's conceptions via tacit or explicit analogies in the transfer from familiar grounds to unfamiliar terrain."

1.2.3 Communication

The final area we will look at where intuition plays a vital role is in communicating ideas. Analogies contribute immensely to our understanding, and when one wants to teach or develop another's intuition, an effective technique is to appeal to the intuition they already possess.

Take for example the proof we presented earlier, in Figure 1.1, which showed that all triangles are isosceles in p-adic spaces. It is interesting that most people, even those with a passing familiarity with the p-adics, find this proof unintuitive. However, if the presentation were to include the proof of a particular p-adic space, then some people may find the general result a little easier to grasp. For example, if we consider the set of triangles whose sides are all integer powers of 2, then one can convince oneself that these must all be isosceles. In doing so one is developing the intuition which makes this proof more palatable.

Strikingly, the intuitions used to help people develop the right thought patterns need not be correct. Primary schools everywhere display flat world maps, though children are usually familiar with the globe but not with projective geometry and Mercator. So powerful is intuition, we routinely accept that "the ends justify the means", routinely leaning on flawed and inaccurate uses of intuition, in order to develop one's intuition in a new domain.

An important objective of a mathematical proof is to explain a result, to impart the understanding and develop a learner's intuition. Diagrams, for example, are often used to present a result; not only can they cultivate a reader's intuition, but they can ground a proof, making it accessible. As Avigad notes, "some arguments can be nearly unintelligible until one has drawn a good diagram" [9]: the pictorial proof of Pythagoras' Theorem is one good example, rendering the sum of squares familiar long before one has learned about trigonometry.

1.3 Intuition is Fallible

Having established the importance of intuition for understanding, discovering, and communicating, let us now begin the assassination of its character and make the case for logic as a remedy: intuition is a deceptive and unreliable fellow. Diagrams are one example of how he can so cavalierly give us false confidence: it is the diagram for the Euclidean isosceles "proof" in Figure 1.1 which makes the flaw so hard to spot, and in fact this example was cited by Klein for precisely this point [100]. To take another geometric example, reminiscent of the flat world maps in Section 1.2.3, recall the once-common belief that the Earth was flat. Our day-to-day activities are indeed flat; it was ancient Greek philosophers — some say Pythagoras himself — who stood watching ships disappear over the horizon who first raised questions about this intuition, and it was much, much later that the idea took currency in the West. Or, once one accepts the roundness of celestial bodies, take the problem of rolling out a ball of yarn along the Earth's equator and the Sun's circumference. If we now move out one foot from each surface and encircle them again, which ball do we need to add the most yarn to? For most people, the intuitive answer is, wrongly, the Sun: the linear relationship between circumference and radius means we add the same to each, 2π feet. Finally, let us visit another example of Lewis Carroll's [32]: imagine that a bag contains one counter, known to be either white or black; an additional white counter is then put in, the bag shaken, and a counter removed, which proves to be white; what is the chance that the remaining counter is also white? Most people's intuition tells them that the answer is, again wrongly, $1/2$, when the laws of probability prove the answer is $2/3$.

It may be argued that these problems only fool people unfamiliar with the domain. Experience in an area gives better intuition: people know about the globe "all around the world" (and use that very phrase without confusion or contradiction); and within mathematics, neither of the other two problems would be regarded as surprising. This does not, however, imply that experts are immune from the fallible nature of intuition. Even in mathematics, numerous are the proofs and theories that have been believed for years with unrecognised gaps or flaws.

1.3.1 Veridical Monsters

Quine distinguishes two types of paradox which occur in mathematics [152]: *falsidical* paradoxes where an argument seems to follow logically but the result is clearly false, such as the Euclidean isosceles proof given earlier; and *veridical* paradoxes where a true result appears intuitively false, such as the yarn example and the counter example[2]. The former are easily spotted, by definition, but the latter pose a much greater challenge.

In the 19th Century, mathematicians started to encounter such pathological veridical paradoxes that intuition — and even common sense — were utterly confounded, and mathematics faced a "crisis" [73]. Möbius's strip teased people's minds, and Klein's bottle made people infinitely drunk. Weierstrass demonstrated a function everywhere continuous but nowhere differentiable; Peano exposed a curve which could fill any bounded region to any arbitrary density; and Brouwer produced a map of three countries such that at every boundary point all three countries touched. As Poincaré recounts:

> Logic sometimes breeds monsters. For half a century there has been springing up a host of weird functions, which seem to strive to have as little resemblance as possible to honest functions that are of some use. No more continuity, or else continuity but no derivatives, etc...[3][144]

The parallel postulate leads us to one of the most influential veridical paradoxes. Since its incarnation in Euclid's Elements, most mathematicians were not only convinced of its truth, but many also felt that it could be proven from Euclid's other axioms. The endeavour to prove the existence of parallel lines lasted for almost two thousand years, only ending with the serious exploration of "non-Euclidean" geometries, by Gauss, Bolyai and Lobachevsky. Here, the familiar axiom admitting parallel

[2]Quine calls out a third category of paradoxes, *antinomy*, or self-contradictory statements, but any sound mathematical system by its nature avoids these.

[3]As his use of the word "honest" suggests, Poincaré did not attribute quite so much significance to these monsters as others did. The quote continues by mocking the crisis in mathematics attributed to an over-reliance on intuition: "Formerly, when a new function was invented, it was in view of some practical end. Today they are invented on purpose to show our ancestors' reasonings at fault, and we shall never get anything more out of them." However non-Euclidean geometries, discussed in the following section, are a rather more "honest" and practical domain, where intuition is exceptionally difficult.

lines is changed, and the result is not only logically consistent, it more accurately resembles our universe — at the fringes, as relativity is to Newton's physics — even if it is far less intuitive.

1.3.2 Logic and Hilbert's Program

Staying with Euclid, around 300 BC, we have one of the progenitors of the logical method: for most of modern history, his work has been regarded as the paragon of careful mathematical reasoning, but he makes a host of unjustified assumptions [128]. It is these omissions, where our intuition gave us false confidence and led to the "monstrous" confusion in the 19th Century; resolving this crisis, around non-Euclidean geometries in particular, demanded a logical method far stronger than had been theretofore employed.

One such endeavour, commencing in 1872, was the *Erlangen* program of Klein, Lie, and others, seeking a unifying group theoretical approach to geometry, based on *infinitesimal* rotations and translations of rigid bodies and appealing to our understanding of motion and space. But however attractive it seems, this intuitive appeal is a shaky foundation; in 1902, Hilbert pointed out *conceptual confusion* at its core [86], and promoted the formalist, axiomatic approach of Pasch, himself, and others.

The central tenet here, owing to Pasch, is that deduction should be independent from the meanings of the non-logical terms involved. No longer would imprecise or implicit definitions be allowed, and no longer would it be permitted to rely on intuition in the course of a proof. In 1882, he demonstrated this in a rigorous axiomatisation of projective geometry [140] now labelled as the "birthplace of modern axiomatics" [49].

Hilbert continued this effort in his seminal *Grundlagen der Geometrie* [85] of 1899. This supplied a set of axioms for Euclidean geometry which left nothing to chance or intuition: points, lines and planes were taken to be three undefined primitives and the relationships between them were characterised solely by the axioms. These axioms, for the first time, insisted on making explicit even the most obvious claims, such as that the geometry is not empty: "for every two points there exists one and only one line which contains them", and "there are at least three points not on the same line". And following the publication of the *Grundlagen* and the attack on the *Erlangen* program, Hilbert's research ambitions grew: he wanted not only a solid and complete logical foundation for geometry, but for all of mathematics. This began what was called *metamathematics*, and has since become known as Hilbert's Program, aiming to show that:

- mathematics follows from a suitable chosen finite system of axioms; and
- some such system of axioms is provably consistent.

1.3.3 Modern Logic and Incompleteness

At roughly the same time as the axiomatic method was being developed, the field of logic was also advancing. Boole, Pierce, and later Peano made significant progress, but the maturity of the field really came to bear with the publication of Frege's epic *Begriffschrift* in 1879 [59]. For the first time, a formal system was given in a definitive form and one could speak precisely about complex proofs and axiomatic systems in an unambiguous way. He later proceeded to give an axiomatisation of Cantor's set theory, intended as a solid foundation for mathematics. However, even with the use of logic, mistakes crept in. Zermelo, and Russell independently, discovered a paradox in Frege's work: he had permitted the existence of the set of all sets. This was repaired and refined in the axiomatic system of Zermelo-Fraenkel (ZF), the most common underpinning of mathematics today, but it reveals how major flaws can be overlooked by even the most scrupulous community of mathematicians and logicians.

The emphasis on rigour and logic led to the creation of *metalogic* — a means by which the properties of a logical system could be studied. As a consequence, another groundbreaking result was uncovered: Gödel's incompleteness theorem [65]. This states that under the assumption of the consistency of classical mathematics, there exist true propositions which are unprovable in the formal system of classical mathematics. In short, it is impossible to establish a set of axioms encompassing all of mathematics. With this result, "Gödel showed ... that the Hilbert Program was doomed" [37], and he revolutionised the world of mathematics, logic and philosophy. For nearly all mathematicians, his findings flew in the face of intuition. For many, it was demoralising, opening a crisis on the scale of that introduced by the monsters of the previous century. It was "a constant drain on the enthusiasm and determination with which I pursued my research work", according to Weyl [181]. But for others, it magnified the importance of the logical approach, stressing how vigilant they must be in light of new monsters whose existence had just been proven: an infinity of theorems which could never be proven.

1.4 The Nature of Proof

Hilbert's program and logic have given an incredibly powerful mechanism to guard against intuition's fallibility; but few would be so foolish as to believe intuition is rendered superfluous. Logic did not tell Hilbert, nor Euclid, how to choose their axioms, and a primary reason the axioms chosen by Euclid and of Hilbert remain in currency today is their intuitive appeal. As Gödel puts it, the axioms of a system should possess a feeling of familiarity such that they "force themselves upon us". This sense of their obvious correctness signals the fact that our belief in them has been generated by an intuition of mathematical reality. As Poincaré puts it:

> Thus logic and intuition have each their necessary role. Each is indispensable. Logic, which alone can give certainty, is the instrument of demonstration; intuition is the instrument of invention.

What engenders this intuition that forces an assumption upon us? Beauty, neatness, utility, elegance — these words have all been put forward as reasons to believe in something. Consider the Riemann Hypothesis (RH), a conjecture about the distribution of zeros in the Riemann zeta function. Despite its unproven status, it has been assumed in many proofs, and a major body of mathematics now relies upon it. It implies, among other things, that primes are distributed as regularly as possible, a result so profound the Riemann Hypothesis is now widely believed to be true. Not all mathematicians are so confident — Littlewood stated outright that he believes it to be false [107] — but the grand status of RH is reflected in its inclusion in both the Clay Mathematics Institute "Millennium Prize Problems", and Hilbert's list of 23 unsolved problems.

Furthermore, at a 2004 Royal Society meeting devoted to the nature of proof [161], there was overwhelming consensus that the value of a proof lies in its capacity for generating new mathematics, and further reinforcing known results. Both Gauss and Atiyah are famous for proving the same result in multiple different ways, with each proof giving different insights. Atiyah remarks that "any good theorem should have several proofs, the more the better" [153]. It was also noted at that meeting that "logical gaps" could be excused, if they were the scaffolding to support sufficiently interesting results: in contrast to the full axiomatic proof sought by Pasch and Hilbert, the ideal held aloft at this meeting is that of a proof which is succinct, beautiful, insightful and inspiring.

Another undecided problem which has had tremendous impact, and also from Hilbert's list, is the Continuum Hypothesis (CH). However, while it is generally expected that future mathematicians will resolve RH, the case of CH is closed, for the moment, with a verdict of “not proven”. In 1940, Gödel showed that it is consistent with the standard set theory axioms (ZF), but in 1963 Cohen showed that it is independent of those axioms. Gödel’s incompleteness theorem proved that there were undecidable statements in mathematics, and here, as if to mock the generations of mathematicians who had toiled on it, was an illustration of undeniable magnitude.

If mathematics is conceived as a perfect, logical edifice being pieced together from axioms, Gödel’s result removed a small padstone supporting it. It was generally believed that his undecidable assertions lay at the fringes of mathematics, and there remained an intuitive faith in the edifice itself. Cohen’s result shattered that faith. A mathematician never had any guarantee that she would discover the next proof, but she had worked secure in the belief that there was at least a proof to be found. This security was now gone. The axiomatic method cannot build mathematics. Until it is firmly put in place, any girder might prove undecidable; the very nature of proof is called into question. That question, as posed and answered by Kline, is:

> What then is mathematics if it is not a unique, rigorous, logical structure?
>
> It is a series of great intuitions carefully sifted and organised by the logic men are willing and able to apply at any time [102].

The only answer is that, while logic plays an important role in the sifting and sorting, it is not fundamentally what mathematics is about. There is a deeper, bigger, messier truth being sought. Weyl puts it more succinctly:

> My work always tried to unite the truth with the beautiful, but when I had to choose one or the other, I usually chose the beautiful [150].

And so we come back, not to a logical, rigorous housing block, but to Wiles’s mansion. And though we know from Gödel and Cohen that there may not be a light switch, we are drawn in by our love of beauty, and our curiosity. We are guided in that dark by intuition, and when we must we retrace our steps by logic, but we explore in hope that we find that switch, that next big intuitive leap, by which everything is illuminated.

Chapter 2

Maths through the Liquid Crystal

By the middle of the 20th Century, most mathematicians believed that the crisis surrounding the nature of proof had been settled: although there are undecidable assertions, even important ones, for the realist this detracts neither from the intuitive and logical value of the mathematics which is discoverable. Within the past thirty years, however, the power of the computer has opened fresh debates about what constitutes proof. Controversy centres around the role these machines can and should play in constructing the mansion of mathematics, reaching a climax with claims that the Four Colour Theorem [2] and, quite recently, Kepler's conjecture [75], have been *proven* with the aid of a computer.

In this chapter, we will begin by surveying two distinct branches of computer science developed to assist with mathematics: symbolic computation and automated reasoning. The former has led to computer algebra systems, and the latter to theorem proving systems. While both of these families of tools can be viewed as software which helps with formal symbolic manipulations, there is relatively little common ground between them. For the most part, they attract disparate communities and have achieved different ends. Several notable achievements for each will be described, along with an exposition of some of the most significant systems, their strengths and their weaknesses. We will then return to the contentious topic of computer-assisted proofs, pondering the question if it strays far away from any human comprehension, what value is there to *mathematics through the liquid crystal*?

2.1 Computer Algebra Systems

Computer algebra systems (CAS) were borne out of a desire to automate tedious and sometimes difficult algebraic manipulation tasks, which are ubiquitous in many scientific and engineering tasks. Due to the appeal of such a tool, it is not surprising that the first systems in the 1960s evolved out of two different sources; experimental physics and artificial intelligence[1]. The physicist and Nobel Prize laureate Martinus Veltman was a pioneer in the field. In 1963 he developed Schoonschip[2], one of the first computer algebra systems capable of performing symbolic manipulation [177]. This was soon followed by MATHLAB ("mathematical laboratory")[3], written in Lisp by Carl Engelman [50]. This system was motivated by the observation that computers were being used, at that time, for numerical computations, but were not present “during the most creative phases of the scientist’s labor”. Engelman wanted to build a system that would “provide the scientist with computational aid of a much more intimate and liberating nature”. MATHLAB was capable of numeric computation as well as a wide spectrum of symbolic computation, such as differentiation, integration, Laplace transforms and multiplication of matrices, to name but a few. Engelman produced a paper describing not only MATHLAB, but also laying out the criteria he believed were essential for a successful CAS. He did not lay out the range of mathematical operations which should be available, but instead wanted to capture the spirit and feel these systems should possess. He wrote:

1. It should be capable of ordinary numerical computation. This implies the ability to perform arithmetic, to compute functions or to look up their values in tables, and to draw graphs.
2. It should be capable of a wide spectrum of symbolic computations.
3. The user commands should be simple. MATHLAB is intended for a physicist, not a programmer. The commands should be no more complicated than the user’s thoughts. If he wishes to enter an equation into the computer, he should need only to type the equation in a notation like that of ordinary mathematics. If he should then wish to differentiate that equation with respect to x, he should have to give a command no more complicated than "differentiate (x)".
4. It must be expandable by the expert. The language, functions, and subroutines of the laboratory must be such that it will grow as an organism. If today we write programs for symbolic differentiation, we should expect, tomorrow, to employ them in programs for power series expansions. The opportunity to expand the

[1]The fields of computer algebra and artificial intelligence are now regarded as largely separate.

[2]Dutch for “clean ship”.

[3]MATHLAB should not be confused with MATLAB ("matrix laboratory") which is a system for numerical computation built at the University of New Mexico 15 years later.

programs should be open to anyone who masters a well-defined and common computer language.

5. It should be extensible by the user. While the ability of the physicist to augment the existing programs will no doubt be severely limited compared to that of the programming expert, he should be provided tools for doing certain simple things for himself, such as changing notational conventions or teaching the machine the derivatives of his favorite functions.
6. The computer, as viewed by the user, must be intimate and immediate. [...] Above all, the response time to the user's requests must be short.

2.1.1 The Roots of Computer Algebra Systems

The creation of CASs was preceded by some notable work in symbolic mathematics and computation. Worthy of mention is the software of Laning and Zierler from 1954 [106]. It was written for the MIT Whirlwind, and although its primary use was for numerical computation, it was one of the first systems operating algebraic compilers. This enabled it to accept mathematical formulae in algebraic notation; a user with no machine language experience could now write simple symbolic maths expressions and pass them to the system.

At around the same time, a system called FORTRAN was being developed at IBM. This was also investigating the problem of compiling algebraic notation. Some sources have claimed it was inspired by the work of Laning and Zierler, but the main developer Backus [11] has stated this to be a misconception, saying that "We were already considering algebraic input considerably more sophisticated than that of Laning and Zierler's system when we first heard of their pioneering work".

Another fundamental development in the adaptation of computers to symbolic computation took place in 1958. Inspired by Church's lambda calculus, McCarthy developed a language called LISP (for LISt Processor) [116], with the aim to provide a practical mathematical notation for computer programs. Steve Russel was the first to implement this language on an IBM 704, and it soon became the language of choice for most early computer algebra systems.

One of the first mathematical problems encoded in LISP was by Maling [111]. He wrote a program to perform symbolic differentiation and it was regarded as an early demonstration of the ability of LISP to handle advanced mathematical computation. Maling's work did suffer from a few weaknesses however; one being the restriction of the input/output to well formed LISP expressions. Another weakness was its procedures for simplifying expressions; whilst differentiation has exact rules — and so is easily encoded — the simplification of mathematical expressions is less well understood.

As well as differentiation, computer scientists also wrote programs for other areas of mathematics. The doctoral thesis of Slagle, under the supervision of Minsky, was one such endeavour [170]. He produced a program to tackle indefinite integration, using heuristics rather than an algorithm per say. His system contained a small table of integrals and when posed with a problem, it tried to reduce it to one of several found in that table, employing the same bag of tricks possessed by a good first year undergraduate.

This collection of research and development—focused on programs performing particular symbolic processes such as simplification, differentiation or integration—paved the way for integrated software tools which aimed to combine many powerful algorithms and make them easily accessible to a target audience.

2.1.2 State of the Art

At the time, MATHLAB enjoyed immense popularity, in part due to a mature collection of algorithms it could call upon, but also, we feel, in large measure due to Engelman's design principles listed earlier. There were several other endeavours contemporary to MATHLAB, but they did not share either its success or its focus on end-users.

The following two decades saw a flowering of numerous computer algebra systems which emulated MATHLAB's ambition, including muMATH, Reduce, Derive (based on muMATH), and Macsyma. Reduce and Maxima (a copyleft version of Macsyma) are actively maintained and used today. The current market leaders are mainly commercial: Maple, Mathematica, and MATLAB are commonly used by research mathematicians, scientists, and engineers.

The most popular CASs have vast and varied functionalities. Many have extensive support for numeric computations, to arbitrary precision, in addition to symbolic computations. Some of the most common mathematical operations available are:

- Simplifying algebraic expressions to the smallest possible expression or some standard form
- Factoring polynomials
- Solving systems of algebraic equations and inequalities
- Expanding products and powers
- Evaluating the limit of a function
- Differentiating algebraic equations (full and partial)

- Integrating algebraic equations (definite and indefinite)
- Computing matrix operations
- Rewriting trigonometric functions as exponentials

Typically, these operations can be carried out over various numeric domains. The reals, rationals, complex numbers, interval arithmetic, and algebraic number fields are commonly supported. Importantly, solutions can be obtained as exact values such as $\sqrt{5}$ or $\sqrt{2} \leq x < 5$ or 4π rather than the usual decimal values given by numeric methods of successive approximation. Many CASs also provide a high level programming language, which permits a user to extend the available mathematical operations by implementing the algorithm themselves.

Additionally, CASs often include the facility to plot graphs and parametric plots of functions in 2 or 3 dimensions, allowing the user to interactively explore or animate them if desired. This facility can be invaluable for refining or guiding one's intuition about a particular problem and has the potential to assist with the formation of new conjectures.

The user interfaces of today's popular CASs are fairly advanced. Not only do they produce plots and animations, but they also permit the immediate editing of mathematical expressions and provide a useful "document mode" for mathematicians to typeset their work — this can include special purpose style sheets, control of headers and footers, bracket matching, auto execution regions, command completion templates, syntax checking and auto-initialization regions.

2.1.3 Contributions and Noteworthy Results

Computer algebra systems have been a major success in many aspects of mathematics: they have been widely taught to students, used to perform routine calculations in proofs, assist with authoring of theories and — what Engelman envisaged — used for exploratoration and experimentation.

It is hard to measure quantitatively the extent to which CASs have been used by mathematicians in the exploration of theories and problems, as often it is only the polished end-product which is published in a journal, with the details of the journey omitted. However, it is clear that systems for symbolic computation and computer algebra have played significant roles in many of the biggest results in mathematics during the last fifty years.

Fermat's last theorem is perhaps the most famous modern achievement. Although Wiles does not rely on any computed results in his proof, a host of computer experiments preceeding his result gave renewed confidence in the truth of the theorem. For example, Brillhart *et al.* showed that given a, b, c, n if $n \nmid abc$, then the theorem holds for prime powers $n < 3 \cdot 10^9$ [23]; Wagstaff proved that Fermat's Last Theorem is generally true for all primes $n < 125,000$ [178]; while Buhler *et al.* raised this lower bound to four million [27]. This type of experimentation, made possible with computers and made easy with computer algebra systems, is rife in other areas, from finding the zeroes of the Riemann zeta function to discovering Mersenne primes.

Beyond mere experimentation, the development of intuition for elliptic curve groups — crucial to Wiles's proof — is enormously facilitated by CASs. Figure 2.1 shows a screenshot of a dynamic visualisation in Mathematica where these groups can be interactively explored[4]. While we are not claiming that Fermat's Last Theorem would not have been proven without these tools, it is indisputable that a paper-and-pencil exploration that might have taken days can now be performed in minutes. Furthermore, by making it so easy to explore such problems, these tools make the domain more accessible to a wider audience, increasing the pool of people who can work in an area and accelerating the rate at which they contribute to further development.

In addition to facilitating the exploration of these groups, computer algebra systems played a vital role in another recent mathematical milestone. The "enormous theorem" — the recently-completed classification of finite simple groups — has made heavy use of results from computer algebra systems [5]. Here, CASs were used to prove the existence and/or the uniqueness of some sporadic groups, automating what would otherwise be a labourious and painstaking task. However, in an attempt to make the proof completely human-generated, these computer parts are steadily being eliminated.

Our final example is the only one of the Millenium Prize Problems to have yet been solved: Poincaré's Conjecture, proposed in 1903, states that every simply-connected 3-manifold without a boundary is homeomorphic to a 3-sphere. The proof is due to Perelman, completed in 2003, and while it does not rely on computers, CAS-generated visualisations of Ricci flows were an important step along the way [142].

[4]This demonstration is freely available on the web from `http://demonstrations.wolfram.com/AdditionOfPointsOnAnEllipticCurveOverTheReals/`

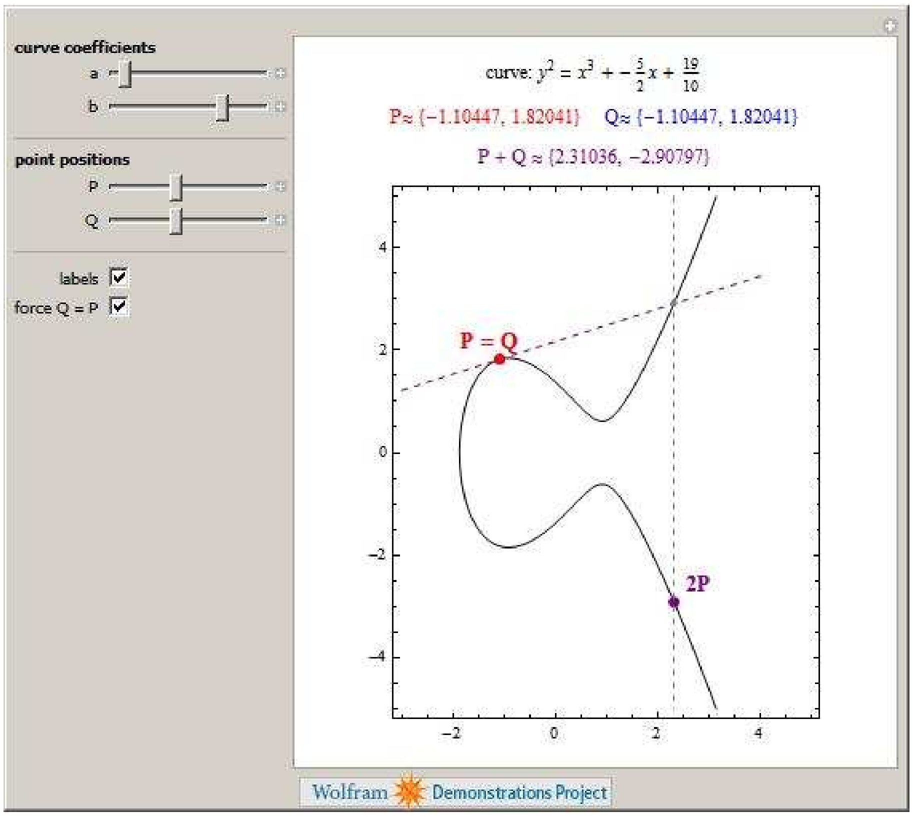


FIGURE 2.1: Exploring Elliptic Curve Groups: a screenshot of a dymanic visualisation from Mathematica.

2.2 Theorem Proving Systems

The 1950s and 1960s also saw the development of another type of mathematical assistant, namely theorem provers. These tools emerged from research into automated reasoning and unlike CASs — which were invented to help humans rapidly explore and solve mathematical problems — theorem provers were built to automatically discover formal proofs of theorems. As input, theorem provers take logical formulae which express the definitions, axioms and conjectures of a theory. A mechanical search is then carried out, looking for derivations which justify the conjectures from the axioms and the rules of the encoded logic. Theoretical mathematics is clearly one application, but these tools can, and are, employed in many other disciplines which can benefit from provably correct results. The verification and synthesis of both hardware and software are prime examples, as these domains are perfect for mechanisation due to the fact they can be given correct axiomatisations.

Undoubtedly, the development of logic and the formalisation of mathematics, as mentioned in Section 1.3.2, was a precursor to the creation of these tools, and the idea of reducing reasoning to mechanical calculation, in a manner similar to arithmetic, can trace its origins back to Hobbes in the 17th Century [88]:

> Reason [. . .] is nothing but reckoning. For as Arithmeticians teach to adde and subtract in numbers [. . .] The Logicians teach the same in consequences of words [. . .] And as Arithmetique, unpractised men must, and Professors themselves may often erre, and cast up false; so also in any other subject of Reasoning the ablest, most attentive, and most practised men, may deceive themselves, and inferre false conclusions.

Creating tools which produce formally correct proofs was motivation enough for many of the developers. However, there were some who had another driving force behind creating these tools. For this community, the main goal was to explore the “higher” faculties of human reasoning through experimenting with computer programs which attempted to emulate how humans discover mathematical proofs. In fact, this was the impetus for the the first theorem proving system, the Logic Theory Machine. It was invented by artificial intelligence pioneers Newell, Simon and Shaw in 1955 [62]. Five basic axioms of propositional logic, given in chapter one of Russell and Whitehead’s *Principia Mathematica*, were supplied to their system, together with three rules of inference: substitution, replacement and detachment. Proofs of theorems from the Principia were then searched for. If no immediate one-step proof could be found, a set of subgoals were generated and proofs of these were looked for, and so on iteratively. Newell and Simon realized that the search tree would grow exponentially and that they needed to "trim" some branches, using "rules of thumb" to determine which pathways were unlikely to lead to a solution. They called these ad-hoc rules "heuristics", using a term introduced in George Polya’s book *How to Solve It* [146]. The Logic Theory Machine proved 38 of the 52 theorems it was presented with and even found some proofs which were more elegant than the written versions from the Principia. The system set the stage for nearly all approaches since.

The Geometry Machine [63], developed by Gelernter in 1959 at the IBM Research Center in New York, was the Logic Theory Machine’s immediate successor. Similarly, it relied on notions of Euclidean deductive geometry. A backward chaining strategy was adopted and the use of semantic guidance was employed by using a diagram to restrict the search of a proof. Branches in the search space that were false in the diagram were not further explored since they could not lead to a proof. The Geometry Machine was an early AI success and the basic ideas have been generalised to the algebraic method of *semantic resolution* and appear to a greater or lesser extent, in many other systems. In particular, Goldstein’s Basic Theorem Prover is essentially an extension of the Geometry Machine. Gilmore, Nevins and Elcock also did more work using the same approach, with additions such as forward chaining. It is interesting to note that model-checking provers based on diagrams do not seem to have moved on significantly from the Geometry Machine.

The endeavour to emulate human reasoning inside a theorem prover was commendable, but it faced many challenges. One of the biggest challenges developers had to contend with, was the fact that only vague and anecdotal information was available on how mathematicians work. In contrast to these AI approaches, many believed that in designing a machine it was better to replace heuristic methods with algorithmic ones and indeed, the community who favoured the pure logic techniques and decision procedures achieved much more impressive results in the early days. Wos, one of the most successful practitioners of AR, attributes the success of his research group in no small measure to the fact that they adopted a machine oriented approach and Wang himself remarked that his simple, systematic program for the AE calculus[5] was dramatically more effective than Newell and Simon's. Prawitz was another researcher interested in mechanical theorem proving which played to a computer's strength. His work was inspired by results in logic, namely Gentzen's cut-free sequent calculus. Other notable researchers working on the machine-oriented algorithms were Davis, Putnam, Logemann and Loveland who produced the DPLL algorithm; a complete algorithm for deciding the satisfiability of propositional logic formulae in conjunctive normal form. Even after 50 years, this algorithm still forms the basis for most of the current satisfiability solvers (SAT solvers), as well as for many theorem provers which deal with fragments of first-order logic [163].

Today, there is still a preponderance of research on the machine-oriented side, but there have been notable results based on human-oriented approaches too. Some of the most successful contemporary tools commonly have an integration of these approaches. We will decribe these in Section 2.2.2.

2.2.1 Reality Check

In the early days, ambitions for theorem proving systems were high, with Simon famously predicting in 1958 that within 10 years an important mathematical theorem would be proved by a computer. This optimism stemmed from successes of early pro-

[5]The AE Calculus is a decidable fragment of first order logic which contains all the formulae which can be put into prenex normal form so that the initial string of quantifiers is either empty, single, or if multiple, consists of a sequence of universals followed by a sequence of existentials.

grams such as the Logic Theorist and the Geometry Machine, and although it was recognised that these systems only dealt with microworlds which contained few objects, it was generally thought that scaling up to larger problems was simply a matter of faster hardware and larger memories. In practical terms, they had expected the anecdotally exponential increases in computing power (the so-called Moore's law) to allow them to conquer sophisticated theorems; they had not appreciated that for many mathematical problems (*e.g.* in domains such as Presburger arithmetic), the computational complexity is super-exponential[6]. The failure of these systems to produce the promised ground-breaking mathematical results is one of the causes attributed to the *AI winter*.

With the benefit of hindsight, the early optimism seems naive, particularly in view of major results in theoretical logic which were discovered in the decades just prior. In addition to Gödel's incompleteness theorem (see Section 1.3.3), related results were achieved in computation theory. The Church-Turing hypothesis implies that there is no algorithm for deciding if a statement is true in arithmetic or any other sufficiently complex formal system, and Tarski's undefinability theorem shows that there is no mathematical formula capable of determining such truth [163]. It is important to recognize, however, that while these results limit what can be done in mathematics — whether by a human or a computer — they do not imply anything about the complexity of those problems which are solvable. In fact, other results in theoretical logic supported the belief that a computer could mechanically reason about interesting and complex mathematics; some important classes of problems had already been shown to be decidable, such as Presburger arithmetic and the first order theory of real closed fields, and it was accepted that the vast majority of conventional mathematics could be described by a finite set of schemas, such as ZF, where there is at least a semi-decision procedure that can in principle verify any logical consequence of those axioms. What the early researchers failed to grasp, however, was that these results also imply nothing about complexity. The fact that a program can find a solution in principle does not mean that the program contains any of the mechanisms needed to find it in practice. It was only after years of research into theorem proving that it was recognized just how enormous the gap between decidability in theory as opposed to in practice was; even gigabytes of memory and quadrillions of compute cycles is often insufficient.

Clearly, cutting down the search space is key to a successful tool, and whilst early researchers recognised this, such as Newell and Simon with their heuristics, the dif-

[6]The theory of NP-completeness had not yet been developed.

ficulty of automating this effectively is now better appreciated; the search space for decidable theories can lead to a combinatorial explosion and undeciable theories (such as Peano arithmetic) have the added difficulty of needing effective heuristics, which can be hard — even impossible — to discover. Whilst fully automated provers are still actively researched, another type of theorem prover has emerged in recent years. Interactive (or semi-automated) theorem provers have been developed to work with human guidance. At the very least these tools attempt to verify a proof found by a human by checking the correctness of the argument and guarding against typical human errors such as implicit assumptions and forgotten special cases. The grander aim is that these tools should help the proof process substantially by automating certain parts and providing advice to the user to aid the proof discovery. Interactive theorem proving alleviates the problems of semi-decidability and large search spaces to some extent, because the human can use their intuition as to how the proof should proceed and also has the option to cancel any query and try another approach if the query is taking too much time.

The concept of a human and a machine cooperating with one another goes back to the SAM (Semi-Automated Mathematics) system from 1966 [163]. It was a notable success, being used to construct a proof of a hitherto unproven conjecture in lattice theory. Not long after the SAM project, the three first-generation interactive theorem provers, as they came to be known, emerged: Automath, Mizar, and LCF. Many of the most successful interactive theorem provers around today are based heavily on at least one of these efforts [6], as we shall see in the next section.

2.2.2 State of the Art

The most popular interactive theorem provers today are HOL [30], HOL Light [82], Isabelle [134], Coq [42], Mizar [162], PVS [171] and ACL2 [98]. These systems vary on several fronts, which we shall briefly look at in this section: their underlying logical frameworks and implementation strategies, their modes of interaction, and their automation.[7]

[7]The reader wanting a more detailed understanding of these and others is referred to the survey by Wiedijk [184], which includes a comparative commentary on how these systems prove the irrationality of $\sqrt{2}$.

2.2.2.1 Logical Frameworks

Theorem provers have an underlying logic which specifies the language in which assertions are to be expressed and the admissible rules of inference. Zermelo-Fraenkel set theory (ZF) (Section 1.3.3), a common foundational framework for mathematics, is one option. Mizar, mentioned previously, builds on a variant of ZF known as Tarski-Grothendieck set theory.

Other options are possible, however, and have proven very popular in practice. Several descendants of LCF — including HOL, HOL Light, and Isabelle/HOL — use a formulation of higher-order logic (HOL) in Church's simple type theory [35], in which every term is assigned a type. Even within these three, Isabelle's foundations are quite distinct: the inference rules are theorems in a metalogic, which means the system can be made generic, *i.e.* to work for a family of object logics, including ZF and first-order logic (FOL) in addition to HOL. The ability to express assertions in higher order logic is advantageous as it permits more natural mathematical statements to be made due to the fact predicates and functions can be quantified.

Some other descendants of the LCF approach, such as Coq, have taken a radically different foundation based upon dependently typed constructive logic. Here, the type system is capable of carrying more information; for instance, the `tl` function giving the tail of a list could have a *type* which maps from a list of length n (for arbitrary $n > 0$) to a list of length $n-1$. In contrast, in the Isabelle definition the type of this function is `list ⇒ list`, and a lemma is introduced to provide the fact that the length of the list reduces by one. This different approach makes some concepts easier to work with in proofs, as more work is done in defining the types, but it comes at a cost in the complexity of the type system.

When it comes to formalising much of mathematics, however, the differences between the logical foundations are negligible. The two popular systems Isabelle and Coq both have libraries for a wide range of common mathematics, even though the former is simply typed and the latter dependently typed. However the choice of foundation does impact the proof process, making a big difference in the way things are expressed and how and when assertions are proved.

Whilst contemporary theorem provers have different logical frameworks, they all share one critical design principle: the logical framework is always described in a small kernel at the heart of the prover. Everything constructed within the prover boils down to the minimal set of axioms and logical inference rules within this kernel. As this core is easily inspected, and many reviewers have satisfied themselves of its correctness, every proof which is validated by the kernel inherits a high degree of confidence. It may be the case that the prover has impenetrable and perhaps erroneous implementations for the user interface and procedures such as search control, but these will not jeopardise the correctness of the proofs, as they must pass through the trusted kernel. For a good account of how to believe a machine checked proof, see the book chapter by Pollack [145].

2.2.2.2 Interaction

In addition to having an assertion language, which allows users to state definitions and lemmas, interactive theorem provers also require a proof language. This specifies how to represent a mathematical argument inside the system and provides the user with the means to communicate which rules and techniques should be applied to the current goals. Most contemporary provers support a *procedural* style of interaction, where a completed proof is merely input “code" giving the list of instructions which tell the proof assistant how to act on the evolving proof states; if one wants to understand the proof they need to replay it within the system. There are a few exceptions, however. Mizar, and more recently an extension to Isabelle called Isar, support a *declarative* proof language, which makes intermediate goals explicit. These languages have the benefit that when a proof is complete it can stand alone from the theorem prover. For some situations this is necessary. However, for many people, proof exploration and efficiency take priority, and the procedural mode of interaction seems better suited for these ends. Whether supporting procedural or declarative proofs, an orthogonal consideration is the ability of the language — and the system — to allow a hierarchical proof representation where intermediate subgoals or steps can be expressed even when there is not a formal logical chain leading to them. This is a central tenet in an area of theorem proving called proof planning [28].

Communication with interactive theorem provers is achieved through user interfaces (UI), which today come in a wide variety. Proof assistants typically have a command line mode, but many have a more advanced graphical UI too. Some of the graphical UIs support extended characters for mathematical syntax, and allow users to turn on and off the type information attached to the variables. Many also provide point-and-click buttons for commands such as processing or halting proof commands. It is common that a different team of developers will build the UI environment, and it usually sits separately from the prover. A notable UI success has been the generic, emacs based Proof General [7], which provides a powerful and configurable front-end for proof assistants. It has been used with Isabelle and Coq for many years and can be easily customized to work with any theorem prover.

2.2.2.3 Automation

The touch points where the user interacts with the theorem prover vary enormously depending on the amount of automation for a given domain. Early interactive provers required extravagant low level details of proofs to be spelled out by the user. Semi-automatic systems attempt to engage the user at a higher level. For the classes of decidable theories, it is typical to have decision procedures encoded. Common examples are procedures for linear arithmetic, tautology checking, SAT solving and model elimination. For undecidable problems, many proof assistants attempt to encode search procedures, which can be heuristically driven and rely on domain specific knowledge.

Because theorem provers permit new theories to be developed, an important aspect of their interactivity is the extent to which users can extend the automation. This can be done through various techniques, including programmatic tactics or pattern matching rewrite rules.

Another way to increase the automation of a proof assistant is to enable it to call out to external tools, such as other provers or computer algebra systems. In particular, the integration with contemporary automated provers has been popular, as after fifty years of development automated theorem provers have reached a fairly mature state. Some of the most advanced automated systems today are Vampire [154], SPASS [179] and E [166], all of which have been linked to Isabelle in recent years [20]. There have been two techniques for integrating tools, namely the oracle approach and the proof reconstruction method. The latter is usually adopted when a proof certificate can be returned from the external tool, enabling the proof to be reconstructed in the theorem prover. This gives the highest level of confidence in the results. Unfortunately, this is not always possible. In these situations, the oracle mode is adopted — where the result is taken on trust. In these cases the validity of the result is only as good as the correctness of the external tool and the translations between the systems. A more detailed description of tool integration techniques and past implementations will be given in Section 6.3.

2.2.3 Motivations and Previous Contributions

One motivation for building theorem provers is to find mathematical proofs automatically. However, as the previous section described, it's often not feasible for a computer to automatically generate a proof in full. There are a number of other related, more modest objectives which motivate researchers. These motivations along with some noteworthy results are summarised below.

Automatic Proof Discovery of Open Conjectures

One of the most famous results produced by an automatic theorem prover was achieved in 1996 by McCune of the Argonne National Laboratory. He used the EQP system to find a proof of the longstanding Robbins Conjecture which states that every Robbins Algebra is boolean [117]. The result was reported worldwide and was heralded as a great success as the conjecture had resisted human proof for over 60 years.

Obtaining Rigour

Interactive theorem provers have the potential to construct fully formal proofs of serious mathematical results, all with a high degree of correctness. Notable theorems which have thus far been proven are Gödel's Incompleteness Theorem (Shankar [168], 1986, in the Boyer-Moore theorem prover), the Four Colour Theorem (Gonthier [66], 2004, in Coq), the Prime Number Theorem (Avigad [10], 2005, in Isabelle), Dirchlet's Theorem on primes in arithmetic progression (Harrison [80], 2009, in HOL Light), and the Jordan Curve Theorem (Hales [76], 2005, in HOL Light). Despite these theorems having previously published human proofs (with the exception of the Four Colour Theorem), their formalisation gave the benefit of a more rigorous argument which had the potential to unearth any ambiguous definitions or erroneous proof steps which may have previously gone unnoticed.

Indeed, Fleuriot's formalisation of Newton's Principia brought to light an error in Newton's reasoning [56]. Using Isabelle, he showed that Newton had made the mistake of cancelling an infinitesimal quantity on either side of an equation, an error that Newton himself had elsewhere explicitly avoided but had unwittingly made on this occasion. The proof was easily repaired by Fleuriot and for this reason some may say the discovery was not significant. However, what is remarkable is that three centuries of analysis on one of the world's oldest and well-known mathematics books had failed to uncover this mistake.

Even the staunch formalist Hilbert was not immune. In prior work, we formalised much of Hilbert's *Grundlagen*, and found that many of his proofs had missing steps [119]. Although the lacunae would normally be unremarkable — as the missing steps are trivial to fill in for one familiar with the domain — they are noteworthy considering Hilbert's ambition. As described in Section 1.3.2, he intended that no intuition should be needed in verifying his proofs. To fill in the gaps we found, intuition was certainly required. For one particular case, Hilbert's theorem three, five proof steps were missing from his written proof, three of which were non-trivial for us to find. This highlights that fully formal proofs are extraordinarily difficult for a human, even a mathematician as brilliant and rigour-obsessed as Hilbert. Theorem provers help keep our mathematical arguments as rigorous as can be.

Validate a Proof Generated by an Ad-hoc Computer Program

In Section 2.4 we will describe two mathematical proofs which turned out to be highly controversial: Appel and Haken's proof of the Four Colour Theorem [2] and Hales's proof of the Kepler conjecture [75]. These proofs were heavily critised as they were derived from ad-hoc computer programs which could not be easily inspected. To obtain assurances about these programs, theorem provers could be employed to verify that the software is correct.

Include a Wider Audience

Some mathematicians receive thousands of letters from amateur/hobbyist mathematicians, who claim to have discovered a proof to important open conjectures in the field. Ramanujan was one lucky enough to be noticed, but in most cases, such correspondence is ignored by the establishment. Theorem provers permit amateur mathematicians to explore mathematics and have their findings taken more seriously by the mathematical community.

The tools have enormous potential to be used in collaborative projects too, where a formalised theory can be held on-line whilst in development. Anyone interested in the theory can view the up-to-date file and contribute if they are able to progress the work. This is in keeping with the ethos of the 2009 Polymath Project [147] which was instigated by Gowers and Nielsen. Inspired by open-source enterprises such as Linux and Wikipedia, they decided to use blogs and a wiki to mediate a fully open collaboration with other mathematicians. The aim of the project was to find an elementary proof of a special case of the density Hales-Jewett theorem, and the collaboration achieved far more than Gowers expected. An article in Nature described the project as "showcasing what may turn out to be a powerful force in scientific discovery, the collaboration of many minds through the Internet". In the future, theorem provers could play an important role in these types of projects.

Verify Algorithms, Software and Hardware

The three cornerstones to computer science — algorithms, software and hardware — are all areas where proof is not a tradition. However, they are all areas where boundary cases can easily be omitted. Making a mistake in any aspect of their development can be catastrophic, costing time and money in industrial applications or even loss of life when they are employed in mission critical situations. Theorem provers have had success in formally verifying many developments in this field, from systems security to compilers. Intel and NASA even have their own teams dedicated to investigating the formal correctness of many of their projects. As an example, at Intel Harrison famously used HOL-Light to prove the correctness of floating point arithmetic [81]. ACL2 has also had success in this discipline, being used to prove the correctness of the floating point division operations of the AMD K5 microprocessor in the wake of the Pentium FDIV bug.

Perform Huge Proofs

There are many important theorems which may only ever be proved by analysing a gigantic number of possible cases. The Four Colour Theorem and Kepler's conjecture may turn out to be of this class. If so, computerised proof may be the only way to reason about these problems. Theorem provers in particular can — and are being used to — provide reassurances that these proofs are correct.

2.3 Contrasting TPs and CASs

Theorem provers and computer algebra systems have, for the most part, been adopted by very different communities. While CASs have become mainstream tools in mathematics (and various other disciplines such as physics and engineering), interest in TPs has been much less extensive, with their target community mainly being confined to logicians and computer scientists interested in formal correctness proofs. The popularity of CASs can be attributed to their ease of use and expediency at solving problems automatically, making them attractive for quickly exploring problem domains. This contrasts greatly with TPs, which require a user to be highly skilled. Even for one well aquainted with TPs, interacting with them can be a time consuming task. This acts as a barrier to their wider adoption.

Despite TPs not being as easy to use as CASs, they do have two clear advantages — they provide clear sematics and logical rigour. The following sections will describe these aspects in further detail.

2.3.1 Usability

CASs are normally easier to use than TPs for several reasons: they commonly have a polished UI, a language which comes close to the structure of ordinary mathematical language, the functionality to easily undo and redo changes made and a development environment which does not crash when a long running proces is cancelled.

The UI is an important consideration if a tool is to be usable and CASs typically have a frontend which is like an IDE for software development or a word processor, providing the user with context sensitive help. Whilst some TPs have graphical UIs they have not reached such a mature state.

Providing the functionality for the user to write mathematical statements as they typically would be written is hugely appealing. CASs such as Maple have embraced this mode of interaction, in keeping with Engelman's early vision that a CAS should have simple commands. As an example, consider expressing a definite integral. In Maple we can use the familiar notation:

$$\int_a^b x^{n-1}\,dx$$

By contrast, in TPs it is often non-trivial to express high level mathematics. The same integration problem in Isabelle has to be stated as:

```
Integral {a..b} (%x. x powr ((n::real) - 1))
```

Note that Isabelle's built in libraries currently only permit definite integration to be expressed. This is due to the complexities of formalising the intuitive concept of *plus C* (where *C* is a constant), which is required for correctly reasoning about indefinite integration.

Despite the mathematical language of a TP not being as easy to write as that of a CAS, there is one benefit of such a cumbersome logical language — it permits the expression of far more sophisticated mathematical concepts such as epsilon-delta definitions of continuity.

Many users of mathematical assistants also place a premium on efficient automation. CASs are faster and better than TPs at solving many problems automatically. This is because CASs implement decision procedures in a low level computer language (close to machine code), whereas formal tools tend to use tactics which have to be interpreted and tracked in the kernel. This makes TPs much slower. There are however, some problems which CASs fail to solve. For these problems, a more configurable implementation might succeed. This is an area where interactive TPs have the advantage as the user is able to guide the system to find a solution.

2.3.2 Clear Semantics

Frequently, there are expressions in mathematics which can be ambiguous. Mathematicians have conventions which allow them to know how expressions should be interpreted and how to supply just enough content to make it clear for their intended readers. For example, when reasoning about integration, a mathematician will know if the Riemann, Lebesgue or Gauge integral is being referred to — or if it matters. With polynomial expressions, such as $x^3 + 7x + 5$, the mathematician will be aware which field the variable x belongs to (the reals, complexes, or something else).

In contrast, computer algebra systems cannot deduce this context information. They make certain default assumptions and implementation decisions which a user is expected to acquaint themselves with. Unfortunately, the underlying semantics of some expressions are unclear (or unexpected) in many of these systems. One potential confusion in CASs can be which branch of various complex functions such as square root, logarithm and power is considered. Knowing which field the CAS has assumed to be working in can also be a source of confusion. For example, if Maple is told to assume that $x^2 < 1$ and then asked whether $|x| < 1$, it says `false` — it is including complex numbers. Without any context, however, one might assume the question is being asked over the reals; and such a minor error risks invalidating an entire computation.

Theorem provers take a very different approach, where a precisely defined logical basis must be established and made explicit. Other mathematical concepts and theories tend to be constructed as conservative extensions of this. For any statement, it is mandatory to specify the theories that are being built upon, so the TP can always unambiguously determine the semantics of any expression. As an example Isabelle has a small set of axioms for logic and set theory, and all its other concepts are derived or defined from this set. All its theories, from lists and natural numbers to vectors and integration, declare at the outset which parent theories are imported, and this determines what expressions can be written and how they will be interpreted.

2.3.3 Logical Rigour

Even when a CAS can be relied upon to give a result that admits a precise mathematical interpretation, the user can not rely on the answers always being correct. It is well known that CASs are notoriously unsound, particularly when singularities and other irregularities at the boundaries of domains have to be taken into account.

Integration is one typical domain where the user has to be careful. Many modern computer algebra systems use an implementation of the Risch-Norman algorithm [156] as the basis of their integration routines. Due to its programmatic intricacy, some systems rely on a large number of recognised solutions (a lookup table), with a small number of methods for converting a presented problem to one of these forms. Where the Risch algorithm is used, the systems are insensitive to boundary conditions and occasionally return false statements. For example, evaluating the previous integration in Maple gives:

$$\int x^{n-1}\,dx = \frac{x^n}{n}$$

Despite the problem being easy to express in the system, it has given a wrong result; there should be the addition of an arbitrary constant appearing in the solution (although Maple warns us of this omission in the user manual). More seriously however, this result is true so long as $n \neq 0$; a correct answer must include the special case:

$$\int x^{-1}\,dx = \ln x + C$$

A system would fail if it relied on that general solution and instantiated it whenever $n = 0$. While many people will recognise the problem in this case, there are other areas where similar convenient sloppiness is applied. And if the tools are relied upon in real-world domains, an error such as this could be severe — in structural engineering or space flight.

By contrast, TPs have the ability to reason with due consideration to boundary conditions. They take considerable care that all alleged *theorems* are deduced in a rigorous way, and all conditions made explicit. Indeed, as mentioned previously, many construct a complete proof using a very simple kernel of primitive inference rules.

Although nothing is ever completely certain, a theorem in such a system is very likely to be correct. TPs inherit Hilbert's mantle, valuing soundness above all, whereas CASs continue the pragmatism of the Erlangen school, more interested in results than they are afraid of the monsters of Section 1.3.1.

2.4 The Changing Nature of Proof

The preceeding sections have focussed on the two types of tools for computerised mathematics; computer algebra systems and theorem provers. However, these systems do not completely encapsulate the computer's role in aiding mathematics. Two of the most notable — and controversial — proofs of the digital era have come from ad-hoc computer programs. The long standing Four Colour Theorem was the first to be proved using a computer in this way. The theorem was first conjectured in 1852 by Guthrie, and it stated that any planar map can be coloured with at most four colours in a way that no two regions with the same colour share a border. For over a century many famous mathematicians — including De Morgan, Peirce, Hamilton, Cayley, Birkhoff, and Lebesgue — worked in vain to prove the conjecture. It was not until 1976 that Appel and Haken announced that they had resolved the problem [2]. Their work was ground breaking as they had used a computer to carry out a gigantic case analysis which could not be carried out by hand. Despite this, however, the majority of mathematicians were critical of the proof; the ostensible objection was that the computer program which Appel and Haken wrote (in IBM 370 assembly language) was difficult to verify and therefore potentially erroneous. A simpler computer proof was later produced by Robertson, Sanders, Seymour and Thomas [158], but it was not humanly verifiable either, thus receiving the same criticisms.

Controversy over computer generated proofs recently flared again with the announcement by Hales that he had finally proven the long standing Kepler's conjecture [77]. This conjecure was first posited in 1611 by Kepler, who believed the most efficient way to pack spheres in a box is the way grocers usually pack oranges — in a face-centred cubic lattice arrangement whereby each layer of oranges is shifted so that an orange touches four oranges in the layer below. It resisted efforts of proof for centuries and as a result Hilbert made it one of his 23 most difficult and fundamental questions in mathematics. In 1998, Hales claimed he had a solution to the conjecture which used a clever trick to simplify the problem; rather than reason about an infinite number of spheres in a infinite space, Hales reduced the problem to be one about a finite, but very large, number of mathematical objects. He then used the computer to prove bounds about these objects (around 100,000 cases had to be considered). Despite Hales' conviction that the proof was correct, it took 7 years for it to be accepted by the Annals of Mathematics, the field's most prestigious journal. Even then it was accompanied by an unusual disclaimer stating that the computer programs accompanying the paper have not undergone peer review. This was not for want of trying, however. Many man years were spent attempting to understand the computer-assisted parts of the proof but the reviewers had to eventually conclude that it was just too impenetrable[8].

Dissatisfied with his proof being published with a disclaimer, Hales responded in 2003 by instigating an ambitious, collaborative project, called Flyspeck [57], with the aim of demonstrating Kepler's conjecture formally using contemporary theorem provers. The proof will thus be constructed out of transparent logical steps instead of obscure computer code, giving it a high level of correctness. As of this writing, ten years later, the project is ongoing but very near completion. Hales at one point estimated that 20 person-years of work would be required, and given the widespread and enthusiastic response to his proposal, and involvement of many researchers, this estimate seems not far off. To date HOL Light, Coq, and Isabelle have been used to prove many of the fundamental results the proof will rely upon, including the Jordan curve theorem and many properties relating to tame planar graphs.

Interestingly, the endeavour to prove the Four Colour Theorem has also followed this route, with Appel-Haken's original proof verified in Coq in 2004 by Gonthier [66].

The response to these Herculean efforts in mathematical circles, however, has been mostly in the range of disinterest to disdain. One critic commented, "a good mathematical proof is like a poem — this is a telephone directory!" [55], in response to the Appel-Haken proof, a sentiment even more applicable to proofs in TPs. The topic became a focus of discussion at the Royal Society meeting referred to in Section 1.4. Cohen stated a view that the essence of mathematics is to generate new insights and new techniques, and commented that these are absent from mechanised proofs. Atiyah went further, declaring that "the aim of mathematics is to explain as much as possible in simple terms", and predicting that as long as computers are what they are now, a human-made proof will always be superior to a computer-assisted one [161].

The changing nature of proof brings us back to one of the points made in Chapter 1, that intuition is crucial to the understanding of mathematics, from an initial discovery through to its final presentation. We have seen in this chapter that CASs aid this im-

[8]It is interesting to note that the board of reviewers did not have any computer scientists.

mensely, by accelerating solutions to many kinds of problem solving and facilitating exploration and insight into certain domains. But from Aristotle and Pythagoras to Hilbert and even Thurston, few, if any, mathematicians have denied the importance of logical justification. Surely we have not met the last of Poincaré's monsters! TPs provide the most extensive logical reasoning capabilities the world has ever known, and the resulting proofs, however lengthy and obscure to humans, are the pinnacle of rigour. Yet these systems remain relatively neglected among mathematicians, unlike their CAS cousins, and even the most active area of TP activity — verifying software and hardware — engages a mere fraction of those who could conceivably benefit. This chapter has explored some of the stated reasons for this, boiling down to their difficulty of use, caused in large part by the very length and obscurity of the proofs. If this obstacle could be overcome, and the role of intuition and beauty celebrated in the formal proof process, the potential to transform mathematical and even scientific knowledge stands in waiting.

Interlude

Having introduced the spectrum of rigour in the context of mathematical proof, let us pause for reflection and an overview of what is to proceed in the core of this thesis.

As Chapter 2 described, the digital era has engendered some of the most contentious mathematical proofs ever. It has made possible the rigour desired for so long, but the resulting proofs, and the tools that produce them, have found a largely apathetic audience amongst mathematicians. Our research begins by exploring this contradiction, asking what would be necessary to increase the appeal of TP systems to a wider community.

To answer this, we embarked on a case study to afford us greater insight into the strengths of these tools — interactive theorem provers in particular — and the weaknesses which render them so unappealing. We start in Chapter 3 by introducing our case study and the tools and formal theories we use, specifically the interactive TP Isabelle and its formulation of Hoare logic. Our case study looks at formally verifying the Graham's Scan algorithm for finding convex hulls; the algorithm and our proof are presented in Chapter 4. In Chapter 5 we collect observations on the use of TPs, noting two major difficulties, that they can require an inordinate amount of time on trivialities, and that they can get in the way of a user's all-important intuition. Despite these barriers, however, we observe how the collective understanding of a theorem or algorithm can ultimately be improved by a rigourous mechanical proof.

In Chapter 6 we turn our attention to looking at ways in which the proof process could be made easier, giving a survey of several tools and techniques which could potentially improve the user's experience; these range from extending Isabelle's simplifier and classical reasoner to examining techniques from the field of mechanical geometry theorem proving, such as the Area Method and Cylindrical Algebraic Decomposition. The benefits afforded by combining CASs and TPs are then discussed and the ways in which these tools have been combined in the past described. The chapter concludes by introducing the emerging paradigm of Proof Engineering, which looks at how to engineer theorem provers to best suit the users.

In Chapter 7 we hypothesise that the formal proof process is improved by giving the user a suite of tightly integrated tools within a single development environment where the translations happen seamlessly and support for both novices and experts is provided. Despite these ideas being common in the literature on usability and standard fare in modern software engineering tools, they are often neglected in interactive theorem proving. The field of Proof Engineering offers the closest efforts, and although this field has not addressed CAS integrations, the Eclipse Proof General tool – which is built on that foundation – seemed a logical starting point for integrating tools. This chapter presents our architecture and its implementation in a system we call the Prover's Palette.

The first concrete integration we undertook in the Prover's Palette connects Isabelle with the computer algebra system QEPCAD. This pairing was chosen because QEPCAD performs well with many of the types of problems Isabelle struggled with in our case study, specifically those involving non-linear real arithmetic. Our integration is described in Chapter 8 and has resulted in a useful, out-of-the-box system. The chapter also shows some interesting ways in which this integration can be used.

This integration demonstrates that our hypothesised approach was possible, but it does not show that it is useful beyond QEPCAD. To show that the Prover's Palette is generalisable and extensible, we embarked on a second integration, this time between Isabelle and Maple. This integration is described in Chapter 9. We show that we are able to re-use a lot of code from the Prover's Palette framework, even lifting some code from the QEPCAD integration to be part of the core.

The final test we explore is whether the Prover's Palette is useful in the heat of complex proofs. This is investigated by verifying an algorithm which computes Delaunay triangulations. Chapter 10 shows a number of different ways in which the Prover's Palette was useful in this proof. This includes helping to formulate essential invariants of the algorithm, giving us early indication when certain goals were impossible, and providing us with counterexamples to help repair flawed lemmas.

This thesis concludes in Chapter 11 with a description of how our work has contributed to the field of mechanical theorem proving. The wider context of our research is presented, and some exciting future directions are described.

Chapter 3

Isabelle Preliminaries for Geometric and Algorithmic Verification

The initial goal of our research was to gain a better appreciation of what would be necessary to increase the appeal of mechanised mathematics to a wider community. We embarked on a case study to afford us greater insight into the strengths of these tools — interactive theorem provers in particular — and the weaknesses which render them so unappealing. The Graham's Scan algorithm [69] for finding convex hulls is the particular problem we chose to investigate; the case study itself commences in the next chapter, with a description of the algorithm and a presentation of our verification. First, in this chapter, let us explain the reasons for choosing this particular case study and introduce the tools and formal theories we will use.

3.1 Why Verify Graham's Scan?

Our motivation for choosing the Graham's Scan algorithm is that at its core it requires a large amount of geometric analysis. As discussed in Chapter 1, problems in geometry appeal to intuition in a particularly strong way, with all the attendant benefits and risks. As discussed in Chapter 2, supporting intuition within rigourous TP environments is a particularly important challenge. Furthermore, this algorithm brings in many subtleties about degenerate cases and complex dependencies that stretch the ability for humans to reason reliably, thus playing to the strengths of TPs.

The wider area of algorithmic verification, of course, holds the promise of immense benefits for industry and society, due to the increasing reliance on software in many areas of life. Within computational geometry alone, where convex hulls are one of the fundamental objects, applications are found in domains ranging from mechanical engineering and space exploration to statistics and medicine. Often, the *correctness* of a computer program is demonstrated by showing it passes a suite of unit tests. The justification for correctness may include a pen-and-paper explanation or even a human proof of why the algorithm works, but these proofs are rarely subjected to the same level of scrutiny as mathematical results, and frequently degenerate cases will be overlooked. A mechanised TP approach to reasoning about geometric algorithms would achieve the twin goals of boosting confidence in them and identifying gaps which could be fixed and tested. As algorithms in computational geometry are being considered for use in safety-critical situations such as air traffic control, this is a highly desirable result which is beginning to attract widespread interest.

It could be argued that this case study will be of more interest to computer scientists than mathematicians: a large amount of the proof will require manipulating computer-science specific data structures and reasoning about algorithmic constructs such as loops. However, it is not only the case that convex hulls play a central role in pure mathematics, but algorithms themselves are a legitimate and increasingly popular object of mathematical study. Formally verifying geometric algorithms requires constructing a great deal of rigorous mathematics, not only in the formal statements of mathematical constructs — no different to the majority of traditional mathematics — but also in the mechanisation of fully formal proofs.

There is an additional reason we feel this study is particularly interesting and timely for the mathematical community. There has been a trend in recent years for *ad hoc* computer programs to be written to aid the construction of large proofs, such as the Four Colour Theorem and Kepler's Conjecture mentioned in Chapter 2. Interestingly, Hales's' original proof of the Kepler conjecture itself relied on many algorithms including some closely related to those used in computational geometry. Although he chose to combat the skepticism to his proof by proving the conjecture case-by-case inside a TP, he could have adopted the different approach of verifying the algorithms themselves. The feasibility of this alternative approach will be understood better by our case study.

3.2 Isabelle

Our case study will be carried out in the theorem prover Isabelle. As mentioned in Section 2.2.2, Isabelle is one of the leading contemporary interactive theorem provers, with a good breadth of theories and a mature suite of end-user tools. This section will give a more in-depth description of Isabelle, with the following sections presenting formal theories which are relevant to our verification of geometric algorithms.

3.2.1 Isabelle/HOL

Isabelle is a *generic* proof assistant, providing a language for users to define their own mathematical formulas, and then prove these using a standard logical calculus — or, in some cases defining entirely new logics. Isabelle's built-in logic, the *meta-logic*, is intended only for the formalisation of other logics, known as the *object-logics*, with soundness enforced by type-checking in the underlying programming language (ML). As noted in Section 2.2.2.1, there are a number of object-level logics in Isabelle, including first-order logic (FOL), Zermelo-Fraenkel set theory (ZF), and higher-order logic (HOL), used to express particular mathematical theories.

The Isabelle/HOL [134] implementation, which will be used in our case study, uses the latter of these, and is heavily influenced by HOL theorem prover [67]. Higher order logic provides a framework capable of quantifying over sets and functions, necessary for reasoning about algorithms and other sophisticated mathematical concepts. Furthermore, this logic is strongly typed, ensuring that only type correct terms are permitted, thus simplifying the statement of definitions and theorems.

An important aspect of theorem proving in Isabelle/HOL is the so-called HOL methodology, advocating the use of definitions rather than postulates in formal proofs. This ensures that theories are developed only as conservative extensions of existing ones, thereby ensuring consistency. However caution is still required as a wrong definition can easily lead to the wrong properties.

One of the significant benefits of using Isabelle/HOL is that it provides an extensive library of mature formalised theories covering many areas of mathematics. These include elementary number theory, analysis, algebra, and set theory. As shall be seen in future sections, Isabelle/HOL's theory of lists and its development of Floyd-Hoare logic greatly assist our verification of Graham's Scan.

3.2.2 Proof Construction

Proof construction is the process of formally deriving new rules, called theorems or lemmas, from existing rules or definitions. In Isabelle/HOL this is typically done using *tactics*, which are commands that allow the user to apply rules selectively or invoke

Isabelle's automatic functions[1]. Tactics permit the user to reason about proof goals in a mechanically verifiable way. The simplest tactics use higher order resolution to apply known rules and definitions to a proof goal; these applications can proceed in either a forward or backward direction.

Backward proofs are preferred when one wants to start with a goal and refine it to progressively simpler subgoals until all become an instance of some axiom or previously proved theorem.

Forward proofs are usually used to generate new assumptions. To achieve this the antecedents or assumptions of a rule can be resolved with other rules. This process can continue until either the conclusion is the instance of some assumption or the goal is an instance of a theorem.

Isabelle itself is based on the LCF approach (Section 2.2.2), and it provides strong guarantees of correctness based on a small kernel. The wide range of inference rules and tactics available ultimately decompose down to a sequence of a elementary rules, represented internally as ML functions converting from one expression of a theorem to another (`thm` $\rightarrow$ `thm`). The kernel runs this sequence, and a theorem is verified if the kernel is ultimately able to convert it to `true`. Only a small number of elementary rules are available, and both the kernel and the rules have been closely inspected by a wide audience over several decades: this gives a high degree of confidence in any result so verified. The rules and tactics can be extended, and any verified result (of type `thm`) can be referenced subsequently, making the system itself very powerful.

As mentioned in Chapter 2, Isabelle also has two styles of writing proofs, procedural and declarative. In Isabelle's procedural style of proof, simple commands are of the form `apply (tactic)`, as shown in Figure 3.1; this style makes it explicit what the TP is doing at every step, but it is often necessary to run the proof in the system to see what is happening. In contrast, a more declarative style of proof can be achieved by using the Isabelle/Isar syntax [133], an extension of Isabelle which was influenced by the Mizar system [162], and attempts to mimic the the style used in common mathematical practice. These proofs can stand alone from the system and be comparatively comprehensible, although they can be more effort to compose and, still being formal, tend to fall some way short of the elegance often desired in proof exposition. The declarative style is shown in Figure 3.2.

[1]Isabelle's built-in automation will be described in the following section

3.2.3 Automation

Isabelle has a number of sophisticated tactics and tools beyond mere rule application, capable of performing automation and simplifying proofs considerably. The *simplifier* is a key component which enables this automation: in its most basic form, simplification in Isabelle means repeated application of equations from left to right, substituting one value for another. This is also known as term rewriting, with the equations referred to as rewrite rules. This nomenclature underscores the important point that terms do not necessarily become simpler in the process!

Isabelle's simplifier supports unconditional rewritings as well as conditional rewritings, where a pre-condition has to be met before the substitution can be applied. It can also make use of contextual information, and permits new rules to be added to the rewrite set either permanently or temporarily. As a result, the standard simplification tactic `simp` is one of the single most powerful automation tools in Isabelle. By annotating proved lemmas with the token `[simp]`, the standard simplifier will use that lemma as a rule. This allows a newly developed theory to have expressions easily reduced to a canonical form, and, in some cases, entire decision procedures can be encoded, proving some goals automatically.

In addition to the simplifier, Isabelle has other automation components which are frequently used:

- **Classical Reasoner**: Isabelle has a reasoner which automatically performs certain long chains of reasoning steps in natural deduction style. Several automatic tactics (proof commands) are provided, including `force` and `auto`, which attempt to prove all subgoals using search and backtracking.
- **Arithmetic Decision Procedures**: Many arithmetic expressions are simplified using built-in procedures that go beyond mere rewrite rules. Linear real arithmetic and Presburger arithmetic problems are handled particularly well, and quite recently, the proof method *sos* (sum of squares) has been introduced for nonlinear real arithmetic.
- **Algebraic Decision Procedures**: The Gröbner bases decision procedure is provided for users to solve simultaneous polynomial equations.
- **External Provers**: In addition to its built-in automation, Isabelle has been integrated with external solvers (SMT, SDP) through extensions, and now comes tightly integrated with fully automated first-order provers (E, SPASS, Vampire) through Sledgehammer [20],
- **Automatic Refutation**: Isabelle provides several tools which automatically search for counter-examples and if detected warn the user that the current goal is unprovable. Nitpick [21] searches for a counter model using external SAT solvers, and Quickcheck [17] evaluates formula on random values for free variables using

```
theory Group1
imports Main
begin

typedecl pt
typedecl line

consts onLine :: "[pt, line] ⇒ bool"

definition ptsOfLine :: "line ⇒ pt set"
    where "ptsOfLine a ≡ {X. onLine X a}"

axioms
AxiomI12: "A≠B ⟹ ∃! l. onLine A l ∧ onLine B l"

theorem one: "b≠a ⟹
                  ∃ A. ptsOfLine a ∩ ptsOfLine b = {A} ∨
                       ptsOfLine a ∩ ptsOfLine b = {}"
apply auto
apply (simp add: ptsOfLine_def)
apply (drule_tac x=x in spec)
apply auto
apply (rule ccontr)
apply (drule AxiomI12)
apply auto
done

end
```

FIGURE 3.1: An excerpt from the mechanisation of Hilbert's *Grundlagen* showing Isabelle's syntax for types, definitions, theorems, and proofs.

a code generator. For arithmetic goals, the Isabelle's built-in decision procedure `arith` is capable of finding counterexamples.

3.2.4 Example Theory in Isabelle

This section is intended to give the interested reader a flavour of how formalisations are built up and presented in Isabelle using *theory files*. We will review one example in depth, taking an excerpt from our formalisation [119] of Hilbert's *Grundlagen* [85] shown in Figure 3.1.

To begin with, it is necessary to name a new theory and state what previous Isabelle developments it will build upon. The first line in Figure 3.1 achieves this by stating:

```
theory Group1
imports Main
```

This tells us that theory `Group1` relies on the existing theory `Main`, which is the union of all the basic predefined Isabelle theories like sets and lists. Thus `Main` is the parent theory of `Group1`. This is followed by the declaration of two new types:

```
typedecl pt
typedecl line
```

Note that the types are not defined, so nothing is known about them except that they are nonempty, allowing functions and predicates to take the parameters *pt* (standing for point) and *line*. This is in keeping with Hilbert's intention that points and lines should be primitive notions with their properties only defined through the axioms. Hilbert also introduced some primitive relations linking these types. One of which was the concept of a point lying on a line. In Isabelle this is formalised using the command:

```
consts onLine :: "[pt, line] ⇒ bool"
```

In contrast to many functional programming languages, Isabelle insists on explicit declarations of all predicates (keyword *consts*). The predicate *onLine* takes two arguments, one of type *pt* and one of type *line*. It represents the notion of a particular point lying on a line. Declarations and definitions of functions can be merged by using *definition...where*. Below the predicate *ptsOfLine* is declared and defined using this construct:

```
definition ptsOfLine :: "line ⇒ pt set"
    where "ptsOfLine a ≡ {X. onLine X a}"
```

This predicate takes a specific line and represents the set of points that lie on the line.

Although Isabelle/HOL strongly encourages new theories to be conservative extensions of existing theories (thus ensuring consisteny), it does permit the use of axioms. This can be useful for some projects, such as formalising and exploring new axiomatic systems such as Hilbert's. The first two axioms of Hilbert's Grundlagen are combined and formalized below:

```
axioms
AxiomI12: "A≠B ⟹ ∃! l. onLine A l ∧ onLine B l"
```

Due to the fact Isabelle can infer types, it is not necessary to state the types of each variable explicitly. As the predicate *onLine* takes a *pt* and a *line* it can be inferred that the variables *A* and *B* must be of type *pt* and the variable *l* must be of type *line*. From this it can be deduced that *AxiomI12* formally states that if two points are distinct (i.e. not equal) then there exists a unique (∃ !) line *l* on which both points lie.

The example proof which is presented is that of Hilbert's first theorem. The theorem states that any two distinct lines either have one point or no point in common.

```
theorem one: "b≠a ⟹
                    ∃ A. ptsOfLine a ∩ ptsOfLine b = {A} ∨
                         ptsOfLine a ∩ ptsOfLine b = {}"
```

The first line establishes a new conjecture to be proven and gives it the name `one` by which it can be referred to later on. The main goal is to show that the intersection of the set of points on line a with the set of points on line b is either a singleton set or the empty set. The proof of this theorem in Figure 3.1 follows a procedural style. As can be seen, this proof is difficult to understand. It requires a user to process it back in Isabelle for a full understanding and knowledge of the intermediate subgoals.

As an aside, it is worth noting that theorem `one` only has one assumption. If there had been multiple premises, say n, then the theorem could have been written using the following Isabelle-HOL notation:

$$[|\ \gamma_1\ ;\ \ldots\ ;\ \gamma_n\ |] \Longrightarrow \gamma$$

This is equivalent to $\gamma_1 \wedge \ldots \wedge \gamma_n \Rightarrow \gamma$. The Isabelle notation will be used throughout this thesis when representing lemmas and theorems formalised in that system.

Another consideration is that as an alternative, the proof could have been constructed using Isar [133], an extension of Isabelle which allows more structured, readable proofs to be written (see Figure 3.2). This style of proof allows the naming of assumptions and the proof state to be made explicit, making it easier for a reader to follow. Despite this, however, we found that Isar did not facilitate easy proof exploration. For this reason, all the case studies contained in this thesis have been carried out in the procedural style[2].

Finally, to close a theory file so that it can be inherited as a parent theory, the keyword `end` has to be written. This completes our example.

Now that we have presented how theory files are constructed and proofs written in Isabelle, let us turn our attention to verifying algorithms within Isabelle. This can be achieved using Isabelle's development of Floyd-Hoare logic, which will be described in the following section.

3.3 Floyd-Hoare Logic

Floyd-Hoare logic provides a framework for reasoning mathematically about imperative computer programs in a sound, rigorous and transparent way. It was developed by Hoare in 1969, after being influenced by the work of Floyd [87], and was first fully mechanised by Gordon [67] in the theorem prover HOL using an embedding of an

[2]It is hoped that one day Isabelle will provide the functionality to automatically convert a procedural proof into a declarative proof. Recent work by Whiteside *et al.* gives hope to this dream [182].

```
theorem one_Declarative:
   assumes ab_distinct: "b≠a"
   shows "(∃ A. (ptsOfLine a) ∩ (ptsOfLine b) = {A}) ∨
          (ptsOfLine a) ∩ (ptsOfLine b) = {}"

   proof (rule ccontr)
     assume notConclusion:
        "¬((∃A. ptsOfLine a ∩ ptsOfLine b = {A}) ∨
              ptsOfLine a ∩ ptsOfLine b = {})"

     from notConclusion obtain A where
       "(∀A. ptsOfLine a ∩ ptsOfLine b ≠ {A}) ∧
       A ∈ ptsOfLine a ∧
       A ∈ ptsOfLine b"
     by auto

     from this and ptsOfLine_def have
       non_intersection: "∀A. {X. onLine X a} ∩
         {X. onLine X b} ≠ {A}" and
       A_on_a_and_b: "onLine A a ∧ onLine A b"
     by
       from non_intersection and A_on_a_and_b
       obtain B where
         B_on_a_and_b: "onLine B a ∧ onLine B b" and
         B_not_A:" B≠A"
       by auto

     from this and AxiomI12 have
      unique_line_A_B: "∃!l. onLine B l ∧ onLine A l"
     by
       from unique_line_A_B and ab_distinct and
         B_on_a_and_b and A_on_a_and_b
     show "False"
       by auto
   qed
```

FIGURE 3.2: Declarative proof of Theorem one in Isar

annotated *while* language. More recently, the logic has been mechanised in Isabelle [131]. It is this formalisation we will take advantage of while verifying the Graham's Scan algorithm.

Hoare introduced a notation, called a *partial correctness specification*, for specifying what a program does. It is written as a Hoare triple:

$$\{P\}\ C\ \{Q\}$$

where P and Q are pre- and post-conditions on the programming variables used in the program C. The statement $\{P\}\ C\ \{Q\}$ is true if and only if:

Whenever C is executed in a state satisfying P,
if the execution of C terminates,

then

the state in which C terminates satisfies Q.

As it is not necessary for the execution of C to terminate when started in a state satisfying P, the specification $\{P\}\ C\ \{Q\}$ is only partially correct. A *total correctness specification*, $[P]\ C\ [Q]$, is a stronger kind of specification, true if and only if:

Whenever C is executed in a state satisfying P,
then
the execution of C terminates, and
the state in which C terminates satisfies Q.

Total correctness is what ultimately needs to be proved when verifying a program. Informally:

total correctness = termination + partial correctness

Floyd-Hoare logic provides the axioms and rules of inference needed to prove a program specification correct. Figure 3.3 lists rules for common programming constructs[3]. We will look at two in particular. Firstly, the composition rule: this allows one to prove the correctness of specification for a sequence of statements by splitting them up into smaller statements, identifying *mid-conditions* (intermediate assertions), and proving the correctness of specifications for the smaller statements. For example, given a program of two statements, $S\ ;\ T$, with a pre-condition P true before S and post-condition R to be shown true after T, we can show correctness if we can identify a Q such that $\{P\}\ S\ \{Q\}$ and $\{Q\}\ T\ \{R\}$.

The challenge, typically, is in identifying the mid-condition; this becomes particularly hard when working with loops. The second rule we will look at is the *while*-rule:

$$\frac{\{R \wedge S\}\ C\ \{R\}}{\{R\}\ \ \mathit{while}\ S\ \mathit{do}\ C\ \{R \wedge \neg S\}}$$

In words, if an assertion R is preserved by a program C whenever S holds initially, then R will be preserved by iteratively running C for as long as S is true. We say that R is an *invariant* of C, and S is the *loop test.* The loop test is explicit in the program, but the loop invariant must usually be found as part of the formal proof, and this — as we will see several times in this thesis — is where the difficulty lies.

Note that this form of the *while*-rule is not sufficient to show that the loop terminates. In Hoare's approach, termination — and thus total correctness — is demonstrated with the inclusion of a non-negative integer that decreases on each iteration of

[3]A full explanation of these rules is beyond the scope of this work. The reader is referred to Hoare's original exposition [87].

Rule Name	**Definition**
Assignment axiom	$\{Q[E/V]\}\ V := E\ \{Q\}$
Composition rule	$\dfrac{\{P\}\ S\ \{Q\}\ ,\ \{Q\}\ T\ \{R\}}{\{P\}\ S\,;T\ \{R\}}$
Conditional rule	$\dfrac{\{P \wedge B\}\ S\ \{Q\}\ ,\ \{P \wedge \neg B\}\ T\ \{Q\}}{\{P\}\ \texttt{if}\ B\ \texttt{then}\ S\ \texttt{else}\ T\ \{Q\}}$
Consequence rule	$\dfrac{P' \to P\ ,\ \{P\}\ S\ \{Q\}\ ,\ Q \to Q'}{\{P'\}\ S\ \{Q'\}}$
While rule	$\dfrac{\{R \wedge S\}\ C\ \{R\}}{\{R\}\ \texttt{while}\ S\ \texttt{do}\ C\ \{R \wedge \neg S\}}$

FIGURE 3.3: Common rules of Floyd-Hoare logic

C: this is called the *variant*. In the expanded `while`-rule, the variant is denoted by E and its initial value by an auxiliary variable n. An extra hypothesis $R \wedge S \to 0 \leq E \in \mathbb{Z}$ ensures the variant is non-negative, and as shown earlier, the curly braces are substituted by square brackets to indicate total correctness:

$$\frac{[R \wedge S \wedge (E = n \in \mathbb{Z})]\ C\ [R \wedge (n > E \in \mathbb{Z})]\ ,\ R \wedge S \to 0 \leq E \in \mathbb{Z}}{[R]\ \texttt{while}\ S\ \texttt{do}\ C\ [R \wedge \neg S]}$$

In Isabelle, we will be using Nieto and Nipkow's formal development of Hoare logic [131], which permits all the programming constructs just mentioned. As an example, the notation for a program containing a while loop is as follows:

```
theorem
VARS (V_d)
{ P }
    V_i
    WHILE S
    INV { R }
    DO
        C
    OD
{ Q }
```

V_d are the variable declarations and V_i the variable initial assignments, and, as before, C is the program, P the pre-conditions, R the loop invariant, S the loop test, and Q the post-conditions. To demonstrate the partial correctness of the theorem, it is necessary (and sufficient) to prove that the invariant R satisfies the following conditions: R holds initially; when taken with the negation of the test S, R establishes the post-conditions

Q; and that if R is true at the start of the program C, R will be true after C (in respect of variables which may have changed in C), i.e:

1. $P \rightarrow R$
2. $R \wedge S \rightarrow C$ preserves R
3. $R \wedge \neg S \rightarrow Q$

These conditions translate to a set of purely mathematical statements called *verification conditions* (VCs) which, in the case of Isabelle, are generated automatically by the applying Nieto and Nipkow's VC generation tactic `vcg`. The correctness of this tactic has also been formally proved.

3.4 An Isabelle Theory of Lists

As shall be seen in the following chapter, the Graham's Scan algorithm uses a *stack* data structure. Rather than formalise this data structure ourselves, we chose to use Isabelle's theory of *lists*. Not only was this data type defined for us, it provided all the operations needed to support a stack and also had the advantage of being a mature theory with many list properties proved.

Lists, in Isabelle, are defined for a generic type (`'a`), as follows:

```
datatype 'a list = Nil                  ("[]")
                 | Cons 'a  "'a list"   (infixr "#" 65)
```

This introduces two constructors — `Nil` which is the empty list and `Cons` which is the list concatenation operator that adds an element to the front of a list. Note that Isabelle gives alternative syntax for each of these operators; `Nil` can be written as `[]` and `Cons` can be written as `#`. The annotation `infixr` beside the `#` symbol means that the operator associates to the right. Isabelle also defines several useful functions for acting upon lists. Of particular interest to our formalisation of Graham's Scan were the following recursively defined functions:

`hd L`: returns the first element of `L` (i.e. `hd [1,2,3] = 1`)

`tl L`: returns the list without the first element (i.e. `tl [1,2,3] = [2,3]`)

`last L`: returns the final element of `L` (i.e. `last [1,2,3] = 3`)

`butlast L`: returns `L` without the last element (i.e. `butlast [1,2,3] = [1,2]`)

`L1 @ L2`: returns a list which is the result of adding the elements of `L1` to the front of list `L2`.

`distinct L`: ensures all the elements of `L` are different

`length L`: returns the number of elements in `L`

`take n L`: returns the first `n` elements of `L` (where `n` is a natural number) `nth L n`: returns the element of `L` which is in the *n*th position

In our formalisations which follow, we have adopted Isabelle's alternative syntax for `nth L n` — we shall write the more familiar term `L!n` to represent the element at the *n*th position of `L`. It should also be noted that the first element of `L` is denoted as `L!0` and the last as `L!(length L - 1)`. The function returns an arbitrary value if `n > (length L - 1)`.

The theory of lists also includes many useful lemmas. As an illustrative example let us focus on one which is used frequently in our proofs. The following theorem states that a list contains all distinct elements when any two elements in different positions are not equal:

```
distinct_conv_nth:  distinct xs =
    ∀i < length xs.  ∀j < length xs.  i ≠ j → xs!i ≠ xs!j
```

Its proof, which will not be covered here, goes by induction, as do most of the proofs concerning lists in Isabelle.

3.5 A Formal Theory of Planar Geometry

Verifying the correctness of Graham's Scan algorithm first requires the construction of a formal development of the relevant geometric concepts. In addition to many geometric algorithms, this involves reasoning about the relative position of vertices. This observation is made by Knuth in *Axioms and Hulls* [104], and his formalisation strongly influenced the mechanised foundation of our work. We shall look at this in the following section.

3.5.1 Knuth's Counter-Clockwise System

Knuth defines a counter-clockwise system using the notion of *left turn*, where the ordered triple notation *pqr* means that the point *r* lies to the left of the directed line from *p* to *q* (see Figure 3.4).

Knuth's counter-clockwise (CC) system is then described as one which satisfies five axioms that capture the minimal properties of the orientation predicate:

Axiom 1 (cyclic symmetry): $pqr \rightarrow qrp$

Axiom 2 (antisymmetry): $pqr \rightarrow \neg prq$

Axiom 3 (nondegeneracy): $pqr \vee prq$

Axiom 4 (interiority): $tqr \wedge ptr \wedge pqt \rightarrow pqr$

Axiom 5 (transitivity): $tsp \wedge tsq \wedge tsr \wedge tpq \wedge tqr \rightarrow tpr$

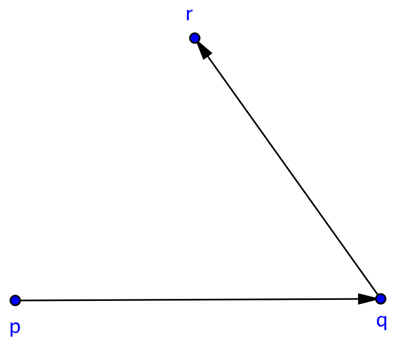


FIGURE 3.4: Point r lies to the left of the directed line from p to q.

Knuth observed that an alternate version of Axiom 5 made many proofs more succinct. He stated it as:

Axiom 5′ (dual transitivity): $stp \wedge stq \wedge str \wedge tpq \wedge tqr \rightarrow tpr$

The property of dual transitivity can be proven in this CC system from five applications of Axiom 5. As Axioms 4, 5 and 5′ are difficult to visualize, their diagrammatic representations are shown in Figure 3.5.

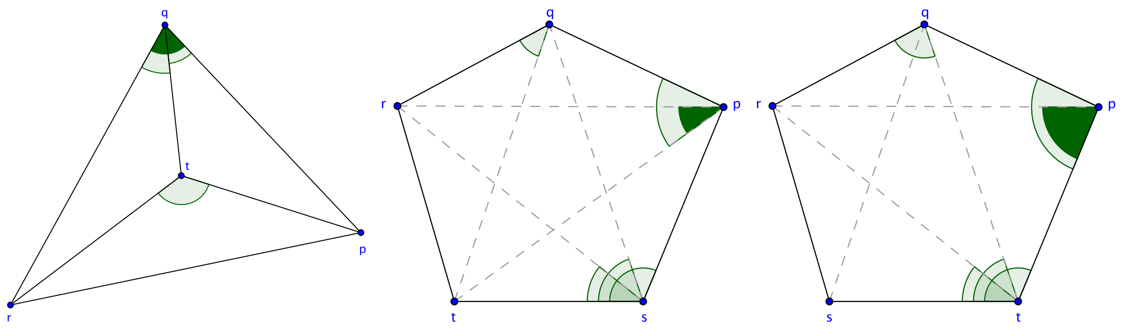


FIGURE 3.5: Knuth's Axioms 4, 5 and 5′; in each case the solid coloured angle marks the left turn which is implied by the left turns at the lightly-shaded angles

3.5.2 Our Signed Area Mechanisation in Isabelle

One drawback of Knuth's CC system for our purposes is that it disallows collinear points. This leads to many elegant results in his framework, but for real-world applications the restriction is not practical. Furthermore real-world systems are frequently based upon coordinate systems and the use of more familiar and accepted axioms is strongly preferred. Conservatively extending the usual Isabelle definition of the real numbers, by defining points as a new type, gives us a theory of geometry which has well-understood definitions and no additional axioms[4].

Our Isabelle theory of planar geometry begins, as one might expect, with the definition of the type `point` as a pair of real numbers:

```
typedef point = "{p::(real*real). True}"
```

In Isabelle, this command produces three constants behind the scenes:

```
point :: real*real
Rep_point :: point ⇒ real*real
Abs_point :: real*real ⇒ point
```

`Abs_point` and `Rep_point` are the derived coercion functions that enable one to move from the newly defined point type to its underlying representation and back. Thus `Rep_point`, for instance, enables reasoning about points to be converted into reasoning about coordinates and hence polynomials.

As our verification relies upon reasoning about the relative positions of points, we must formalise this notion. We use the concept of the *signed area* of a triangle, with the usual convention that positive area corresponds to an anti-clockwise ordering on the points. In our theory this is formalised by expanding the outer product of the edges expressed as vectors:

```
definition signedArea :: "[point, point, point] ⇒ real"
    where "signedArea a b c ≡ (1/2) *
        ((xCoord b - xCoord a)*(yCoord c - yCoord a)
        - (yCoord b - yCoord a)*(xCoord c - xCoord a) ) "
```

The predicates `xCoord` and `yCoord` are formally defined as:

```
definition xCoord :: "point ⇒ real"
    where "xCoord P ≡ fst(Rep_point P)"

definition yCoord :: "point ⇒ real"
    where "yCoord P ≡ snd(Rep_point P)"
```

whose `fst` and `snd` are the projection functions for pairs. Using these definitions it is straightforward to express the orientation of points: we say that three points *a*, *b* and *c* make a *left turn* if the signed area is positive.

```
definition leftTurn :: "[point, point, point] ⇒ bool"
    where "leftTurn a b c ≡ 0 < signedArea a b c"
```

As previously described, our theory deviates from many geometric mechanisations by including so-called *degenerate* cases where the points may be collinear. This is equivalent to the triangle defined by those points having area zero:

[4]We opted against basing our work on our earlier mechanisation of Hilbert's theory of geometry for the same reason. The coordinate system approach yielded further practical benefits of being able to use more of Isabelle's automation and thus better evaluate its strengths and weaknesses.

```
definition collinear :: "[point, point, point] ⇒ bool"
    where "collinear a b c ≡ signedArea a b c = 0"
```

A consequence of permitting collinearity is that an *ordering* of points along a line can be established. We achieve this by defining the concept of *betweenness*. For collinear points *a*, *b* and *c*, we represent and define *b* lying between *a* and *c* as follows:

```
definition isBetween :: "[ point, point, point] ⇒ bool"
                              ("_ isBetween _ _ " [60, 60, 60] 60)
    where "b isBetween a c ≡
            a≠c ∧ collinear a b c ∧
            (∀d. signedArea a c d ≠ 0 ⟶
                0 < signedArea a b d / signedArea a c d ∧
                signedArea a b d / signedArea a c d < 1 )"
```

Finally, some of our theorems benefit from having the notation of scalar multiplication available, `*s`, defined as follows:

```
definition scalMult :: "[real, point] ⇒ point"
                                            (infixl "*s" 65)
    where "a *s P ≡ (λ(p1,p2).
                    Abs_point (a*p1,a*p2)) (Rep_point P)"
```

3.5.3 A Selection of Useful Properties

Our Isabelle theory of planar geometry contains several mechanically proved lemmas concerning the properties of the predicates `signedArea`, `collinear`, `leftTurn` and `isBetween`. These included several trivial facts regarding degenerate cases, such as:

```
areaDoublePoint:  signedArea a a b = 0
twoPointsColl:    collinear a b b
notBetweenSelf:   ¬ a isBetween a b
```

Some of the key properties involving the combination of `leftTurn` and `isBetween` include:

```
newLeftTurn:                 A isBetween C D ∧ leftTurn A B C
                               ⟹ leftTurn B C D
leftTurnsImplyBetween:       leftTurn A B C ∧ leftTurn A C D ∧
                               collinear B C D
                               ⟹ C isBetween B D
conflictingLeftTurnBetween:  leftTurn A B C ∧ A isBetween B C
                               ⟹ False
conflictingLeftTurns:        leftTurn A B C ∧ leftTurn A C B
                               ⟹ False
```

Finally, two well-known results about the signed area of triangles were proven and were very useful when it came to formalising more complicated theorems, including some of Knuth's axioms in the next section:

hausner: $\mathit{signedArea}\ P\ A\ B + \mathit{signedArea}\ P\ B\ C + \mathit{signedArea}\ P\ C\ A = \mathit{signedArea}\ A\ B\ C$

cramersRule: $\mathit{signedArea}\ P\ Q\ R \neq 0 \Longrightarrow T = (\mathit{signedArea}\ T\ Q\ R\ /\ \mathit{signedArea}\ P\ Q\ R) *s\ P + (\mathit{signedArea}\ P\ T\ R\ /\ \mathit{signedArea}\ P\ Q\ R) *s\ Q + (\mathit{signedArea}\ P\ Q\ T\ /\ \mathit{signedArea}\ P\ Q\ R) *s\ R$

3.5.4 Proving Knuth's Axioms

In our theory, four of Knuth's axioms remain true and have been proven from first principles in Isabelle. The only one which required alteration was Knuth's Axiom 3. It had to have the obvious modification, which allowed points to lie in a straight line. Knuth's axioms were formalised in Isabelle as:

cyclicSymmetry: $\mathit{leftTurn}\ b\ c\ a = \mathit{leftTurn}\ a\ b\ c$

antiSymmetry: $\mathit{leftTurn}\ a\ b\ c \Longrightarrow \neg \mathit{leftTurn}\ a\ b\ c$

threeConfigurations: $\mathit{leftTurn}\ a\ b\ c \lor \mathit{leftTurn}\ a\ c\ b \lor \mathit{collinear}\ a\ b\ c$

interiority: $\mathit{leftTurn}\ t\ q\ r \land \mathit{leftTurn}\ p\ t\ r \land \mathit{leftTurn}\ p\ q\ t \Longrightarrow \mathit{leftTurn}\ p\ q\ r$

transitivity: $\mathit{leftTurn}\ t\ s\ p \land \mathit{leftTurn}\ t\ s\ q \land \mathit{leftTurn}\ t\ s\ r \land \mathit{leftTurn}\ t\ p\ q \land \mathit{leftTurn}\ t\ q\ r \Longrightarrow \mathit{leftTurn}\ t\ p\ r$

The first three lemmas, *cyclicSymmetry*, *antiSymmetry* and *threeConfigurations*, were all trivial to prove and required just the expansion of definitions and Isabelle's automatic tactics. The fourth lemma, *interiority*, was a little trickier. It could have been proven by manipulating messy algebraic expressions but an easier and more intuitive proof was found using the lemma *hausner* (from Section 3.5.3). The lemma *transitivity* was much harder to prove than the others. It required reasoning about several different configurations of points and using *cramersRule* (also from Section 3.5.3). We also proved Knuth's Axiom $5'$, represented in Isabelle as:

dualTransitivity: $\mathit{leftTurn}\ t\ s\ p \land \mathit{leftTurn}\ s\ t\ p \land \mathit{leftTurn}\ s\ t\ q \land \mathit{leftTurn}\ s\ t\ r \land \mathit{leftTurn}\ t\ p\ q \Longrightarrow \mathit{leftTurn}\ t\ p\ r$

Although the proof of this lemma could have followed a similar argument to that of *transitivity*, we chose to follow the synthetic proof sketched by Knuth. The impetus behind this was that synthetic proofs are generally preferred by mathematicians as they avoid algebraic manipulations and instead appeal to intuition. Knuth commented that *dualTransitivity* followed from just five judicious applications of *transitivity*. However, the admission of collinear points in our theory dramatically increased the number of case splits required: the synthetic proof did go through, but we ended up with an additional 13 configurations of points which had to be reasoned about formally.

With the concepts of points and their relative positions formalised, and many useful properties relating to them now proven, we are ready to build upon this foundation and verify the Graham's Scan algorithm in Isabelle. This shall be presented in the next chapter.

Chapter 4

Case Study: Proving Graham's Scan

Our case study will focus on verifying the Graham's Scan algorithm for computing convex hulls, one of the most ubiquitous structures in computational geometry. In this chapter we will first describe what a convex hull is and show how we formally define it in the theorem prover Isabelle. The Graham's Scan algorithm will then be introduced along with its formal translation using Isabelle's Hoare logic. To conclude we will present our verification of the algorithm, detailing the loop invariant we discovered and the verification conditions which ultimately needed proved to demonstrate partial correctness of the algorithm. The decreasing measure which proved termination and ultimately total correctness will also be described.

4.1 What is a Convex Hull?

Intuitively, one can imagine a set of two-dimensional points as nails sticking upwards from a board. The *convex hull* of this set of points can then be pictured as the shape produced when a rubber band is stretched around the nails and let go so that its length is minimised (see Figure 4.1).

This analogy makes the concept of a convex hull easy to grasp. However, it is not particularly easy to translate into a formal description. The book *Computational Geometry in C* gives eight different definitions; it is natural to be drawn to the simplest definition, but this is not always the best one for a formal development. Take for example the following definition, which begins with the concept of *convexity*:

> An object is *convex* if, for every pair of points within the object, every point on the straight line segment that joins them is also within the object. The *convex hull* of a set of points Q is then the smallest convex set containing Q.

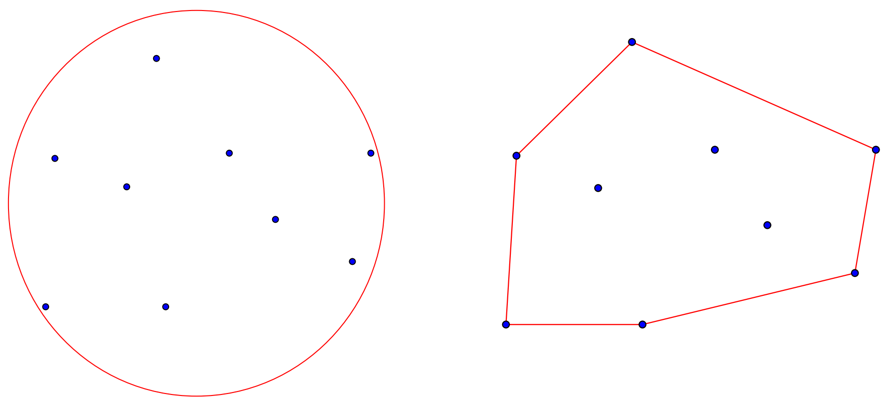


FIGURE 4.1: The diagram on the left shows the stretching of a rubber band around a point set, or nails sticking upwards from a board. At right, this band is shown contracted, tightening around these nails and yielding the convex hull of the corresponding point set.

Despite the simplicity of this definition, the requirement to check containment of all points on all line segments becomes prohibitively difficult in certain contexts. An alternative definition which suits our purposes well is the following:

> The convex hull of a set of points Q can be defined as the smallest convex polygon C, such that every point in Q either lies inside C or on the boundary of C.

Recall that in Chapter 3 we introduced Knuth's counter-clockwise (CC) system of points. Knuth explains that within this CC system, the above definition of a convex hull can be interpreted as:

> The convex hull of a set of points Q is the set of all points $t \in Q$ and $s \in Q$ such that travelling from t to s to p makes a left turn for all $p \in Q$ distinct from s and t.

In this definition the points t and s are known as the *vertices* of the convex hull of Q. Clearly this definition only holds when we are traversing the vertices in a counter-clockwise direction — an observation which is important to capture in the formal translation. If collinearity is permitted, then it is also important to modify Knuth's definition slightly to allow for this — specifically points are allowed to lie between consecutive vertices. This will be seen in the following section.

4.1.1 Convex Hulls in Isabelle

Our Isabelle translation of a convex hull is captured using an infix predicate called `isConvexHull`. This takes as first argument a list of points `C` and checks to see that this is the convex hull of the second list `Q`:

```
definition isConvexHull :: "[point list, point list] ⇒ bool"
                              ("_ isConvexHull _" [60, 60] 60)
    where "C isConvexHull Q ≡ distinct C ∧ set C ⊆ set Q ∧
    (∀n < length Q. ∀i < length C - 1.
        ( leftTurn C!(i+1) C!i Q!n ∨
          Q!n mem [C!(i+1), C!i] ∨
          Q!n isBetween C!(i+1) C!i ) ∧
        ( leftTurn (hd C) (last C) Q!n ∨
          Q!n mem [hd C, last C] ∨
          Q!n isBetween (hd C) (last C) ) )"
```

The predicate `isConvexHull` ensures that every point in `C` is distinct and belongs to the original point set `Q`. We then check every point `Q!n` in `Q` against each edge in `C` (all adjacent points `C!(i+1)` and `C!i`, as well as `hd C` and `last C`), requiring that one of three possible configurations is adhered to. Either:

- `Q!n` lies to the left of that edge; or
- `Q!n` is an endpoint of that edge; or
- `Q!n` lies on the interior of that edge

Note that we are defining this convex hull such that the vertices in `C` are ordered *clockwise*. The reason for this will be apparent when the workings of the Graham's Scan algorithm are described.

4.1.2 Computing Convex Hulls

Now that we have defined the convex hull, let us turn our attention to the problem of computing the hull for a given point set. The first paper to contain ideas which would later be adopted into convex hull algorithms was by Bass and Schubert in 1967 [15]. Since then there has been a rich variety of research on the topic, and today there exists an abundance of algorithms.

Several different methods have been found for tackling the problem in two dimensions. These can be categorised based on the order in which they examine the point set. Some of the common methods are:

- **Incremental:** the points are ordered from left to right
- **Rotational sweep:** the points are ordered using the polar angle they form with some chosen reference vertex
- **Divide-and-conquer:** the points are split into two subsets — one containing the rightmost points and one containing the leftmost points — and the convex hulls of the subsets are recursively computed and then combined

The Graham's Scan algorithm is based on the rotational sweep method. As shall be seen in Section 4.2, the algorithm first finds the rightmost lowest point then orders the remaining points by increasing polar angle around this. Before proceeding with a more detailed presentation of the algorithm, we will first introduce how the concept of rotational ordering is formally captured in Isabelle.

4.1.3 Rotational Ordering in Isabelle

In Isabelle, the notion of points being ordered rotationally is represented using polar angles. Instead of using trigonometric functions to define the polar angles, we have opted for an approach which utilises the properties of signed areas. This gives an equivalent ordering and, as shall be seen later, a more intuitive proof of the Graham's Scan algorithm. The formal definition of rotational ordering is:

```
definition ordered :: "point list ⇒ bool"
    where "ordered Q ≡ Q!0 = lowestPt Q ∧
        (∀n < length Q. ∀m.
            0 < m ∧ m < n ⟶ before Q!0 Q!m Q!n )"
```

where the predicates `lowestPt` and `before` are defined as:

```
constdefs lowestPt :: "point list ⇒ point"
    "lowestPt Q ≡ hd (sort Q)"

constdefs before :: "[point, point, point] ⇒ bool"
    "before a b c ≡ (b isBetween a c) ∨ (leftTurn a b c)"
```

and the ordering on `point` is defined for the `List` theory's `sort` as:

```
instantiation point :: "ord"
begin
definition point_le_def:
    "u ≤ v ≡ ((yCoord u < yCoord v) ∨
        (yCoord u = yCoord v ∧ xCoord v ≤ xCoord u))"
definition point_less_def:
    "x < (y::point) ≡ x ≤ y ∧ x ≠ y"
instance ..
end
```

As can be seen, the ordering on the type `point` used here respects the `yCoord` real ordering primarily, and the `xCoord` predicate secondarily. The definition of `lowestPt` is simply the first point in that ordering. Then, for a reference point `a` (which in our case will always be `Q!0`) we say that `b` comes *before* `c` if either `b` lies between `a` and `c`, or `c` lies to the left of the directed line from `a` to `b`. This allows us to define our rotational ordering `ordered` for a set `Q`, as illustrated in Figure 4.2.

We have formally proven many properties of the `ordered` predicate in Isabelle. One such property was the following:

```
orderedAndCollThenMiddleIsBetween:
    [| ordered Q; collinear Q!k Q!l Q!m;
       k<l; l<m; m<length Q |] ⟹ Q!l isBetween Q!k Q!m
```

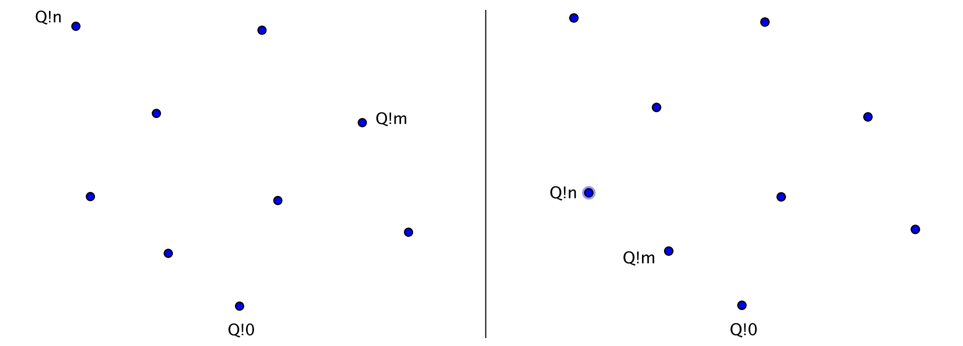


FIGURE 4.2: This figure shows how points are rotationally ordered around the lowest point `Q!0`. We say the point `Q!m` comes before the point `Q!n` in the rotational ordering if either `Q!n` lies to the left of the directed line from `Q!0` to `Q!m` or `Q!m` lies between `Q!0` and `Q!n`.

This lemma states that if the point set `Q` is ordered and three points in it are collinear, say `Q!k`, `Q!l` and `Q!m`, then we can deduce that `Q!l` lies between `Q!k` and `Q!m` if it comes after `Q!k` but before `Q!m` in the ordered list `Q`.

Another important lemma pertaining to the `ordered` predicate is:

```
orderPropWithFirst_RecentLeftTurnImpliesEarlierLeftTurn:
    [| ordered Q; leftTurn Q!0 Q!j Q!k;
       0<i; i<j; k<length Q |] ⟹ leftTurn Q!0 Q!i Q!k
```

This lemma says that if the first point in the ordered list, `Q!0`, makes a left turn with some other two points `Q!j` and `Q!k` (with `j<k` by definition of *ordered*), then for all points `Q!i` before `Q!j` (`0<i<j`), `Q!0` will also make a left turn with `Q!i` and `Q!k`. This is shown pictorially in Figure 4.3.

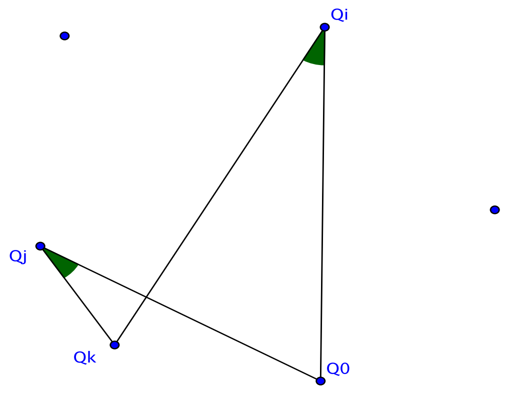


FIGURE 4.3: This illustrates that if the point `Q!k` lies to the left of the directed line segment from `Q!0` to `Q!j` then for any point `Q!i` which comes before `Q!j` in the rotational ordering of the point set `Q`, we can deduce that `Q!k` lies to the left of the directed line segment from `Q!0` to `Q!i`.

4.1.4 A Rotational Ordering Proof

To give a flavour of the proof process we will give a detailed exposition of the cases which had to be reasoned about in Isabelle to demonstrate the correctness of the lemma `orderPropWithFirst_RecentLeftTurnImpliesEarlierLeftTurn`.

From the definition of `ordered` and the fact that `i<k` it can be shown that

`leftTurn Q!0 Q!i Q!k ∨ Q!i isBetween Q!0 Q!k` (1)

If the first disjunct holds then the goal is true by `assumption`. It remains to show that the theorem holds when the second disjunct is the conclusion. Our intuition tells us that `Q!i` cannot lie between `Q!0` and `Q!k`, so there must be a contradiction in the assumptions.

Again, by the definition of `ordered` and the fact that `i<j` it can be shown that

`leftTurn Q!0 Q!i Q!j ∨ Q!i isBetween Q!0 Q!j` (2)

If the first disjunct holds then from this and the fact `Q!i isBetween Q!0 Q!k` it can be shown that `leftTurn Q!0 Q!k Q!j` (using the lemma `newLeftTurn` from Section 3.5.3). But since we already had the fact `leftTurn Q!0 Q!j Q!k` from our original assumptions we have a obtained a contradiction.

If the second disjunct from (2) holds then we can easily show that

`collinear Q!i Q!0 Q!j` (3)

Likewise, from the fact that `Q!i isBetween Q!0 Q!k` it can be shown that

`collinear Q!i Q!0 Q!k` (4)

From (3) and (4) it can be deduced that

`collinear Q!0 Q!j Q!k` (5)

However, this fact cannot hold as it contradicts with the original assumption that `leftTurn Q!0 Q!j Q!k`, completing the proof.

4.2 The Graham's Scan Algorithm

Now that we have established a way to formally represent a rotational ordering on a planar point set, we can return to looking at how the Graham's Scan algorithm works.

The first publication in the field of computational geometry is commonly attributed to Graham for his paper describing an algorithm for finding the convex hull of a set of two dimensional points [69]. Graham developed the algorithm in response to a problem at Bell Labs which required the hull of 10,000 points to be computed. There was an algorithm already implemented, but its running time of $O(n^2)$ was found to be too slow. The Graham's Scan algorithm reduced the running time down to $O(n \cdot \log n)$.

The algorithm solves the problem by maintaining a stack C of candidate points. Each point of the input set Q is pushed once onto the stack, and the points that are not vertices of the convex hull are eventually popped. The precise version of the Graham's Scan algorithm we verify is that given by O'Rourke in *Computational Geometry in C* [138], with the pseudocode included here in Figure 4.4.

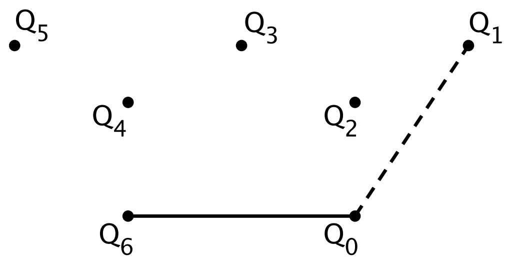
GRAHAM'S SCAN ALGORITHM

```
Inputs: Points in $Q$ such that $Q_0$ is the rightmost lowest point and
   subsequent points are sorted in increasing angular distance from
   the ray rooted at $Q_0$ and pointing along the positive $x$-axis, with
   points nearer $Q_0$ ordered first in the case of ties, $n$ = len $Q$
Variables: Stack $C$=[$Q_{n-1}$,$Q_0$], Integer $i$ = 1
while $i$ < $n$ do
   if $Q_i$ is strictly left of edge $C_0C_1$
      then Push($C$,$Q_i$) and set $i \leftarrow i$ + 1
      else Pop($C$)
```

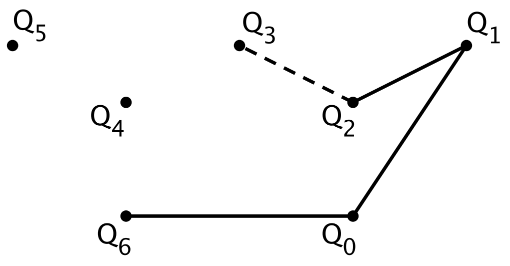
FIGURE 4.4: Pseudocode for Graham's Scan Algorithm.

As the pseudocode reveals, the input points $Q_0, Q_1, \ldots, Q_{n-1}$ are supplied ordered by increasing polar angle (anti-clockwise) around Q_0, which is the rightmost lowest point. The hull C is initialised with the first and last points in the input, Q_{n-1} and Q_0, giving a reference edge with which the algorithms starts. The points Q_1,..,Q_{n-1} are then processed in their sorted order and the hull grown incrementally around the set. As the algorithm recurses through Q it tests whether each point Q_i makes a left turn with respect to the two most recently added points in C. If so, Q_i is pushed onto the stack C and the next point, Q_{i+1}, is examined. If a right turn is made or the triple is collinear, then the most recently added point in C is popped and the algorithm again tests whether Q_i makes a left turn with the two top points in the reduced hull C. When the algorithm terminates, the stack C contains precisely the vertices of the convex hull of Q. This process is illustrated with an example in Figure 4.5.

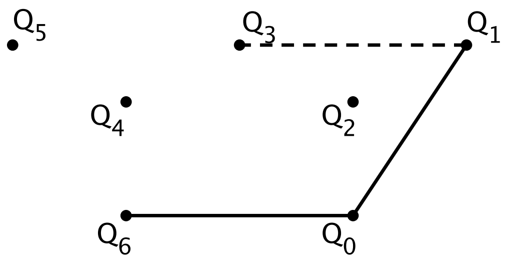
The observant reader will have noticed that the point Q_{n-1} ends up twice on the stack C, so a final pop is required. Another important observation is that the algorithm fails if there are less than three non-collinear points in the input set Q. It is necessary to make this pre-condition explicit when formalising the algorithm using Hoare logic. Section 4.3 will highlight this.



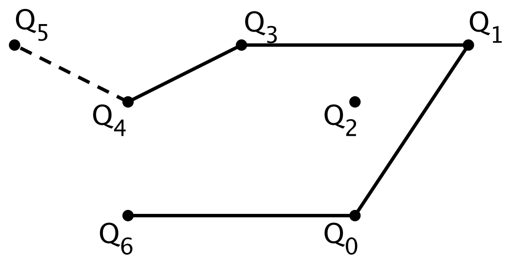
(A) The algorithm starts with all points *rotationally ordered* around the lowest rightmost (Q_0). The stack C is initialised as $[Q_6, Q_0]$ (solid line), and the first edge being considered is Q_0Q_1 (dashed line). Here, this makes a left turn with Q_6Q_0 and so Q_1 is added to the stack.



(B) On the next iteration Q_2 is added to the stack, giving $C = [Q_6, Q_0, Q_1, Q_2]$. Let us now consider Q_3: this makes a *right turn* with the two points at the top of the stack, and so Q_2 is *popped* from the stack.



(C) We now have $C = [Q_6, Q_0, Q_1]$, and we are still considering Q_3. Now this point makes a *left turn* with the two points at the top of the stack, and so Q_3 is added to the stack. On the following iteration, Q_4 is considered with respect to Q_1Q_3, giving a left turn and stack $C = [Q_6, Q_0, Q_1, Q_3, Q_4]$.



(D) We now look at Q_5 with respect to Q_3Q_4; this yields a right turn, so Q_4 is popped. Q_5 is then considered with respect to Q_1Q_3, and as these are collinear, there is still no left turn and Q_3 is popped. Q_5 is then considered with Q_0Q_1, finally yielding a left turn, and Q_5 is added. Lastly $Q_1Q_5Q_6$ makes a left turn, and the algorithm terminates with the convex hull of $Q_0Q_1Q_5Q_6$.

FIGURE 4.5: Illustration of Graham's Scan Algorithm

4.3 Formal Statement of Graham's Scan

Armed with the necessary Isabelle theories of lists and geometry (described in Chapter 3), we can present an Isabelle theorem for the correctness of the Graham's Scan algorithm. This is done using Isabelle's Hoare logic (see Section 3.3), which requires us to annotate the algorithm with the correct pre-conditions, loop invariant, and post-conditions. Finding the correct invariant was one of the most challenging tasks, and we will describe this further in the next chapter; here we will present the completed theorem, shown in Figure 4.6, highlight the key components of the theorem statement, and then proceed to explain its proof.

```
theorem GrahamsScan: "

VARS (Q::point list) (C::point list) (i::nat)

{ ordered Q ∧ distinct Q ∧ ¬ allCollinear Q }

i := 1;
C := [hd Q, last Q];
WHILE i < length Q
INV { invariant_GS Q C i }
DO
  IF leftTurn (C!1) (C!0) (Q!i)
  THEN
    C := (Q!i) # C;
    i := i+1
  ELSE
    C:= tl C
  FI
OD

{ (butlast C) isConvexHull Q }"
```

FIGURE 4.6: Formalisation of Graham's Scan Algorithm in Isabelle using the Hoare logic representation

Figure 4.6 shows how Graham's Scan is formalised in Isabelle's Hoare logic. It can be seen that it closely resembles that of the pseudo-code of Figure 4.4. Note how the pre-conditions of Graham's Scan guarantee that the input conforms to the contract of the algorithm; from Figure 4.6 it can be seen that the first pre-condition states that the input point set `Q` has been correctly ordered (such as by a pre-processing step) and the next two pre-conditions exclude degenerate cases, requiring the input to consist of distinct points not all lying on the same line. The post-condition of the algorithm is, of course, fundamental to the correctness of the proof, stating that the list `butlast C` is a convex hull of the input points `Q`. (The `butlast` is necessary because in this form, the algorithm places `last Q` as the first element and the last element of `C`; removing one of the duplicate vertices in this way gives the same resulting vertex set which would have been produced by *popping* the final point when the loop terminates.)

```
definition invariant_GS ::
           "[point list, point list, nat] ⇒ bool"
   where "invariant_GS Q C i ≡
      i ≤ length Q ∧                                    (I1)
      1 ≤ i ∧                                           (I2)
      (i = length Q ⟶ last Q = C!0) ∧                  (I3)
      ¬allCollinear Q ∧                                 (I4)
      distinct Q ∧                                      (I5)
      ordered Q  ∧                                      (I6)
      (∃L. C = L @ [Q!0, last Q]) ∧                     (I7)
      distinct (butlast C) ∧                            (I8)
      set (butlast C) ⊆ set (take i Q) ∧                (I9)
      (∀j k l. l<(length C - 1) ∧ k<l ∧ j<k ⟶
        leftTurn C!l C!k C!j ) ∧                        (I10)
      (∃n≤i. Q!n = C!0 ∧
        (butlast C) isConvexHull (take (n+1) Q) ∧
        (∀j<i. j>n ⟶ ¬leftTurn Q!n Q!j Q!i) )"         (I11)
```

FIGURE 4.7: The loop invariant for Graham's Scan formalised in Isabelle.

The loop invariant for Graham's Scan is shown in Figure 4.7. We chose to represent it in Isabelle as a new definition which takes three arguments: the input point set `Q`, the set of hull vertices `C` and the loop counter `i`. Constructing this loop invariant was a manual task which required insight into the proof and countless iterations of refinement before the 11 components (which would permit the proof to succeed) were formulated. The discovery of the loop invariant started from the knowledge that in any Hoare logic proof, the post-condition must follow from the loop invariant and negated loop test. In this example, the key component of the loop invariant which is used to prove the post-condition holds on termination of the loop is (I11). This states that, as the loop iterates through the elements of `Q`, `C` will be a convex hull of all points in `Q` up to `Q!n`, where `Q!n` is the vertex at the the head of `C`. The component (I11) also contains the fact that all points, `Q!j`, which have been *popped* while examining the new point `Q!i`, must lie inside the final hull, or in other words the point `Q!i` cannot lie to the left of the directed line from `Q!n` to `Q!j`. The essence of the proof starts to take shape with this observation, but, as the remaining components of the loop invariant testify, there is a large amount of scaffolding which must be declared and carried through the proof to support the essential logic. The need for this scaffolding should become clear as we explore the proof and the myriad of subtle case splits which arise.

4.4 Proving Graham's Scan in Isabelle

In this exposition, to improve readability we will use mathematics notation interchangeably with the formalised Isabelle definitions:

$$\begin{aligned} A &\equiv \texttt{A} \\ \text{len}\, Q &\equiv \texttt{length Q} \\ Q_i &\equiv \texttt{Q!i} \quad \text{(the ith element of list Q, zero-indexed)} \end{aligned}$$

We will also introduce symbols for some of our most frequently used Isabelle expressions:

$$\begin{aligned} \circlearrowleft ABC &\equiv \texttt{leftTurn A B C} \\ \underset{\cdots}{\circlearrowleft} ABC &\equiv \texttt{leftTurn A B C} \;\vee\; \texttt{collinear A B C} \\ \cdots ABC &\equiv \texttt{collinear A B C} \\ B \between AC &\equiv \texttt{B isBetween A C} \end{aligned}$$

Since we will often refer to directed edges we will also introduce the notation:

$$\overrightarrow{AB} \equiv \text{the directed edge from point A to point B}$$

Recall from Section 3.3 that demonstrating partial correctness entails proving three verification conditions (VCs) per loop. Thus Graham's Scan, with its single loop, yields a total of three VCs which the following subsections will explore. Selected details of some — particularly the first and third VCs — are given to familiarise the reader with the mechanical proof process and the extent of Isabelle's automation. For VC2, we focus on the crux of this proof and omit minor subgoals that might distract from proof comprehension. The formal proof includes all cases.

4.4.1 Verification Condition 1

The first of the three verification conditions is:

Preconditions $\rightarrow$ Loop Invariant (initial)

Once Isabelle performs simplification, this reduces to `VerificationCondition1` as shown in Figure 4.8. One can see that Isabelle has automatically discharged many of the invariant components which have to be proved (specifically (I2), (I4), (I5), (I6), (I8) and (I10)) and simplified many of the others (since $i = 1$ and C is known).

The remaining subgoals all become straightforward once we note that (A3) implies $3 \leq \text{len}\, Q$. Key details of their proofs are as follows:

```
lemma VerificationCondition1:
    "[| ordered Q ;                                          (A1)
        distinct Q ;                                         (A2)
        ¬ allCollinear Q                                     (A3)
     |] ⟹
        1 ≤ length Q ∧                                       (C1)
        (1 = length Q ⟶ last Q = hd Q) ∧                     (C2)
        hd Q = Q!0 ∧                                         (C3)
        hd Q ∈ set (take 1 Q) ∧                              (C4)
        (∃n≤1. Q!n = hd Q ∧
            [hd Q] isConvexHull (take (n+1) Q))"             (C5)
```

FIGURE 4.8: Verification Condition 1

- (C1) corresponds to component (I1) of the invariant. It follows from the fact $1 < 3$ and is proven easily using Isabelle's `arith` tactic due to the fact it is a linear arithmetic statement.
- (C2) corresponds to component (I3) of the invariant. Our proof used the fact that the left-hand side of the implicand is false, making the goal trivially true. (It would not be hard to show this conclusion in the general case, but it was unnecessary and more work in this case.)
- (C3) corresponds to component (I7) of the invariant. Isabelle automatically deduces this once we introduce the fact that Q is not empty. (We discovered later that this could have been automatically discharged by Isabelle, had we replaced `hd Q` by `Q!0` and hence had obtained `C=[Q!0, last Q]`.)
- (C4) corresponds to component (I9) of the invariant. Again, Isabelle automatically deduces this once we introduce the fact that Q is not empty.
- (C5) corresponds to component (I11) of the invariant. By instantiating n to be 0, this follows from the definition of convex hull.

This concludes the proof that the loop invariant holds on initialisation of the loop.

4.4.2 Verification Condition 3

The core of the proof of partial correctness involves showing that the loop invariant is preserved (VC2). Before showing this, however, let us present the proof of the third VC, as this is more straightforward. VC3 claims that the loop invariant implies the postconditions on termination:

Loop Invariant $\wedge \neg$ **Loop Test** $\rightarrow$ **Postconditions**

When expanded and simplified by Isabelle, we have the lemma shown in Figure 4.9. By combining (A1) and (A12) we can deduce:

```
lemma VerificationCondition3:
    "[| i ≤ length Q ;                                          (A1)
        1 ≤ i ;                                                 (A2)
        (i = length Q ⟶ last Q = C!0) ;                         (A3)
        ¬ allCollinear Q ;                                      (A4)
        distinct Q ;                                            (A5)
        ordered Q ;                                             (A6)
        (∃L. C = L @ [Q!0, last Q]) ;                           (A7)
        distinct (butlast C) ;                                  (A8)
        set (butlast C) ⊆ set (take i Q) ;                      (A9)
        (∀j k l. l<length C - 1 ∧ j<k ∧ k<l ⟶
            leftTurn C!j C!l C!k ) ;                            (A10)
        (∃n≤i. Q!n = C!0 ∧
            (butlast C) isConvexHull (take (n+1) Q) ∧
            (∀j<i. n<j ⟶
                leftTurn Q!i Q!j Q!n ∨
                collinear Q!i Q!n Q!j) ) ;                      (A11)
        ¬ i < length Q                                          (A12)
    |] ⟹
        (butlast C) isConvexHull Q "                            (C1)
```

FIGURE 4.9: Verification Condition 3

```
i = length Q                                                    (A13)
```

Instantiating the n whose existence is asserted by (A11), and combining this with (A13), yields:

```
n ≤ length Q                                                    (A14)
Q!n = C!0                                                       (A15)
(butlast C) isConvexHull (take (n+1) Q)                         (A16)
(∀j<i. n<j ⟶
  leftTurn Q!(length Q) Q!j Q!n ∨
  collinear Q!(length Q) Q!n Q!j) )                             (A17)
```

We have to explicitly introduce the impossible case $n = \text{len } Q$ in the course of the proof in order for Isabelle to automatically discard it. Once this is done, (A14) becomes:

```
n < length Q                                                    (A18)
```

Next we combine (A3) and (A15) to get:

```
Q!n = last Q                                                    (A19)
```

which together with the fact that all points in Q are distinct (A5), and $n < \text{len } Q$ (A18), implies that:

```
n = length Q - 1                                                (A20)
```

(A16) then simplifies to show (C1):

`(butlast C) isConvexHull Q` (A21)

This completes the proof.

4.4.3 Verification Condition 2: Left Turn Case

VC2 deals with the behaviour of the algorithm as it recurses through the loop. It shows that the invariant is preserved when entering and leaving the loop:

Loop Invariant $\wedge$ Loop Test $\rightarrow$ Loop Invariant (subsequent iteration)

In the Graham's Scan algorithm, recall that the loop recurses through the points of Q, using i as the counter. It also constructs a set C which, when the loop terminates, will be the convex hull of Q. After running Isabelle's VCG tool and performing basic simplification, we have two lemmas to prove. The first lemma deals with the case where Q_i makes a left turn with respect to the last edge constructed, *i.e.* the two points at the head of C, while the second deals with the case where Q_i does not make a left turn with this edge.

This section will cover the "left turn" case whose VC is shown in in Figure 4.10, and Section 4.4.4 will cover the "non left turn" case, with its VC shown in Figure 4.19.

The crux of these lemmas is to show that as i indexes through the points of Q, a set is being constructed in C containing the vertices of the convex hull of a subset of the points examined so far, *i.e.* C will contain a subset of $Q_0,\ldots,Q_n$. Most of the assumptions in these two lemmas are scaffolding to support the main claim, that after every iteration, C is the convex hull of $Q_0 \ldots Q_n$. Let us first discharge many of scaffolding conclusions whose proofs are simple:

- (C1) is proved by assuming `i+1 = length Q`. We then have to show `last Q = Q!(length Q-1)`, as `Q` is not empty. This is not automatically proved in Isabelle, but is simple once one finds the appropriate theorem contained in Isabelle's libraries. In this instance the theorem `last_conv_nth` discharges this.
- (C2) can be shown automatically once we perform quantifier elimination on the existential in (A7) to give us an `L`. Isabelle will then infer that the existential in the conclusion must be `Q!i # L`.
- (C3) requires first showing that `Q!i ∉ set (take i Q)`. Since (A9) tells us `set (butlast C) ⊆ set (take i Q)`, it follows that `Q!i ∉ set (butlast C)`.
- (C4) is a direct restatement of a lemma contained in Isabelle's library; the proof is immediate once `in_set_conv_nth` is found and applied.
- (C5) follows from transitivity of `set (butlast C) ⊆ set (take i Q)` (A9) and `set (take i Q) ⊆ set (take (i+1) Q)` (from Isabelle's library).

```
lemma VerificationCondition2_LeftTurnCase:
    "[| i ≤ length Q ;                                         (A1)
        1 ≤ i ;                                                (A2)
        (i = length Q ⟶ last Q = C!0) ;                        (A3)
        ¬allCollinear Q ;                                      (A4)
        distinct Q ;                                           (A5)
        ordered Q ;                                            (A6)
        (∃L. C = L @ [Q!0, last Q]) ;                          (A7)
        distinct (butlast C) ;                                 (A8)
        set (butlast C) ⊆ set (take i Q) ;                     (A9)
        (∀j k l. l<length C - 1 ∧ j<k ∧ k<l ⟶
         leftTurn C!j C!l C!k ) ;                             (A10)
        (∃n≤i. Q!n = C!0 ∧
         (butlast C) isConvexHull (take (n+1) Q) ∧
         (∀j<i. n<j ⟶
             leftTurn Q!i Q!j Q!n ∨
             collinear Q!i Q!n Q!j )) ;                       (A11)
        i < length Q ;                                        (A12)
        leftTurn Q!i C!1 C!0                                  (A13)
    |] ⟹
        (Suc i = length Q ⟶ last Q = Q!i) ∧                    (C1)
        (∃L. Q!i # C = L @ [Q!0, last Q]) ∧                    (C2)
        Q!i ∉ set (butlast C) ∧                                (C3)
        Q!i ∈ set (take (i+1) Q) ∧                             (C4)
        set (butlast C) ⊆ set (take (i+1) Q) ∧                 (C5)
        (∀j k l. l<length C ∧ j<k ∧ k<l ⟶
         leftTurn (Q!i#C)!j (Q!i#C)!l (Q!i#C)!k ) ∧           (C6)
        (∃n≤i+1. Q!n = Q!i ∧
         (Q!i # butlast C) isConvexHull (take (n+1) Q) ∧
         (∀j<i+1. n<j ⟶
             leftTurn Q!n Q!i+1 Q!j ∨
             collinear Q!n Q!j Q!(i+1) )) "                    (C7)
```

FIGURE 4.10: Verification Condition 2: The Left Turn Case

We now come to (C6) and (C7), the more interesting subgoals that have to be proven.

4.4.3.1 Proving (C6)

Here we are showing that loop invariant component (I10) is preserved when the new point being examined, Q_i, makes a left turn with respect to the edge $\overrightarrow{C_1C_0}$ and has been added to the vertex list C. We have to prove that taking any three vertices, in descending order from their position in C, will make a left turn. In other words we are showing that the points in C are ordered clockwise. Logically speaking, we have to show that, for $0 \leq j < k < l < \operatorname{len} C$:

$$\circlearrowleft (Q_i \mathbin{\#} C)_j \ (Q_i \mathbin{\#} C)_l \ (Q_i \mathbin{\#} C)_k$$

When C contains three or fewer points, this statement follows trivially from the assumptions.

When C has more than three points, we first consider the case when $j > 0$. This simplifies to showing $\circlearrowleft C_{j-1}\, C_{l-1}\, C_{k-1}$ (for the same j, k, l above), which is implied by (A10). It is unsurprising that this case is so straightforward because we are not yet saying anything about the new point Q_i.

The remaining case, $j = 0$, states that the clockwise ordering of C is preserved with the addition of the new point Q_i at the head of C. The new point must lie to the left of all pairings of points in C taken in descending order of their position in C:

$$\circlearrowleft Q_i\, C_{l-1}\, C_{k-1}$$

Here, let us split the proof into three cases: one where Q_0 is considered; one where C_0 is considered; and finally the general case where we are checking two intermediate points in the vertex list C.

In the first case, where we are examining Q_0, we must have $l = \operatorname{len} C - 1$. It has to be proved that $\circlearrowleft Q_i\, Q_0\, C_{k-1}$. The ordering on Q (A6) tells us $\circlearrowleft Q_0\, C_{k-1}\, Q_i$ or $\cdots\, Q_0\, C_{k-1}\, Q_i$. However, from the fact that $\circlearrowleft Q_i\, C_1\, C_0$ (A13), the collinear case cannot be true and we are left with the desired result.

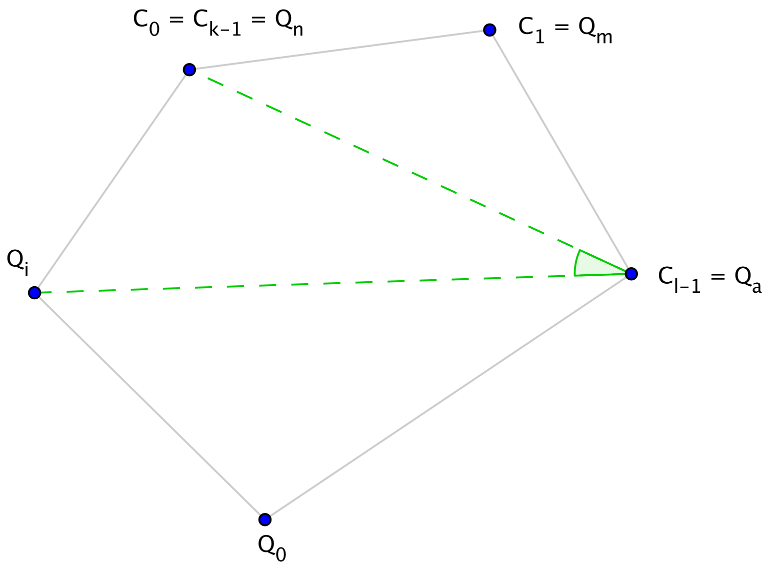


FIGURE 4.11: The second case, where we are considering the point $C_{k-1} = C_0 = Q_n$. We have to show that $Q_i\, Q_a\, Q_n$ is a left turn.

In the second case, where we are considering C_0, we must have $k = 1$. If $l = 2$ the goal is $\circlearrowleft Q_i\, C_1\, C_0$, which matches assumption (A13) and the proof is complete. For

$2 < l < \text{len}\, C - 1$ let us introduce an n and m such that $C_{k-1} = C_0 = Q_n$ and $C_1 = Q_m$; and note that by the ordering on Q we know $m < n < i$. Let us also introduce an a such that $0 < a < m$ and $Q_a = C_{l-1}$ (see Figure 4.11). The goal then reduces to $\circlearrowleft Q_i\ Q_a\ Q_n$.

We now have five distinct points (Q_0, Q_a, Q_m, Q_n and Q_i) with enough information to apply the lemma `transitivity` to yield $\circlearrowleft Q_i\ Q_a\ Q_n$. This completes the proof of the second case.

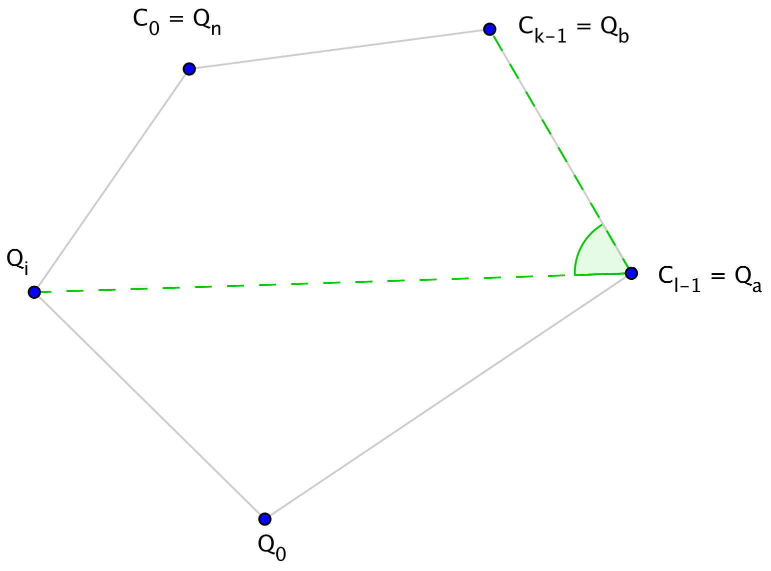


FIGURE 4.12: The general case, when we are considering two intermediate points $C_{l-1} = Q_a$ and $C_{k-1} = Q_b$. We have to show that $Q_iQ_aQ_b$ is a left turn.

In the general case, we are considering two intermediate points, *i.e.* coming after Q_0 and before Q_n in the ordering of Q. Let us call these Q_a and Q_b with $0 < a < b < n$ (and $Q_a, Q_b \in C$), as shown in Figure 4.12. We need to show that $\circlearrowleft Q_i\ Q_a\ Q_b$.

We first show the following five left turns hold:

- $\circlearrowleft Q_0\ Q_a\ Q_b$ (follows from (A10))
- $\circlearrowleft Q_0\ Q_a\ Q_n$ (follows from (A10))
- $\circlearrowleft Q_a\ Q_b\ Q_n$ (follows from (A10))
- $\circlearrowleft Q_0\ Q_a\ Q_i$ (follows from earlier proof when we considered $l = \text{len}\, C - 1$)
- $\circlearrowleft Q_a\ Q_n\ Q_i$ (follows from earlier proof when we considered $k = 1$)

We can then apply the lemma `dualTransitivity` with instantiations $t = Q_a$, $p = Q_b$, $r = Q_i$, $q = Q_n$, and $s = Q_0$, to complete the proof of the general case.

4.4.3.2 Proving (C7)

(C7) is the most significant component of VC2 (left turn case). After quantifier elimination and instantiating the bound variable as i, (C7) reduces in Isabelle to:

```
i≤i+1 ∧
Q!i = Q!i ∧
(Q!i # butlast C) isConvexHull (take (i+1) Q) ∧
(∀j<i+1. i<j ⟶
    leftTurn Q!i Q!i+1 Q!j ∨
    collinear Q!i Q!j Q!i+1 )
```

Nearly all of these are trivial and automatically proved by Isabelle. After applying `auto`, we are left needing to show only that our assumptions (A1) through (A13) imply:

```
(Q!i # butlast C) isConvexHull (take (i+1) Q)
```

So, we have to show that if Q_i lies to the left of $\overrightarrow{C_1C_0}$ (A13), then when Q_i is added to the list of candidate vertices, the resulting list is the convex hull of all points encountered in the sweep so far (through Q_i). Note that (A11) included the statement that C is the convex hull of all points that had been encountered in the sweep up to C_0 (which is equal to some Q_n, where $n \le i$). Essentially we have to show that this property will be invariant when we update C to contain Q_i.

To prove this in Isabelle, we expand the definition of `isConvexHull`, yielding the following facts to prove:

```
Q!i ∉ set (butlast C)                                  (C8)
Q!i ∈ set (take (i+1) Q)                               (C9)
set (butlast C) ⊆ set (take (i+1) Q)                   (C10)
(∀a. a < length Q ∧ a<i+1 ⟶
  (∀k < length C-1.
    ( leftTurn Q!a (butlast C)!k (Q!i # (butlast C))!k ∨
      (butlast C)!x = Q!a ∨
      (Q!i # (butlast C))!k = Q!a ∨
      Q!a isBetween (butlast C)!k (Q!i # (butlast C))!k ) ∧
    ( leftTurn Q!0 Q!a Q!i ∨
      Q!i = Q!a ∨
      Q!0 = Q!a ∨
      Q!a isBetween Q!0 Q!i ) ) )                       (C11)
```

We now note that (C8), (C9) and (C10) are identical to (C3), (C4) and (C5) respectively, and in our mechanisation we re-used their proofs. This leaves us with just (C11), the meat of the argument, left to prove. We need to show that for every directed edge in the new hull — i.e. travelling from `(butlast C)!k` to `(Q!i # (butlast C))!k` for all `k < (length C-1)`[1] — all points Q_a ($0 \le a \le i$) either lie to the left of that directed edge or are on the edge.

We split the proof into three cases:

- the first considers $Q_a = Q_i$ as a special case;

[1]This edge is equivalent to the edge $\overrightarrow{C_kC_{k+1}}$, for all $k < \text{len } C - 1$. We will use this abbreviated form in the remainder of the proof.

- the second case considers all the points Q_a where $0 \leq a \leq n$ (these are the points which lie on or in the previous hull and where $C_0 = Q_n$ by (A11));
- and the third case considers all points Q_a which may have been popped since Q_n was added to C, where $n < a < i$.

Case 1: The Point Q_i

For the case where $Q_a = Q_i$, we first check this point lies to the left or on the two newly constructed edges, namely $\overrightarrow{Q_nQ_i}$ and $\overrightarrow{Q_iQ_0}$. The proof is trivial here as Q_i is an endpoint of each edge.

We then ensure Q_i lies to the left or on the edges which belonged to the previous convex hull. This means showing that $\forall k < \text{len}\, C - 2.\ \circlearrowleft Q_i\ C_{k+1}\ C_k$. This fact is deduced by following a similar argument to that of the proof for (C6) — the proof which showed the preservation of loop invariant (I10) when the left turn case holds. Recall that here we proved $\forall j\ k\ l. 0 \leq j < k < l < \text{len}\, C.\ \circlearrowleft (Q_i\#C)_j\ (Q_i\#C)_l\ (Q_i\#C)_k$ holds under our assumptions. From this it can be shown that $\forall k < \text{len}\, C - 2.\ \circlearrowleft Q_iC_{k+1}C_k$ completing the proof.

Case 2: Points on or in the Previous Hull

In this case, we have to ensure that all the points which lie inside or on the boundary of the previous hull (i.e. all points from Q_0 up to Q_n, in the shaded region of the figures here) lie on or to the left of every edge in the newly constructed hull.

From assumption (A11) it is easy to show that this is true for all edges contained in the previously constructed hull (highlighted in Figure 4.13).

We also need to show that the points on or inside the previous hull lie on or to the left of the edge just added, $\overrightarrow{Q_nQ_i}$ (highlighted in Figure 4.14) and to the new closing edge, $\overrightarrow{Q_iQ_0}$ (highlighted in Figure 4.15).

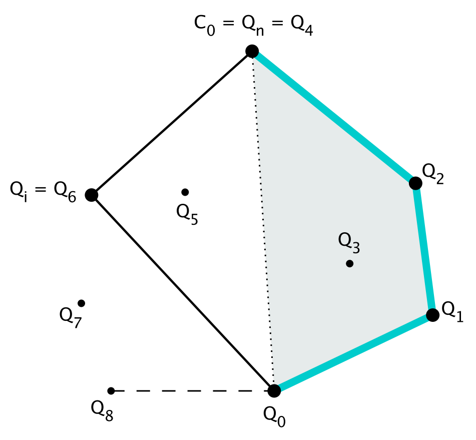


FIGURE 4.13: Case 2, previous hull edges

Let us first look at $\overrightarrow{Q_nQ_i}$: we need to show that Q_a is on or to the left of this edge for all a satisfying $0 \leq a \leq n$. For point Q_0 (when $a = 0$), we know this is true by the same argument used to prove (C6) in the previous section.

Next, consider the point Q_n (when $a = n$): this case is trivial as Q_n is a an endpoint of the edge $\overrightarrow{Q_nQ_i}$.

It remains to consider points Q_a for $0 < a < n$. From the ordering property of Q we can infer that either $\circlearrowleft Q_0Q_aQ_n$ or $Q_a \between Q_0\ Q_n$. If Q_a lies between Q_0 and Q_n then it can be shown using one of the `newLeftTurn` lemmas that $\circlearrowleft Q_nQ_iQ_a$ (i.e. that Q_a lies to the left of $\overrightarrow{Q_nQ_i}$).

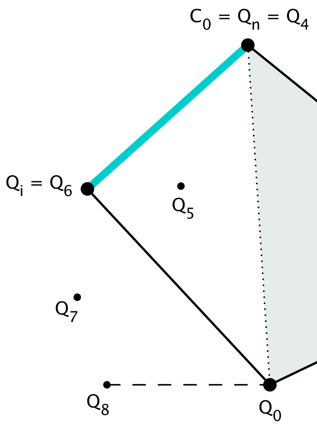


FIGURE 4.14: Case 2, new added edge

If the left turn case holds instead (i.e. $\circlearrowleft Q_0Q_aQ_n$) then we first consider when $Q_a = C_1$. Here we have to show $\circlearrowleft C_1Q_nQ_i$, which is a fact already known from assumption (A13). For $Q_a \neq C_1$ we first show that there is a $k < n$ such that $C_1 = Q_k$. Then the following five left turns can be deduced:

$\circlearrowleft Q_kQ_nQ_i$ (from A13)
$\circlearrowleft Q_kQ_nQ_0$ (from A10)
$\circlearrowleft Q_kQ_nQ_a$ (from A11)
$\circlearrowleft Q_nQ_iQ_0$ (from same argument as C6)
$\circlearrowleft Q_nQ_0Q_a$ (from A11)

Note that in the third left turn fact we also have a between case to consider, $Q_a \between Q_kQ_n$, leading easily to the fact $\circlearrowleft Q_aQ_nQ_i$. With these left turns established, the lemma `dualTransitivity` (see Section 3.5.4) can be applied with the instantiations $q = Q_0$, $t = Q_n$, $p = Q_i$, $r = Q_a$ and $s = C_1$. This gives us $\circlearrowleft Q_nQ_iQ_a$, completing the proof of case 2 for $\overrightarrow{Q_nQ_i}$.

Now let us look at $\overrightarrow{Q_iQ_0}$. For $a = 0$ here, the proof is trivial as this is an endpoint. It remains to show $\circlearrowleft Q_iQ_0Q_a$ for all other points Q_a where $0 < a \leq n$. We can deduce $\circlearrowleft Q_0Q_nQ_i$ using a slight variation of the proof of (C6), giving us the case $a = n$, and for the other points, this fact along with the lemma `orderedPropWithFirstRecent LeftTurnImpliesEarlierLeftTurn` (shown in Section 4.1.3), yields $\circlearrowleft Q_0Q_aQ_i$.

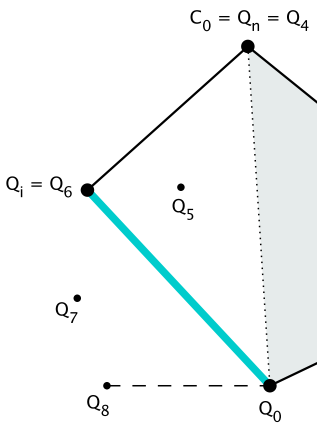


FIGURE 4.15: Case 2, new closing edge

This completes the proof that all points $Q_0, \ldots, Q_n$ lie on or to the left of all the edges in the newly constructed hull.

Case 3: Points Popped After Visiting Q_n and Before Pushing Q_i

In this case we are checking that all popped points Q_a, where $n < a < i$, lie to the left of or on the edges for the newly constructed hull.

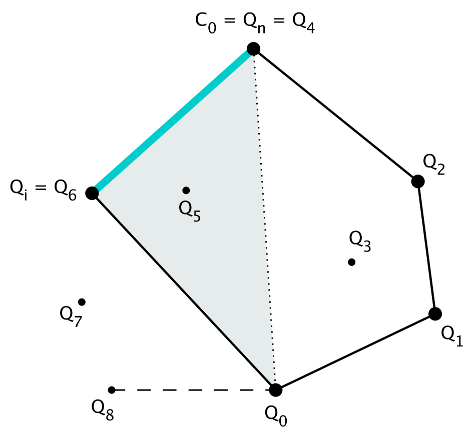


FIGURE 4.16: Case 3, new added edge

First we check these points (again shown in the shaded region) against the edge $\overrightarrow{Q_nQ_i}$ (Figure 4.16). From the last conjunct of (A11) we know that either $\circlearrowleft Q_iQ_aQ_n$ or $\cdots Q_iQ_nQ_a$. If the first disjunct holds we have satisfied the condition that Q_a lies to the left of $\overrightarrow{Q_nQ_i}$. If the collinear case holds we must show that Q_a lies on the edge $\overrightarrow{Q_nQ_i}$. Thus it must be shown that Q_a lies between Q_n and Q_i. This follows from the ordering property of Q (lemma `orderedAndCollThenMiddleIsBetween` shown in Section 4.1.3).

We now check Q_a against the edge $\overrightarrow{Q_iQ_0}$ (Figure 4.17), looking to show $\circlearrowleft Q_0Q_aQ_i \vee Q_a \between Q_0Q_i$. Again, this follows directly from the fact that Q is ordered.

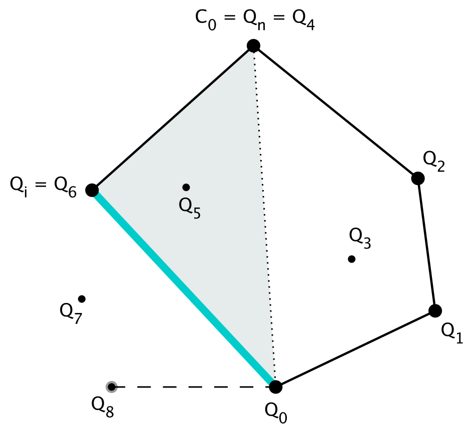


FIGURE 4.17: Case 3, new closing edge

Now let us look at Q_a with respect to edges which belonged to the previous hull (Figure 4.18). We start by deducing the following five left turn facts[2]:

$\circlearrowleft Q_nQ_iQ_0$ (follows from (A10))
$\circlearrowleft Q_nQ_iC_{k+1}$ (follows from (A10))
$\circlearrowleft Q_nQ_0C_{k+1}$ (follows from (A10))
$\circlearrowleft Q_nQ_aQ_0$ (follows from ordering of Q)
$\circlearrowleft Q_nQ_iQ_a$ (follows from (A11))

We can then apply the lemma `transitivity` with the instantiations $s = Q_i$, $t = C_0$, $p = Q_a$, $q = Q_0$, $r = C_{k+1}$ to get the fact:

$\circlearrowleft C_{k+1}Q_nQ_a$

[2]For brevity here we have omitted collinear cases. These are included in the formal proof.

Using this left turn fact with the following four left turns:

- $\circlearrowleft Q_0C_{k+1}C_k$ (follows from (A10)
- $\circlearrowleft Q_0C_{k+1}Q_n$ (follows from (A10))
- $\circlearrowleft C_{k+1}C_kQ_n$ (follows from (A10))
- $\circlearrowleft Q_0C_{k+1}Q_a$ (follows from ordering of Q)

allows us to apply `dualTransitivity`, with the instantiations $t = C_{k+1}$, $q = Q_n$, $p = C_k$, $r = Q_a$ and $s = Q_0$, to obtain the desired fact:

$\circlearrowleft C_{k+1}C_kQ_a$

This completes the proof that all popped points lie to the left or on the edges of the newly constructed hull.

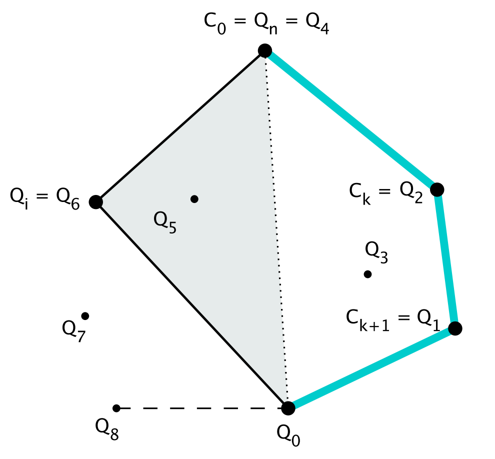


FIGURE 4.18: Case 3, previous hull edges

Special Cases

Cases 1 through 3 have shown the general proof of VC2 (left turn case), which has assumed at least 5 vertices in the previous hull C. Of course, it is also necessary to show that the verification condition holds for the special cases involving fewer points, and while the proof is not trivial, it is not particularly insightful and so has been omitted here.

4.4.4 Verification Condition 2: Non-Left Turn Case

For the case where Q_i does not lie to the left of the directed line segment $\overrightarrow{C_1C_0}$ then C_0 is *popped* from the list of vertices C. In Isabelle notation, `C` is updated to be `tl C`, and we must show that all the loop invariant components remain true with respect to this new instantiation. The lemma to prove is shown in Figure 4.19.

We will only present the proof for (C5) as this is the crux of the lemma. (The proofs of the other subgoals are largely similar to those in the left turn case of the previous section.)

4.4.4.1 Proving (C5)

This goal is saying that the loop invariant component (I11) is preserved after we pop the head of C when a non-left turn is encountered. The proof involves demonstrating:

- the updated list of candidate vertices, `butlast (tl C)`, is a convex hull for all points in Q up to C_1; and

```
lemma VerificationCondition2_NotLeftTurnCase:
  "[| i ≤ length Q ;                                              (A1)
      1 ≤ i ;                                                     (A2)
      (i = length Q ⟶ last Q = C!0) ;                             (A3)
      ¬allCollinear Q ;                                           (A4)
      distinct Q ;                                                (A5)
      ordered Q ;                                                 (A6)
      (∃L. C = L @ [Q!0, last Q]) ;                               (A7)
      distinct (butlast C) ;                                      (A8)
      set (butlast C) ⊆ set (take i Q) ;                          (A9)
      (∀j k l. l<length C - 1 ∧ j<k ∧ k<l ⟶
       leftTurn C!j C!l C!k ) ;                                   (A10)
      (∃n≤i. Q!n = C!0 ∧
        (butlast C) isConvexHull (take (n+1) Q) ∧
        (∀j<i. n<j ⟶ leftTurn Q!i Q!j Q!n ∨
          collinear Q!i Q!n Q!j )) ;                              (A11)
      i < length Q ;                                              (A12)
      ¬leftTurn Q!i C!1 C!0                                       (A13)
  |] ⟹
      (∃L. tl C = L @ [Q!0, last Q]) ∧                            (C1)
      distinct (butlast (tl C)) ∧                                 (C2)
      set (butlast (tl C)) ⊆ set (take i Q) ∧                     (C3)
      (∀j k l. l<length C-2 ∧ j<k ∧ k<l ⟶
        leftTurn (tl C)!j (tl C)!l (tl C)!k ) ∧                   (C4)
      (∃n≤i. Q!n = (tl C)!0 ∧
        (butlast (tl C)) isConvexHull (take (n+1) Q) ∧
        (∀j<i. n<j ⟶ leftTurn Q!i Q!j Q!n ∨
          collinear Q!i Q!n Q!j ) ) "                             (C5)
```

FIGURE 4.19: Verification Condition 2 Non-Left Turn Case

- all points Q_j appearing after C_1 and before Q_i in the ordered list Q, lie either on or to the left of $\overrightarrow{C_1 Q_i}$.

We begin by taking an n and m such that $Q_n = C_0$ and $Q_m = C_1$; it can be shown that $m < n < i$ (and that C has at least 3 points). (C5) can be rewritten as:

```
(butlast (tl C)) isConvexHull (take (m+1) Q)                      (C6)
∀j. m<j<i → ¬leftTurn Q!m Q!j Q!i                                 (C7)
```

The proof for these goals will be shown in the following sections.

4.4.4.2 Proving (C6)

To prove (C6) in Isabelle, we first expand the definition of `isConvexHull` yielding:

```
distinct (butlast (tl C))                                         (C8)
set (butlast (tl C)) ⊆ (take (m+1) Q)                             (C9)
(∀a < m+1. ∀k < length (tl C) - 1.
  ( leftTurn Q!a (butlast (tl C))!k+1 (butlast (tl C))!k ∨
    (butlast (tl C))!k+1 = Q!a ∨
    (butlast (tl C))!k = Q!a ∨
    Q!a isBetween (butlast (tl C))!k+1 (butlast (tl C))!k ) ∧
  ( leftTurn Q!0 Q!a Q!m ∨
    Q!m = Q!a ∨
    Q!0 = Q!a ∨
    Q!a isBetween Q!0 Q!m ) )                                     (C10)
```

The first property, (C8), is proved similarly to (C2) and the second property, (C9), is proved by contradiction. The final fact (C10) is the more interesting one to prove. It is saying that every point Q_a (where $0 \leq a \leq m$) either lies on or to the left of each edge of the updated convex hull; the edge $\overrightarrow{Q_mQ_0}$ together with the edges which travel from `butlast (tl C)!(k+1)` to `butlast (tl C)!k` (for `0 < k < length (tl C) - 1`).

We first show that all the points in Q up to Q_m (shown in the shaded region of Figures 4.20 and 4.21) lie on or to the left of the edges which belonged to the previous hull (the edges highlighted in Figure 4.20). We can prove this using (A11), the assumption which describes the convex hull of the previous loop iteration.

The final case is where we consider the edge which closes the hull, $\overrightarrow{Q_mQ_0}$ (see Figure 4.21). The points $Q_a = Q_m$ and $Q_a = Q_0$ are on the edge, and so (C10) holds, and the remaining points Q_a, $0 < a < m$, the rotational ordering gives us the desired property. This concludes the proof of (C6).

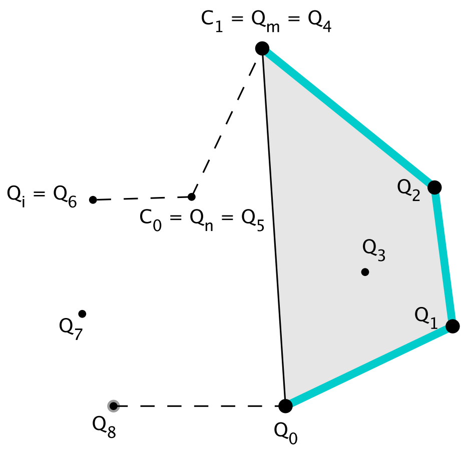


FIGURE 4.20: Previous hull edges

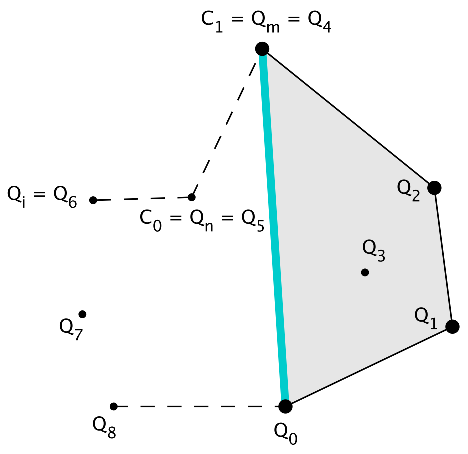


FIGURE 4.21: Closing edge

To complete the proof of (C5) we must also demonstrate (C7). This will be shown in the following section.

4.4.4.3 Proving (C7)

(C7) says that all popped points Q_j ($m < j < i$) do *not* make a left turn with respect to $\overrightarrow{Q_iQ_m}$. Let us first discharge some boundary cases:

- Whenever $j = n$ the goal reduces to (A13).
- When C contains exactly 3 points, it must be the case that $Q_m = Q_0$. Our goal reduces to showing $\neg\circlearrowleft Q_0Q_jQ_i$ for all j satisfying $0 < j < i$.

  For $0 < j < n$, we know from (A11) that `[Q!n, Q!0] isConvexHull (take (n+1)) Q`. Thus Q_j lies between Q_0 and Q_n, giving the desired non-left turn.

  If $j > n$, we proceed with a proof by contradiction, first assuming $\circlearrowleft Q_0Q_jQ_i$. By `orderPropWithFirst_RecentLeftTurnImpliesEarlierLeftTurn` (see Section 4.1.3), we get $\circlearrowleft Q_0Q_nQ_i$. But (A13) tells us $\neg\circlearrowleft Q_0Q_nQ_i$, giving the contradiction.

Having shown (C7) for the special cases, it remains to demonstrate it in the presence of the following assumptions:

$$j \neq n$$
$$\text{len}\, C > 3$$

Recall we are showing that $\neg\circlearrowleft Q_mQ_jQ_i$ for $m < j < i$, or equivalently, $\underset{\dots}{\circlearrowleft} Q_mQ_iQ_j$. This is shown in Figure 4.22, where we are proving that Q_j lies either inside the shaded region or on the dotted boundary edges Q_mQ_i or Q_iQ_0.

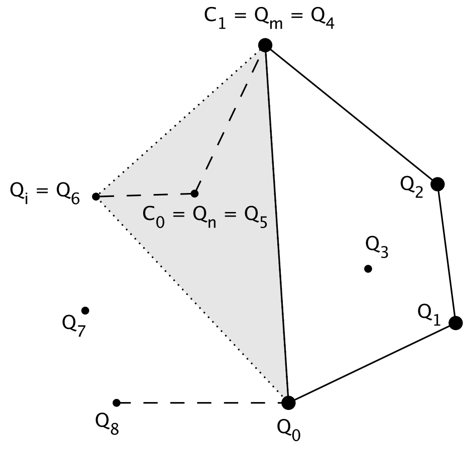


FIGURE 4.22: (C7) states that Q_j must lie in the shaded region or on the dotted edge

The proof proceeds by contradiction. We assume $\circlearrowleft Q_mQ_jQ_i$. Under this assumption, the shaded region where Q_j lies is now to the right of $\overrightarrow{Q_mQ_i}$, as shown in Figure 4.23. We will show that this cannot be the case.

From the rotational ordering of Q we can derive $\circlearrowleft Q_0Q_mQ_j$ and $\circlearrowleft Q_0Q_mQ_i$ [3]. Using `dualTransitivity` with instantiations $s = Q_0$, $t = Q_m$, $p = Q_j$, $q = Q_i$ and $r = Q_n$, we can derive $\circlearrowleft Q_mQ_jQ_n$.

If $j < n$, the convex hull fact from assumption (A11) tells us that $\underset{\dots}{\circlearrowleft} Q_mQ_nQ_j$, and we have the contradiction.

[3] Again for convenience the collinear cases are omitted.

Otherwise, where $j > n$, we use the "popped vertex" fact from (A11) to yield the fact that

$$\circlearrowleft Q_nQ_iQ_j \vee \cdots Q_nQ_jQ_i$$

From the rotational ordering, we infer that

$$\circlearrowleft Q_0Q_nQ_j \vee Q_n \between Q_0Q_j$$

These pairs of disjuncts give us four putative configurations:

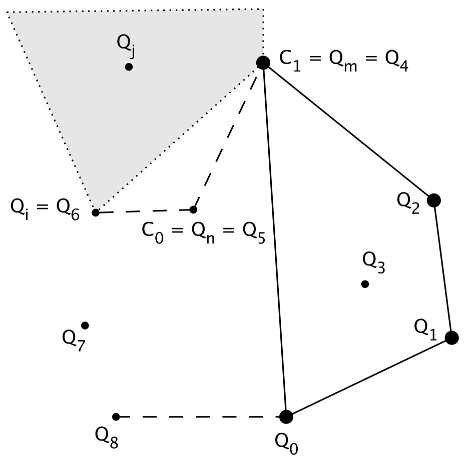


FIGURE 4.23: Contradiction of (C7)

- $\circlearrowleft Q_nQ_iQ_j \wedge \circlearrowleft Q_0Q_nQ_j$: the lemma `dualTransitivity` with the instantiations $s = Q_m$, $t = Q_j$, $p = Q_0$, $q = Q_n$ and $r = Q_i$ implies $\circlearrowleft Q_jQ_0Q_i$ which contradicts the ordering property of Q.
- $\cdots Q_nQ_jQ_i \wedge \circlearrowleft Q_0Q_nQ_j$: the collinear triplet is constrained by the ordering property of Q such that $Q_j \between Q_nQ_i$; however this implies $\circlearrowleft Q_iQ_jQ_m$ which contradicts the assumption $\circlearrowleft Q_mQ_jQ_i$.
- $\circlearrowleft Q_nQ_iQ_j \wedge Q_n \between Q_0Q_j$: it can be shown that $\circlearrowleft Q_iQ_jQ_0$; however this contradicts the ordering property of Q.
- $\cdots Q_nQ_jQ_i \wedge Q_n \between Q_0Q_j$: from the ordering property of Q, it can be shown that $Q_j \between Q_nQ_i$, which combined with the fact $\circlearrowleft Q_0Q_mQ_i$ reveals that $\circlearrowleft Q_jQ_mQ_i$; however this contradicts $\circlearrowleft Q_mQ_jQ_i$ from earlier in the proof.

This completes the proof of (C7) and hence (C5).

4.4.5 Total Correctness

So far we have concentrated on partial correctness, that is showing that if the algorithm returns an answer, that answer must be correct. It remains to show *termination*, that the algorithm does return an answer, *i.e.* that it does not run forever. As shown in Section 3.3:

total correctness = partial correctness + termination

Termination is typically shown by providing an invariant or measure which can be shown to be:

- always decreasing
- always positive
- always integral

It is easy to see that an algorithm with such a measure must terminate. Finding an appropriate measure, however, can require some ingenuity.

For the Graham's Scan algorithm, there are essentially two cases to consider. Either a new vertex is added to the convex hull stack C (left-turn case), or a vertex is popped from the stack C (non-left-turn case). In the former case, we increment by 1 both len C and i (the index of the point being looked at). In the latter case len C decreases by 1 and i remains unchanged.

By taking the linear combination of $2i - \text{len}\, C$ we get a measure which always increases. As i is always less than or equal to len Q, and C is non-empty, $2 \cdot \text{len}\, Q - (2i - \text{len}\, C)$ is a decreasing integer measure which is always positive, giving us termination.

The bound can be tightened slightly, and in our formalised proof we used:

$$2 \cdot \text{len}\, Q - 2i + \text{len}\, C - 2$$

To show termination in Isabelle, this measure must be supplied as part of the Hoare logic expression, and the verification conditions that must be subsequently proved include showing the positive decreasing nature of the measure. The proof itself is tedious, involving substantial arithmetic inequality manipulation, but is not elucidating beyond the above summary.

With termination demonstrated, a proof of total correctness for the Graham's Scan algorithm has been achieved.

4.5 Conclusion

The reader will undoubtedly have noticed that the formal proof is lengthy and in places difficult to follow, but we wanted to give a good sense of the intricacies required. As we will examine in the next chapter, the difficulties of reading the proof pale in comparison to the arduous task of creating such proofs!

Chapter 5

Observations on the Verification of Graham's Scan

The reader cannot have failed to make one overwhelming observation on the proof of Graham's Scan, even if he skipped much of the preceding chapter: the formal proof is *big*. On top of this, many of the one-line steps are the result of a lot of time spent searching Isabelle's library for the right lemma and entering the right instantiations. There is no perfect measurement for the difficulty of a formal proof, but by way of indicative quantitative measures — where the order of magnitude is significant even if the mantissa is not — we note that:

- 170 lemmas were formally proven (lemmas are usually introduced where a fact is needed multiple times)
- 5000 lines and 130,000 characters are contained in the theory files (of which about 3% are comments and extra whitespace)
- 6 months elapsed time was spent on the formal proof (one person, part-time, starting with reasonable Isabelle skills)

The difficulties associated with constructing our formal proof are detailed in the following section: we focus on the factors which contributed to the considerable time spent proving, and observe that productivity is substantially hampered by one's intuition being obscured in the process.

Despite the difficulties, it is exciting that such a formal proof is possible with current technologies. The value of this endeavour is emphasised when we compare our work with two published written proofs for the correctness of the Graham's Scan algorithm and find notable flaws in both. These flaws are described in depth in Section 5.2 and re-iterate the point we made in Chapter 1 that intuition is fallible. In the latter part of that section we draw comparisons between our mechanisation and a proof of Graham's Scan carried out in Coq. Thus, this chapter sets out two of the fundamental motivations for our research: formal proof is difficult, and formal proof is useful.

5.1 Formal Proof is Difficult

Some of the challenges in creating our proof were necessary ones, even enjoyable in cases where we were tackling rich mathematical questions: Which definitions and representations are the most natural to work with? What are the essential components of the loop invariant which give us our post-condition? What decreasing measure gives us termination? What is the essential argument of the proof?

However much of the time (and size) of our mechanical verification was consumed by more mundane activities, making the task overall a painful and difficult undertaking. On top of this, the formal proof process is rendered more difficult by our intuition being obscured along the way. We categorise the root causes of these problems in the next two sections, not merely for cathartic reasons, but because many of these activities seem fundamentally unnecessary, and in the subsequent chapters of this thesis we will turn our attention to alleviating these impediments.

5.1.1 Black Holes of Time

Proving Minute Details

Formal proof by its nature needs to be exhaustive, but wading through enormous amounts of low level detail causes exhaustion well beyond what seems needed[1]. Much of this is minute detail that would be left out in a written proof. As an example, consider massaging `(tl (butlast C))!0` to the equivalent form `C!1` when `C` contains two or more points. To introduce this equivalence in the proof one can insert it as a new subgoal (copying and pasting the terms involved), and then prove it later by applying a more generalised lemma from the library. Alternatively, for any equivalence we wish to use more than once, we could choose to make it into a lemma. In both cases, a considerable amount of time and work is needed to accomplish the obvious, and the frequency with which this is necessary is daunting.

Searching the Isabelle Library

Formalisations in Isabelle often rely on other theories; this is almost inevitable when developing conservative extensions, as usually recommended to ensure consistency. Constructing such a formalisation unsurprisingly requires familiarity with, and

[1]This can sometimes be a warning that you are working at the wrong level of abstraction.

references to, definitions and lemmas from parent theories. We found navigating these extremely tedious: even if one knows the parent theories well, it is nearly impossible to remember all the lemma names.

Our signed area geometry formalisation builds on Isabelle's theory of the reals, and our Graham's Scan mechanisation builds on these two libraries as well as Hoare logic, lists, and sets. This is a large corpus of proof and, because there are often many minute details which have to be addressed (as noted in the previous point) there are a staggeringly large number of lemmas in these libraries. The nature of interactive proof means that a great many of our proof steps required finding the relevant lemma or definition to apply.

Having so many lemmas to navigate and so many places where a lemma had to be applied meant that a large amount of our time was spent searching libraries for the single exact lemma required. We estimate that this library lookup problem accounted for approximately 30% of our time, and this is not an isolated problem: in personal correspondence with Hales, he estimates that 50% of his time has been spent looking up HOL Light's library while working on his ambitious Flyspeck Project [57].

Entering the Correct Instantiations

In addition to manually finding an applicable lemma for a given goal state, one has the added burden — in many cases — of mapping a large number of variables from the lemma to the goal. As an example, consider how the lemma *`transitivity`* (shown in Section 3.5.4) would be applied to a goal state, or how you select among lemmas with multiple variants, such as one containing *`collinear A B C`*, another containing *`collinear A C B`*, and another containing *`collinear B C A`*. A great deal of care and time is required to select the right variant and to match the instantiations of variables in a lemma with those in our subgoal. This was frequently compounded by subgoals in our proof which contained several very similar assumptions, where one misaligned variable or wrongly selected variant would still match: the mistake might not be noticed until several steps or subgoals later.

The declarative Isar language, as mentioned Chapter 3, can help resolve this situation by allowing the user to specify the *result* of applying a lemma instead of the instantiations. However specifying the correct result raises similar difficulties, and in our experiments, producing Isar scripts involved substantially more work than the procedural approach. We found it simplest to use the procedural approach (where we must attend carefully to the variable mappings) first, to generate the resulting proof states, and then to cut-and-paste that output to write the Isar as needed. For *exploration*, the procedural approach is clearly preferable, yet toggling between the two modes is awkward.

Refactoring

Another drain on our time was *refactoring* our theories, that is making changes to the structure or names. There are many reasons why these changes are desirable, from readability (to help our intuition) to necessity (correcting something that was wrong), but making these changes is not straightforward. Some of the specific refactorings we performed were:

- renaming lemmas
- generalising functions
- extracting lemmas
- changing our loop invariant
- changing our definitions

Updating the loop invariant and definitions, in our experience, are things which can happen numerous times as the proof evolves and the user realises that components therein need to be added or changed. Any of these refactorings require the user to manually locate all areas of the proof impacted — often in other files — and ensure they are updated. Isabelle has no support for making these changes or highlighting impacted areas; in fact each such change requires all affected files to be completely reprocessed in Isabelle, taking up to several minutes. Gonthier has also commented on this difficulty in the current technology, stating that in his proof of the Four Colour Theorem he spent several months refactoring his theories [66].

Resolving Isabelle Version Incompatibilities

Our initial formalisation built upon a theory of the reals, as we noted earlier. Soon after our proof was complete, a new version of Isabelle was released, where the theory of reals had been re-organised to make use of axiomatic type classes and to enable more abstraction and proof re-use. An unfortunate consequence for us was that many of our proofs broke. Lemmas and definitions which our proof had referenced were changed or removed altogether. In addition, *`simp`* and *`auto`* now performed differently. Some steps now completely failed, others which had previously discharged a subgoal now only partially simplified it, and yet others now went further or in a different direction, meaning our subsequent steps failed. Because it is difficult to see what these tactics are doing under the hood, repairing these steps is a time-consuming affair.

Subsequent Isabelle releases have not been quite as radical, but they have nearly always required at least some patching to our proofs. Besides losing time, these version incompatibility problems impede the ability to share libraries with others, as all collaborators must be synchronized to the same Isabelle version. We discuss one workaround to this problem in Section 10.2, and note that our theory files from the Graham's Scan proof have since been made compatible with Isabelle2012.

5.1.2 Intuition is Obscured

While constructing our formal theories, one of the biggest difficulties we found was that our intuition felt handicapped, as compared to how intuition usually helps when doing normal mathematics. The additional cognitive load of doing formal proof significantly impaired our productivity. The contributing factors are discussed below.

Opaque Presentation

It will have been clear to the reader in Chapter 4 that formal proof is verbose. Every fact must be explicit, and the set of assumptions can quickly grow large and hard to read. We frequently found ourselves spending many minutes trying to figure out the subset of assumptions relevant to a particular proof step, often using pencil and paper to sketch a more convenient representation. The presentation in formal proof is far less intuitive than in traditional mathematics.

This is compounded by the fact that once we had picked out the key elements in the subgoal at a given proof step, we would then apply a tactic and change the subgoal. Even with Isabelle's powerful support for mathematical symbols, we would then have many more minutes' work to again pick out the relevant portions and relate them to the mental model of the proof we had in our head.

In paper-based mathematics the variables and facts don't move around the page; we can write a note next to a fact and it stays there. The procedural style of proof in Isabelle permits us to place comments in the proof source, but none of these notes appear in the subgoal window showing the state of the proof. We cannot highlight an important fact, in pink say, or label a term as "popped points", and have that annotation kept in the subgoal on the subsequent step. Combine this seemingly-small difference with dozens of propositions and hundreds of terms in our proof state, many of which are quite similar, and the reader will have a sense of the cognitive gap between what our perception is trying to absorb and what our intuition is reasoning about.

As with the problem of supplying the instantiations (Section 5.1.1), Isar partially addresses this problem. It allows one to attach a label to individual terms or propositions and to use this label as a reference where it is needed subsequently in a proof step. However these labels do not appear in the proof state window, and considering the other difficulties of using Isar for exploration mentioned previously, the drawbacks so far outweigh the advantages.

Distracting Minutiae

We noted previously that formal proof can involve an extreme amount of reasoning about minute detail. Each lemma application might introduce several new subgoals, mostly small, but often needing attention and time nevertheless. What can be skipped in normal mathematics, or conflated into a single argument with a magic four letter acronym, must be laboriously spelled out in formal proof: there is no magic "WLOG" tactic for Isabelle.[2]

Consequently, our concentration on the essence of the proof is repeatedly interrupted by each of these minor subgoals and case splits, and due to the opaque presentation, each such interruption can cost several minutes. Even a relatively straightforward proof — say, a one-sentence hand-written proof of two "interesting" lemma applications — can easily have dozens of such details needing formal justification. By the time all the details of the first lemma are resolved, our intuition has long since forgotten what the second interesting lemma application was.

Expensive Mistakes

Many of our proof attempts started down a track which, after more or less time, we ultimately discovered was flawed. These false paths happen when doing normal mathematics, of course, but in formal proof we found they occurred with much more frequency, and that many of the flaws were ones our intuition would have spotted immediately had it not been diminished. This becomes a vicious cycle, as our intuition is further diminished by following the faulty track, abandoning it, and then restoring our state to a good one after our intuition has become attuned to that faulty track.[3]

[2]There has been recent work done in the system HOL Light implementing a "WLOG tactic" for specific common situations [81], which we will discuss in Section 6.3.3.

[3]There have been several additions to Isabelle over the years to attempt to rectify this problem somewhat. These are described in Section 6.1.

5.2 Formal Proof is Useful

Without doubt, a formal verification produces results in which we can have more confidence, for the reasons we previously mentioned at the end of Section 2.2.2.1. Another area where formal proof can be useful, surprisingly, is in improving one's intuition about the domain. When deep within a formal proof, as we have said, one frequently finds her intuition obscured, but after successfully manoeuvring through the foggy landscape, one emerges with a deeper understanding of intricacies involved in a proof and in particular a better appreciation of degenerate cases.

In our case, when we first read a variety of written proofs for the correctness of convex hull algorithms, we believed the arguments were sufficient, as presumably have editors and readers through the years. Looking back on the proofs after having completed our case study, however, we noticed multiple mistakes, some significant, in the proofs presented in two widely used textbooks. The fact that it is possible for mistakes to pass undetected for many years, in two canonical works on computational geometry and algorithms, clearly demonstrates the usefulness of formal proof.

5.2.1 Flaws Found in Canonical Proofs of Graham's Scan

O'Rourke's Proof

The textbook *Computational Geometry in C* [138], by Joseph O'Rourke, is one of the leading textbooks in the field, and was a useful source in this research for understanding Graham's Scan and existing correctness proofs. When we reviewed O'Rourke's proof sketch after our case study, we were surprised by ambiguities and even flaws which now jumped out at us.

One of these flaws is minor; in Section 3.5.3 of the book (pg. 83), a point p_n is referred to but only $p_0, \ldots, p_{n-1}$ are defined. It is not hard to determine that p_{n-1} is intended, once one is familiar with the algorithm; but for novice readers unfamiliar with the algorithm, such a mistake is potentially confusing.

The second error is more concerning, as it is the crux of the justification he gives for correctness:

> The points are now processed in their sorted order, and the hull grown incrementally around the set. At any step, the hull will be correct for the points examined so far, but of course points encountered later will cause earlier decisions to be reevaluated.

GRAHAM'S SCAN ALGORITHM (O'ROURKE'S ALTERNATE VERSION)

```
Find the interior point $X$; label it $Q_0$
Sort all other points angularly about $Q_0$; label $Q_1,\ldots,Q_{n-1}$.
Stack $C = (Q_1, Q_2) = (Q_{t-1}, Q_t)$; $t$ indexes top.
$i \to 3$
while $i < n$ do
  if $Q_i$ is strictly left of $(Q_{t-1}, Q_t)$
    then Push($C, i$) and increment $i$.
    else Pop($C$).
```

FIGURE 5.1: Pseudocode for the alternate version of Graham's Scan given in O'Rourke's book on pg. 82. Here we order points around an interior point instead of the lowest, rightmost point in Q. Note that O'Rourke's book refers to p where we have used Q and S where we have used C. We are sticking to our variable names for consistency with Chapter 4.

Firstly, let us examine what is meant by "the hull will be correct for the points examined so far". This is ambiguous: does it mean that the hull at any step is the intersection of the points examined and the final (correct) convex hull, or that the hull is the convex hull of the points examined so far? Both interpretations would yield the desired goal on termination; but a proof is of limited value if the reader cannot be clear what is meant. Ambiguities such as this are of course never permitted in formal proof.

Moreover, both interpretations are flawed! If we take the former interpretation, that the hull-so-far is the subset of the points examined which are on the final hull, it is easy to find a counter-example. If a point is ever popped, the final hull does not contain the popped point, but of course the hull-so-far contained that point prior to popping it, and during every previous iteration since the point was pushed. This interpretation is clearly incorrect.

If instead we take the invariant to be that the hull-so-far is the convex hull of the points examined so far, it can still be violated. With respect to the algorithm we describe in Figure 4.4, it is violated whenever a point has just been popped, as was the case for the former interpretation — although this interpretation could be repaired, for this algorithm, if it were amended to say "after a point is pushed". O'Rourke however uses this justification for an alternate version of the algorithm which assumes an interior point exists; it uses this point as the origin for the rotational sweep, and does *not* count it as part of the hull. The pseudocode for this alternate version is shown in Figure 5.1. With respect to this algorithm, the suggested invariant can be violated even after a point is pushed onto the hull. We also note that O'Rourke does not mention how an interior point is chosen or indeed how the alternative algorithm should behave if there is no interior point in the original set.

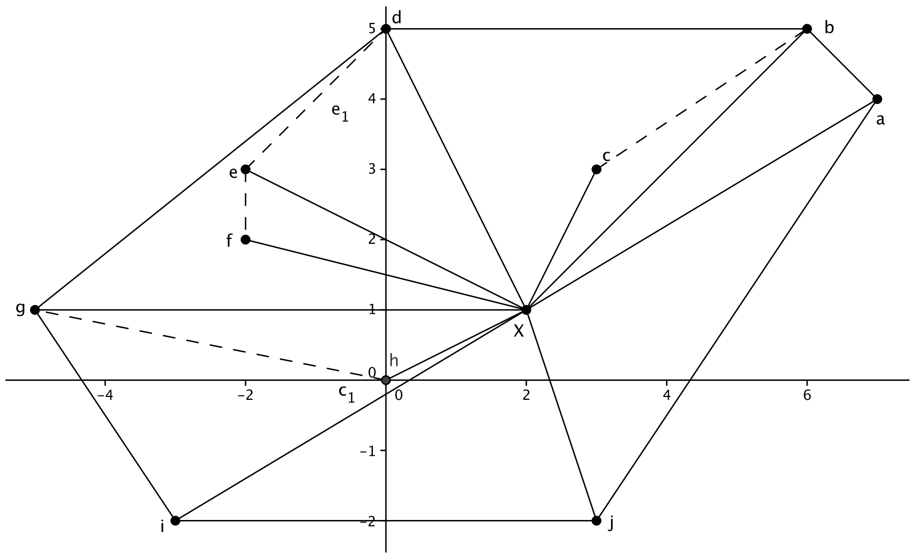


FIGURE 5.2: O'Rourke's alternate version of Graham's Scan algorithm with the interior point $X = (2,1)$ used as the reference for the rotational sweep. The remaining points, in rotational order, are $[(7,4),(6,5),(3,3),(0,5),(-2,3),(-2,2),(-5,1),(0,0),(-3,-2),(3,-2)]$.

Figure 5.2 illustrates both violations[4]. In the second iteration of the loop we examine point d and find that c needs to be popped. The hull-so-far then becomes $[a,b]$, which fails to be a hull containing some of the points considered so far, *viz.* c and d. On the third iteration of the loop, when d is pushed onto the list of candidate vertices and the hull-so-far becomes $[a,b,d]$, we still do not have the correct hull: it does not contain the point c previously examined.

This shows how seductively misleading invariants can be. O'Rourke's claim that the hull is correct is trivially true for the first two points, and for the first three points, and we assume it is true on termination; so a reader following one's intuition may believe that the preservation of correctness of the hull-so-far is the reason the proof is correct. Because this "invariant" can be false, however, its preservation is not guaranteed, and it cannot be regarded as the reason the algorithm is correct. It is little more than a flavour for why the algorithm works, and in fact it is not clear to us what the invariant should be for the algorithm referenced by O'Rourke.

O'Rourke's flawed justification for the correctness of the algorithm highlights how easy it is to overlook or be ambiguous about details of the crucial facts involved in a geometric proof. Producing a fully mechanised proof can be laborious, but does mean

[4]Nor did we have to look far for this: the diagram is taken directly from the textbook [138], p. 81.

the user cannot fall in to these types of errors. Not only is the proof more rigorous, but the user ends up having a better appreciation of the workings of the algorithm.

Cormen, Leiserson and Rivest's Proof

In the textbook *Introduction to Algorithms*, Cormen, Leiserson and Rivest also give a written proof for the correctness of Graham's Scan [41]. It is one of the simplest and most elegant proofs we have read, based around preservation merely of a *convex polygon* instead of a convex hull. But despite the obvious care and cleverness that went in to the proof, we observed two flaws once we had completed our formal proof exercise in Isabelle.

Before we discuss these errors, it is worth noting that the version of Graham's Scan used by Cormen *et al.* has several differences to the one we presented in Chapter 4. The pre-processing step orders points around the leftmost lowest point rather than the right-most lowest point. It then discards any point lying between the reference vertex Q_0 and another point in Q. The initialisation of the hull C and the loop counter i are also different: C is initialised to $[Q_2, Q_1, Q_0]$, and i starts at 3. Figure 5.3 shows the pseudocode of this alternate version of the algorithm.

GRAHAM'S SCAN ALGORITHM (CORMEN'S VERSION)

```
Find leftmost lowest point; label it $Q_0$
Sort all other points by increasing polar angle around $Q_0$,
   (if more than one point has the same angle, remove all but
    the one that is furthest from $Q_0$)
   label $Q_1,\ldots,Q_m$
top[$C$] $\leftarrow$ 0
PUSH($Q_0$, $C$)
PUSH($Q_1$, $C$)
PUSH($Q_2$, $C$)
for $i \leftarrow$ 3 to $m$
   do while the angle formed by points NEXT-TO-TOP($C$),
       TOP($C$), and $Q_i$ makes a non-left-turn
     do POP($C$)
   PUSH($C$, $Q_i$)
return $C$
```

FIGURE 5.3: The Pseudocode for the Graham's Scan Algorithm shown in the textbook *Introduction to Algorithms*. For ease of comparison we have adjusted the variable names and layout to be consistent with that used in Chapter 4.

Cormen *et al.* claim that:

Graham's Scan maintains the invariant that the points on stack C always form the vertices of a convex polygon in counterclockwise order.

They justify the claim as follows:

> The claim holds immediately after the third vertex has been added, since points Q_0, Q_1 and Q_2 form a convex polygon. Now we examine how stack C changes during the course of this algorithm. Points are either popped or pushed. In the former case, we rely on the simple geometric property: if a vertex is removed from a convex polygon, the resulting polygon is convex. Thus, popping a point from C preserves the invariant.

The authors then address the case when points are pushed. A separate argument establishes the claim that "each point popped from the stack is not a vertex of the convex hull", and the implication of these two simultaneous claims is that the resulting list must be the convex hull, according to the authors.

The two flaws in their proof which we observed are:

- The two claims are sufficient to conclude that the resulting list is *a* convex hull *if* a convex hull exists. The proof assumes both existence and uniqueness. Although this is not hard to demonstrate, at the very least reliance on uniqueness should be mentioned in the proof, particularly as the authors use the definition "the convex hull of a set Q of points is the *smallest* convex polygon for which each point in Q is either on the boundary of C or in its interior" (emphasis added, as uniqueness makes the word *smallest* redundant).
- The invariant that C is always a convex polygon is falsified if Q_2 is popped due to some Q_j satisfying $\forall k. 1 < k < j \to \neg\circlearrowleft Q_1 Q_k Q_j$ (where $j > 2$). This leaves a hull containing just two points, as shown in Figure 5.4, rather than a polygon. This can be repaired by adding the words "or C is equal to $[Q_0, Q_1]$" to the proposed invariant.

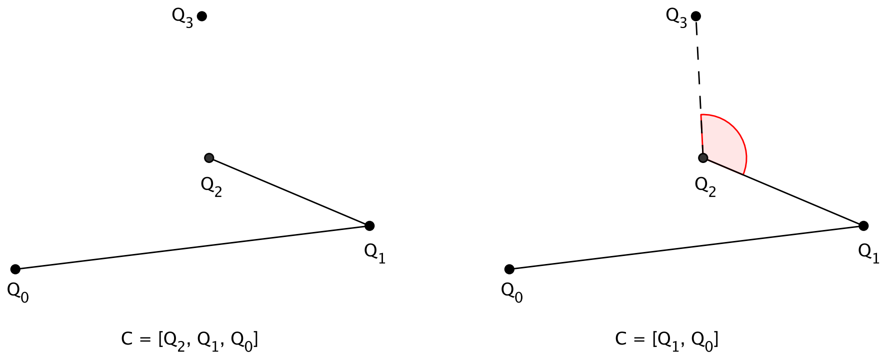


FIGURE 5.4: The proposed invariant is incorrect in this case, when Q_2 is popped because Q_3 lies to the right of directed segment Q_1Q_2.

5.2.2 Related Mechanised Proofs

Having made the case for the importance of formal proof in the realm of convex hulls, let us turn our attention to the contribution of our mechanisation in relation to other formal proofs of algorithms in the area.

The first such proof was published in 2001, by Pichardie and Bertot [143] using the theorem prover Coq. Like us, they were inspired by the work of Knuth [104], but we differ in choice of algorithms: Pichardie and Bertot look at an incremental algorithm and a package wrapping algorithm for finding the convex hull. As Graham's Scan algorithm is the algorithm commonly used in computational geometry[5], our proof of it is a useful and novel addition.

More recently, Brun, Dufourd and Magaud [25] present a formal proof of an *executable* program specification. They also chose the incremental algorithm, rather than the more efficient Graham's Scan, but what is exciting in their development is that an executable program can be generated automatically from the exact syntactic functional description for which they have constructed a formal proof.

A novel contribution of our work is to handle the case where points are collinear. Pichardie and Bertot, like Knuth, disallow collinear points, as do Brun Dufourd and Magaud. This allows their proofs to focus only on the more interesting cases, as we did in certain sections of our written exposition in Chapter 4; but without treating the special cases, the proof is of limited value to someone looking for confidence in the algorithm. (It's not much good having a guarantee that a bridge will stand up so long no three rivets are aligned!)

Pichardie and Bertot do make two partial attempts at treating the collinear case. In one attempt they propose perturbing the points to achieve the desired non-collinearity; as they comment, however, this leads to difficulties reconciling the case where a perturbed point becomes part of the convex hull.

Their other attempt proposes a mechanism for incorporating collinear points, much like our own, with the minor but significant difference that their formalism takes betweenness as a fundamental concept, expressed $\rceil pqr \lceil$ for q lying inside the segment pr. Collinearity is derived from this notion, as:

$$\cdots pqr \;\equiv\; \rceil pqr \lceil \vee \rceil qrp \lceil \vee \rceil rpq \lceil$$

It is our view that this is likely to complicate proofs, and that may be why only partial results for that line of research are reported.

We also note crucial errors in the statement of Pichardie and Bertot's axioms, $\forall pqr.\ \rceil pqr \rceil \;\Rightarrow\; \widehat{pqr}$, where $\widehat{pqr}$ is their notation for our $\circlearrowleft pqr$. By context it is clear that $\rceil pqr \rceil$ should be $\rceil pqr \lceil$; but the resulting lemma would be inconsistent. With some

[5]One reason why Graham's Scan algorithm is often preferred is that it is generally more efficient, $O(n \log n)$ time as opposed to $O(n^2)$ worst-case for the incremental algorithm.

thought it becomes apparent that the statement should read $\forall pqr.\ \rceil pqr \lceil \ \Rightarrow\ \neg\widehat{pqr}$. However the fact that obvious errors like this occur in other people's formal proof statements, emphasises the points in Section 5.1.2 of intuition being obscured and mistakes being made, in other people's work as well as in our own experience. We also note our own preference for conservative extensions to existing theories, instead of declaring fresh axioms as Pichardie and Bertot do, in part because mistakes introduced through erroneous axioms can be very tricky to spot.

One thing our work and theirs have in common is that the special cases can account for a huge amount of effort required for formal proof: formal proof is difficult. But without formal proof done carefully, as we have repeatedly seen, mistakes are easily made and overlooked. All too often the flaws lie in the edge cases that are so tempting to omit, as in the proofs of O'Rourke and Cormen *et al.*, and so we conclude: formal proof is useful.

5.3 Conclusion

It is clear that formally proving geometric algorithms in a theorem prover like Isabelle adds confidence in their correctness, and consequently we believe it should become an important stage in the development process of such algorithms. And it is not just algorithms which so benefit; recall from the discussions of Chapter 1, intuition can lead us astray in any domain, even mathematics. Despite the difficulty of constructing these proofs, we believe that by building libraries of useful theories and, crucially, by *improving the tools*, this task will get easier. The next chapter surveys related work which can help bring us closer to this ambition.

Chapter 6

Improving the User's Experience

In the previous chapter we showed how ambiguous statements and flaws in reasoning can be made easily in written expositions. Formal definitions eradicate ambiguities and mechanical proof constructions are far less likely to contain errors in the reasoning steps. The added guarantee of correctness which a formal development brings is without doubt beneficial. However, many mathematicians and software developers are currently reluctant to adopt this approach, for many of the reasons we outlined in Section 5.1. We believe that by improving the current theorem proving technology the user base would be dramatically increased.

This chapter will outline ways we believe the formal proof experience can be enhanced, with particular attention to mechanical geometry theorem proving. The possible improvements we consider include extending Isabelle's simp set for geometric problems, using automated geometry techniques available in other tools, and borrowing user-interface best practices from software engineering.

6.1 Automation in Isabelle

In Section 3.2.3 we described the automatic tactics and tools Isabelle provides, from the classical reasoner and simplifer, to Sledgehammer [20, 141], QuickCheck [17] and Nitpick [21]. In some situations, these will automatically provide a proof or indicate whether a statement is false. They may even find a counterexample. This powerful automation can go well beyond what one's intuition would notice and is one of the contributing factors to Isabelle's success. There is more that can be done here, however. For many of our goals, and for mistaken statements we have tried, these built-in automation solvers could not reason about our problem domain, or were thwarted by the complexity — nonlinear real arithmetic and higher-order logical constructs in particular[1].

One notable addition to Isabelle, which addresses this somewhat, is the proof method *sos* (sum of squares), which was mentioned in Section 3.2.3. This was introduced in 2009, which was unfortunately after we had completed our mechanisation of Graham's Scan. It may have helped in some cases to discharge some of the arithmetic goals or to inform us earlier if we had formalised the loop invariant incorrectly, but in work we have done since this time, we have found `sos` to be of only limited assistance.

In particular, in experiments we have run with the lemma *`transitivity`* from Section 3.5.4, *`sos`* did not complete after 12 hours. Even with a simplified version of the lemma involving only 4 points (8 variables) and maximum total degree 2, it was not able to complete within that time bound. The lemma involves many variables, several inequalities, and terms which are nonlinear combinations of multiple variables; these are all likely to be factors which contribute to *`sos`* being highly inefficient. These conditions of course correspond to more difficult lemmas we encountered in our proof, all of which were tedious to prove and where we would most like assistance. However, the availability of *`sos`* for many simpler problems is a big step in the right direction.

Solving our geometric lemmas by converting them into algebra is not the only way, of course. A more synthetic approach could be adopted instead. This would have the benefit of producing a more insightful and intuitive proof. The automation afforded by Isabelle's simplifier lends itself to this task. We discuss our work and experiments with this approach in Isabelle next.

6.1.1 Extending Isabelle's Simplifier and Classical Reasoner

Isabelle makes it easy for a user to encode some types of automation by enabling them to extend its simplifier and classical reasoner on demand. The rules which we found useful to add to the corpus used by *`simp`* and *`auto`* are presented next.

Simplification Rules

To a human mathematician the statement that three points are collinear is natural, without attention to the order of those points. However, in Isabelle the terms `collinear a b c` and *`collinear b a c`* are symbolically evaluated and interpreted differently.

[1]Sledgehammer has improved considerably since we completed our Graham's Scan proof and now offers more assistance. This is discussed further in Section 10.2.

One of the most tedious parts of our earliest proofs was dealing with this very issue; in order for a lemma to be applied or a goal to be discharged it was often necessary to compare and adjust the ordering of points manually. Thus, for our theory of planar geometry, the first and most obvious automation was the establishment of rewrite rules to express the geometric terms in a canonical form.

Developers can annotate proved lemmas, *e.g.* with the token "`[simp]`", to indicate that the standard simplifier should attempt to use that lemma as a rule. This allows expressions to be reduced to canonical forms in many cases; in some cases entire decision procedures can be encoded and goals proven automatically.

The following set of simplification rules allow our geometric predicates to be written in a canonical form automatically, removing much tedium (and in some cases proving goals automatically):

```
signedAreaRotate [simp]:   signedArea b c a = signedArea a b c
signedAreaRotate2 [simp]:  signedArea b a c = signedArea a c b
collRotate [simp]:         collinear c a b = collinear a b c
collSwap [simp]:           collinear a c b = collinear a b c
swapBetween [simp]:        a isBetween c b = a isBetween b c
leftTurnRotate [simp]:     leftTurn b c a = leftTurn a b c
leftTurnRotate2 [simp]:    leftTurn b a c = leftTurn a c b
```

Each rule is expressed as an equality, with the convention that the simplifier replaces the left-hand side of the equality with the right-hand side of the equality whenever possible. The rules above are known as permutative rewrite rules as each side of the equation is the same up to renaming of variables. It is worth noting that such rules can be problematic because once they apply, they can create infinite loops. However, Isabelle's simplifier is aware of this danger and treats permutative rules by means of a special strategy, called ordered rewriting: a permutative rewrite rule is only applied if the term becomes smaller with respect to a fixed lexicographical ordering on terms. Recognising this special status automatically is a very useful feature of Isabelle.

In addition to the above rules, we added several other proved rules to Isabelle's simplifier, not for the purpose of reducing to a canonical form, but in order to supply trivial facts automatically where needed:

```
areaDoublePoint [simp]:    signedArea a a b = 0
areaDoublePoint2 [simp]:   signedArea a b b = 0
twoPointsColl [simp]:      collinear a b b
twoPointsColl2 [simp]:     collinear a a b
notBetweenSelf [simp]:     ¬ a isBetween a b
notLeftTurn [simp]:        (¬ leftTurn a c b) =
                               (leftTurn a b c ∨ collinear a b c)
```

Despite these simp rules discharging many of our subgoals automatically, there exists a large collection of trivial subgoals that require manual proof. One such example is showing collinearity when we know a betweenness relation holds, *i.e*

```
isBetweenImpliesCollinear:   a isBetween b c ⟹ collinear a b c
isBetweenImpliesCollinear2:  b isBetween a c ⟹ collinear a b c
```

Of course, these facts are trivially proven by expanding the definition of `isBetween` but it is cumbersome for the user to always perform this step. Applying a general tactic is preferential, so we first tried adding these rules to Isabelle's simp set, assuming they would work as conditional rewrites. However, these rules together can cause Isabelle to enter an endless loop. While it is unsurprising that looping can occur, an unexpected difficulty in using Isabelle was in trying to understand why looping occurs in certain situations; even after inspecting the trace output the reasons for looping are rarely clear. This emphasises that care must be taken when extending the simplifier. The Isabelle manual advises that users should include only canonical simplifications, *i.e.*, only rules which are universally desirable, and while this is sensible in practice, it means that much useful control knowledge cannot be expressed as simplification rules.

There is however another means of easily automating the "conditional rewrites" in Isabelle: extending the classical reasoner rather than the simplifier. With this approach, we are able to automate the inferencing just described.

Conditional Rewrite Rules

In Isabelle, classical reasoning is different from simplification. While the latter is deterministic, classical reasoning uses search and backtracking in order to prove a goal outright using a natural deduction style of reasoning [134]. We can add rules to Isabelle's classical reasoner by marking them as introduction, elimination, or destruction rules. This gives a powerful automation framework alongside the default simplifier. Regretfully the Isabelle tutorial is somewhat vague on their use — distinguishing between them as follows: "Introduction rules allow us to infer new information ..."

Elimination rules allow us to deduce consequences." Isabelle further distinguishes between two types of elimination rules: where information may be lost by a rule's application, it should be marked as a destruction rule. We have found that the following rules are useful introduction rules for our problems:

```
notCollThenDiffPoints [intro]:
    ¬collinear a b c ⟹ a≠b ∧ a≠c ∧ b≠c
isBetweenImpliesCollinear [intro]:
    a isBetween b c ⟹ collinear a b c
isBetweenImpliesCollinear2 [intro]:
    b isBetween a c ⟹ collinear a b c
isBetweenImpliesCollinear3 [intro]:
    c isBetween a b ⟹ collinear a b c
isBetweenPointsDistinct [intro]:
    a isBetween b c ⟹ a≠b ∧ a≠c ∧ b≠c
leftTurnDiffPoints [intro]:
    leftTurn a b c ⟹ a≠b ∧ a≠c ∧ b≠c
onePointIsBetween [intro]:
    collinear a b c ⟹
        a=b ∨ a=c ∨ b=c ∨
        a isBetween b c ∨ b isBetween a c ∨ c isBetween a b
```

Another type of automation we wanted to implement was the identification of contradicting assumptions and subsequent discharge of these subgoals. This is a common problem in geometry theorem proving as case splits are often needed to identify the positioning of points relative to each other. This method of proving will generally introduce some cases which cannot exist. It is up to the human user to identify these cases and discharge them. This is not an easy task when there is an enormous list of assumptions; manually discovering which assumptions contradict and then finding the correct lemma to apply is difficult, not to mention mundane. To automatically find certain contradictions, we added the following destruction rules to the classical reasoner:

```
areaContra [dest]:
    [| signedArea a c b < 0; signedArea a b c < 0 |] ⟹ False
areaContra2 [dest]:
    [| 0 < signedArea a c b; 0 < signedArea a b c |] ⟹ False
notBetweenSamePoint [dest]:
    a isBetween b b ⟹ False
notBetween [dest]:
    [| a isBetween b c; b isBetween a c |] ⟹ False
notBetween2 [dest]:
    [| a isBetween b c; c isBetween a b |] ⟹ False
notBetween3 [dest]:
    [| b isBetween a c; c isBetween a b |] ⟹ False
conflictingLeftTurns [dest]:
    [| leftTurn a b c; leftTurn a c b |] ⟹ False
conflictingLeftTurns2 [dest]:
    [| leftTurn a b c; a isBetween b c |] ⟹ False
conflictingLeftTurns3 [dest]:
    [| leftTurn a b c; collinear a b c |] ⟹ False
```

Limitations of Adding Automatic Rules

The rules listed above for simplification, introduction and elimination remove a large amount of the low-level manipulation that was otherwise necessary when working in our theory.

Despite this automation being easy to implement in Isabelle, there is a specific limitation we wish to point out: the behaviour of the simplifier is rather opaque. Currently it provides a resulting proof state (or failure notification), and it can supply an extremely verbose trace of its activity. It does not provide a concise statement of which substitutions led to the resulting proof state. And—in the all-too-frequent situation where the simplification set includes potentially looping rules—it is very hard to determine which simplification rules are causing non-termination.

6.1.2 Tactics and GUI Support

If a user wants to add more sophisticated automation they can create their own tactics. However, this is not an easy task, for it requires one to be familiar with the underlying ML code for Isabelle. This codebase is large and fairly complicated, and it is not nearly as well documented as the more user-friendly end-user mode. The developers of Isabelle have recognised this drawback to the tool and in the last few years have written *The Isabelle Cookbook* [176], a tutorial on how to programme Isabelle at the ML-level. This has helped somewhat but it is still a non-trivial task to write your own tactics. In addition, once tactics have been written, the task of maintaining them so that they work with new releases of Isabelle can be painful.

If we were to create our own tactics to provide better automation for geometric reasoning within Isabelle, then re-implementing some of the successful techniques used in the field of mechanical Geometry Theorem Proving (GTP) could hold promise. We review some of these techniques in the following section.

6.2 Mechanical Geometry Theorem Proving

Two main approaches for mechanical GTP have evolved over the past few decades: coordinate free methods and algebraic techniques. In what follows we shall look at some of the major achievements made in each.

6.2.1 Synthetic and Coordinate Free Techniques

The coordinate free techniques focus on synthetic proofs, attempting to automate the traditional proving methods. Gelernter's Geometry Machine, which we mentioned in Section 2.2, was a pioneer of the coordinate free approaches. Following this, many researchers built geometric reasoners based on a purely synthetic approach. These include Nevis [130], Elcock [48], Greeno *et al.* [70] and Coelho and Pereira [38]. Systems following this technique have the desirable property that they produce human readable proofs, where a geometric interpretation can be attached to the arguments. Unfortunately, the approaches in practice tend to be extremely limited in the types of problems they can handle.

One leading technique which addresses this problem somewhat is the Area Method of Chou, Gao and Zhang [33]. This technique uses triples of points equated to the the signed area of the triangle they define, and expresses common geometric properties including collinearity, parallelism, and congruence as algebraic statements about these triples. For example, to express that point d is the midpoint of segment bc, we could write:

$$\mathit{signedArea}\ d\ b\ c = 0 \wedge \forall a.\ \mathit{signedArea}\ a\ b\ c = 2 \cdot \mathit{signedArea}\ a\ b\ d$$

Once a geometric question is expressed in terms of signed areas, its truth can be evaluated by reducing the algebra. The Area Method has motivated further techniques, such as the Clifford Algebraic Reduction Method [89] and the Full Angle Method [34], and has been used to prove a range of sophisticated theorems, including the well-known results of Ceva, Menelaus, Gauss, Pappus, and Thales. It has also been used by Fleuriot in his work to mechanise Newton's Principia in Isabelle [56], though he used the axioms interactively and not as the basis for an automatic decision procedure.

In keeping with the purely synthetic approaches, the Area Method and related techniques tend to produce human-readable proofs where the terms correspond to understandable geometric entities. However, these approaches are sometimes labelled as *quasi-synthetic* as even though intuitive interpretations are possible for each entity individually, the proofs can involve manipulating algebra.

Apart from the original implementation by the authors who proposed the Area Method, there have been three others: one implementation within the Theorema tool [160], one within the generic proof assistant Coq [129], and one within the dynamic geometry tool GCLC [92]. The authors of these implementations recently joined forces to write a paper describing the algorithmic and implementation details which were omitted in the original presentations [91]. Their paper also gives a variant of Chou, Gao and Zhang's axiomatic system, which is proved sound using the Coq proof assistant. This is a desirable result as the axioms of the original Area Method were never claimed to be minimal or complete, and in fact almost looked like an ad-hoc group of properties that had been discovered to be useful in many cases and hence asserted as primitives. The alternative axiom system is more likely to be rigorous.

As our proof of Graham's Scan reasoned so much about signed areas, we wondered if incorporating the Area Method into Isabelle would automate much of the reasoning we found tedious or difficult. After discussions with Narboux (who had formalised the method in Coq), we came to the conclusion that the method would not perform particularly well when there are many expressions involving betweenness, and would therefore not suit our purposes. Although we chose not to use this method for our work, we did borrow the concept of signed area to represent the notion of a left turn because it is commonly used in computational geometry too.

It is also worth noting that there are variants of the basic method which could reason better about inequalities. However, Janičić *et al.* state that, "these techniques are applicable only in special cases and not in a uniform way" [91]. Thus, it is not clear how they would perform with our theories. Interestingly, some work which has looked at extending the Area Method to better handle inequalities has used the Cylindrical Algebraic Decomposition (CAD) algorithm [39], one of the most powerful techniques for manipulating algebra. We will cover CAD and other algebraic techniques in the next section.

6.2.2 Algebraic Techniques

Following the spirit of Descartes, one obvious translation is to recast the geometric problems as algebraic statements about the coordinates of the points involved [2]. Proving the geometry can then be achieved by simply leaning on one of the many algebraic theorem proving techniques. Although this approach generates proofs which are difficult to relate to geometric intuition, the techniques can perform efficiently for many complicated problems.

Wu's Method [188], developed in 1977,[3] led to a resurgence of activity in geometry theorem proving. Wu's Method expresses geometric problems as a set of multivariate

[2]This is in keeping with how we proved many of our geometric lemmas from Section 3.5.

[3]It used to be the case that Chinese work rarely appeared in Western journals. Due to this, Wu's work did not become widely known until the next decade, when students of his emigrated.

polynomial equations, which it then solves using Ritt's concept of a characteristic set [157]. The technique performs well in many domains, but it does not perform well with *inequalities* which are essential to our theorems, expressing concepts such as betweenness and left turns.

Soon after, a similar technique was developed applying Buchbergers's theory of Gröbner Bases to solve geometric problems automatically by reducing them to Cartesian algebraic statements [95, 105]. Theoretically, this technique requires an algebraically closed field — such as the complex numbers — but in practice it can be used for reasoning about real geometry, with some caveats. Inequalities (such as $x \geq c$) are expressed as equations involving an additional variable ($x = c + a^2$): this increases the complexity of the problem statement, possibly causing solutions to take exponentially longer before they are found, and typically returning solutions which cannot easily be reduced from $\mathbb{C}$ to $\mathbb{R}$. For our purposes, the Gröbner Basis, like Wu's Method, is of limited benefit.

Decades before Wu's Method and the Gröbner Basis technique, Tarski outlined a theoretical decision procedure for statements over real closed fields including, for example, multivariate polynomial inequalities. Tarski's original *quantifier elimination* algorithm is difficult to understand and staggeringly inefficient both in theory and in practice, but since that time, a number of advances have led to the development of practical algorithms [167, 96]. The first such decision procedure to be implemented on a computer is the Cylindrical Algebraic Decomposition (CAD) method introduced by Collins [39]. The algorithm has since been refined and optimized [40], and implemented in the tool QEPCAD [151, 24] — quantifier elimination by partial cylindrical algebraic decomposition — which is today commonly regarded as one of the most powerful algebra solvers available.

More general-purpose computer algebra systems, such as Maple, Mathematica and MATLAB, now tend to have implementations of CAD and Gröbner basis, and have also been used for geometric theorem proving [52, 105]. In our experiments, however, QEPCAD performed significantly faster and was the only one capable of solving several of the geometric problems we came across in our proof of Graham's Scan. It is worth noting, however, that some CASs provide additional capabilities, including plotting and animation, which can also be useful in reasoning about geometry.

6.3 Combining Logic and Symbolic Computations

We believe it would be beneficial to combine the power of the algebraic approaches with the transparency and precise semantics of the coordinate free methods. In other words, being able to combine symbolic CAS with logic theorem provers is an attractive direction. This goal is certainly not new: the design of environments to combine several heterogeneous systems has been widely studied over the past decade and there is even a conference (Calculemus) dedicated for research in this specific area. We do not attempt to survey the breadth of concepts and implementations here, but will instead focus on related work combining TPs and CASs.

Calmet *et al.* distinguish three main approaches to how logical and symbolic computations can be combined [29]:

1. **Extending a CAS to enable deduction.** Notable examples of this approach are the Theorema project [26] and Analytica [36], both providing theorem-proving capabilities inside Mathematica.
2. **Implementing a CAS inside a theorem prover.** One example of this approach has been described in Section 3.2.3 where Gröbner bases is implemented inside Isabelle. There are numerous others, including a quantifier elimination procedure within HOL Light [118], a prototype CAS environment on top of HOL Light to ensure precise semantics [94], and Cayley algebra in Coq [61].
3. **Combining existing TPs and CASs.** Instances of this approach are illustrated in the integrations of Isabelle with Maple [13], HOL with Maple [83], Maple with PVS and QEPCAD [78], and the tool MetiTarski which combines Metis with QEPCAD [1].

The first two ways have the advantage that the issues of communication and common knowledge representations do not need to be addressed. However, these approaches often require a substantial amount of work. Re-implementing existing systems or decision procedures can be an enormous undertaking considering that, in some cases, these systems have evolved and been tuned over decades. We will focus on the third approach to combining logic and symbolic computations. This is due to the fact that for the complexity of the problems we are solving it would be challenging to reproduce the functionality and efficiency desired from existing tools within others. We will specifically look at how TPs can be integrated with external CASs, focussing on communication between systems (Section 6.3.1) and how the results from a CAS can be used in a formal proof (Section 6.3.2).

6.3.1 TPs Calling CASs

As mentioned in Section 2.2, TPs can be categorised as either fully automated or interactive[4]. We can also use these categories to describe how a TP invokes an external CAS.

Typically a CAS will be automatically called if it is integrated into a fully automated theorem proving (ATP) environment: the TP will check for certain patterns and if these are encountered the problem will be sent to a suitable CAS. The CAS will then perform some function, such as simplifying a polynomial or solving an equation, and send the result back to the TP to use. The first-order theorem prover Otter is one example of a proof system which calls out to external algorithms [117]. MetiTarski [1], mentioned previously, is another example, automatically solving inequalities involving real-valued functions such as `sin`, `ln` and `exp` by using QEPCAD to compare Taylor series expansions from within Metis.

Many of the projects which have combined CASs with interactive theorem provers have used a different approach for invoking the external tools: rather than the prover automatically choosing when to use a CAS, the user decides which problems look applicable for an external tool and they invoke it. This is usually done by writing a command in the proof script to tell the prover to communicate with the CAS. The HOL-Maple integration of Harrison and Théry [83] requires the user to invoke Maple by writing commands like:

```
# call_CAS "(((FACT 5)EXP2)-1)MOD(3EXP2)" 'SIMPLIFY';;
# call_CAS "(x*x)+(7*x)+12" 'FACTORIZE';;
```

As an alternative, it is possible that interactive TPs could run external tools automatically, similar to how ATPs do: background procedures could lurk, looking for patterns in the proof goal, and when triggered, send the problem to an external tool, only reporting back to the user if useful information has been gleaned. The information reported back could be a simplified formula which would make the remaining proof easier, or a counterexample which alerts the user that their subgoal is not provable. Running tools automatically and in parallel to the prover can be incredibly useful and is the approach which Isabelle's QuickCheck [17] and Nitpick [21] have adopted.

Of course, it would be possible for an interactive TP to offer both an interactive and automatic mode to invoke a CAS. This is in keeping with how Isabelle's Sledgehammer works [20]: it either runs automatically in the background or the user can turn this off

[4]Some authors include semi-automated as an additional category. We do not as most modern "interactive provers" are technically semi-automated, as they provide a significant amount of automation.

and only invoke it when desired. Ballarin's integration of Isabelle and Maple [13] also provides this dual approach, providing some specific commands the user can write to call certain functions in Maple and also allowing Isabelle's `simp` tactic to call Maple automatically.

6.3.2 Trusting an External Tool

Whenever a theorem prover is integrated with an external system, the issue of trust must be considered. One obvious mode of integration is to take the result from a CAS and use it within a proof inside the TP: in many areas, this is acceptable, just as one believes a calculator even though correctness of the hardware has not been verified. However this can impact confidence in the resulting proof, especially in situations where absolute guarantees of correctness are needed (*e.g.* when verifying safety-critical systems). As mentioned in Section 2.2.2.1, theorem provers typically have a simple kernel of primitive inference rules which are well-studied and believed to be sound. CASs, on the other hand, do not have this inspectable core and can return imprecise solutions [83]. In addition, different systems being integrated will typically use different representations, and the two-way translation introduces additional unreliability.

Despite these issues, the sheer practicality of using CASs has motivated several integrations which assume the CASs soundness. As Ballarin & Paulson note:

> Computer algebra systems also contain implementation errors. Depending on how rigorous one wants to be, one can reject any result of a computer algebra system without formal verification in the prover. Considering the amount of work ... we decide to live with possible bugs [14].

Their integration of Isabelle with the CAS Sumit uses Isabelle's *oracle* mechanism to introduce new assumptions, based on equivalences shown in Sumit and taken on faith [14]. MetiTarski takes a similar pragmatic approach, trusting QEPCAD's comparison of polynomial equations [1]. In general, one knows which areas of a CAS are reliable, but surprising bugs are occasionally found. In our experiments with QEPCAD, we encountered one where the solution space was pruned too aggressively and equations were being reported as irreducible. The maintainers fixed this very quickly once we reported it, releasing v1.42, demonstrating how committed they are to the tool's correctness, but as we have shown in Sections 1.3 and 5.2.1, it is not hard to make mistakes when formal proof is not used.

For some types of problems — and especially in safety-critical domains — the increased confidence given by a mechanically verified proof is essential. Taking the result of a CAS on faith is not an option. However, there are still several ways in which external tools can be useful to theorem provers without compromising soundness:

- sanity check a line of reasoning;
- guide a user's understanding; and
- guide a prover's automated proof construction.

Isabelle's QuickCheck and Nitpick are examples of the first of these, where untrusted systems (or code) may be used to find counter-examples to alert the user if it looks like their proof is going in the wrong direction. Examples of the second category include integrations which produce candidate proof plans [99], or visual diagrams [185] aimed at the user, to assist them in constructing or understanding a fully formal proof. Sledgehammer is an example of the third, as it calls out to first order ATPs then reconstructs their proofs inside Isabelle.

The integration of HOL with Maple [83] is another instance of the third category. The authors make the nice observation that it is often easier to search for a solution in a CAS and then check it in a theorem prover; this is put into practice by using Maple to generate factorizations which are then proven internally in the theorem prover HOL.

Motivated by some of this previous work, we raised the possibility of extending QEPCAD so that it could return a witness for existentially quantified formulae. Brown, the maintainer of QEPCAD, was happy to implement this straightforward addition to the tool, allowing it now to be used in a variety of trusted ways: as a counter-examples generator (to sanity check), as a tool for producing examples (to guide a user), or as a tool for providing instantiations (to guide formal proof).

6.3.3 Without Loss of Generality

After some experiments with manually translating geometric lemmas and sending them to QEPCAD, we concluded that it would be a useful CAS to integrate with Isabelle. It would offer some automation which Isabelle lacks — that of reasoning about nonlinear arithmetic. That said, however, we also noted limitations with the tool. Some of the queries we sent to it exceeded reasonable time- and/or space-complexity: either it ran out of memory or hadn't terminated after 12 hours. One of the problems which caused it trouble was:

```
segExtensionStillIntersects:
    X isBetween A B ∧ straightEdgesIntersect e {X,B} ⟶
                               straightEdgesIntersect e {A, B}
```

where `straightEdgesIntersect` is defined as:

```
definition straightEdgesIntersect ::  "[edge, edge] => bool"
    where "straightEdgesIntersect ea eb ≡
        ∃a1 a2 b1 b2.  ea={a1,a2} ∧ eb={b1,b2} ∧
            ( ( {a1,a2} = {b1,b2} ) ∨
            ( (b1 isBetween a1 a2) ∨ (b2 isBetween a1 a2) ∨
              (a1 isBetween b1 b2) ∨ (a2 isBetween b1 b2) )
            ∨ ( leftTurn a1 a2 b1 ∧ leftTurn a2 a1 b2 ∧
              leftTurn b1 b2 a2 ∧ leftTurn b2 b1 a1 ) )"
```

With geometric intuition, it is easy to convince oneself that this lemma is translation invariant: it is true if and only if the problem is slid in the plane such that one point is the origin. If we perform this translation on the intersecting edge problem and send the revised one (in 8 variables, instead of 10) to QEPCAD, then it yields a result in 4 seconds!

This problem is not unique, and translation invariance is a common property used to justify proving geometric theorems where "without loss of generality" (WLOG) one point is the origin. Unfortunately Isabelle does not have a WLOG tactic. A recent development within HOL Light, however, has seen the introduction of a WLOG tactic [81]. This tactic reasons about many situations in mathematical written proofs where the WLOG is commonly found, including geometry. We have since extended our Isabelle theory of geometry to simplify the reduction whereby one point is taken as the origin:

```
origin ≡ Abs_point ( 0,0 )
negative a ≡ Abs_point ( -(xCoord a),-(yCoord a) )
translatedBy a Δ ≡ Abs_point (
    (xCoord a + xCoord Δ), (yCoord a + yCoord Δ) )
```

We prove that the origin is equivalent to a point negated by itself:

```
originTranslated: origin = translatedBy a (negative a)
```

And then subsequently prove:

```
signedAreaTranslates: signedArea a b c = signedArea
    (translatedBy a Δ) (translatedBy b Δ) (translatedBy c Δ)
leftTurnTranslates: leftTurn a b c = leftTurn
    (translatedBy a Δ) (translatedBy b Δ) (translatedBy c Δ)
isBetweenTranslates: a isBetween b c =
    (translatedBy a Δ) isBetween
        (translatedBy b Δ) (translatedBy c Δ)
```

With these lemmas it becomes straightforward to show that propositions in our theory of planar geometry which involve a point (x, y), are equivalent to the same proposition translated by $(-x, -y)$; simplification rules then yield the proposition with one of the points being the origin.

Armed with this synthetic approach to formally translating a planar geometric problem into one with fewer variables, one can prove that sending a translated query to QEPCAD is sound. The algebraic solver QEPCAD can then be used to solve the problem more efficiently.

6.4 Proof Engineering

Recognising that no automation is a cure-all, let us look at a very different way of alleviating some of the difficulties set out in Section 5.1, based on the observation that software engineers routinely face many similar challenges to those of the users of TPs. Almost anyone who writes source code, in almost any language, is familiar with dealing with large libraries, remembering unfamiliar names, aligning variables with parameters very precisely, and maintaining code over time — just like those of us doing interactive proof. In the software engineering world, highly sophisticated tools and techniques have been developed and combined in *integrated development environments* (IDEs) to help with these challenges. Compared with the IDEs used by the modern software engineer, the tools for constructing formal proofs are generally very poor. Some researchers have been advocating that to rectify this, TP environments should borrow the ideas from IDEs. Since this idea was proposed more than two decades ago [108, 173], there have started to be some notable successes:

- Proof General builds upon the emacs interface for several theorem provers [7].
- CtCoq provides a UI for Coq based on the generic environment Centaur [18].
- PR — used for verification of reusable software components — uses IDE techniques to enable the reuse of abstract proofs and specifications [31].

However, much of this work has not kept pace with advances in IDE technology. Things have dramatically improved for the software engineer: emacs, for example, offers barely any of the graphical interaction features common in modern development environments. In addition, the importance of making UIs for TPs benefit from the new IDE technology is now becoming widely acknowledged. The blossoming field of *proof engineering*, like software engineering before it, studies how the engineering process can be improved by designing tools in ways that best suit both the domain (theorem proving) and the users (humans).

6.4.1 Modern IDEs

Modern IDEs can aid the software engineer in developing, deploying and managing software across its lifecycle. Commonly, they provide the functionality for:

- syntax highlighting,
- project building,
- refactoring components which help improve design by making large-scale structural changes easy,
- visually representing hierarchies, dependencies, and other relationships,
- automatic documentation lookup,
- completion of identifiers,
- integration with version control and build systems,
- context-aware search which finds related definitions, and
- content assistance through mechanisms which insert declarations and instantiations, and which can sometimes provide quick fixes to common problems.

It is easy to see how many of these can address the difficulties we noted in Section 5.1. If these functionalities were available when using an interactive TP, the burden currently placed on the user — when constructing or attempting to understand a formal proof — would be greatly reduced. By using modern IDE environments *as the GUI for theorem proving*, many of these capabilities can be directly applicable [127].

6.4.2 Proof General Kit

Inspired by the proof engineering approach, the developers of the emacs Proof General UI, noted above, decided to embark on an ambitious project to modernise and replace their tool in 2004. The updated framework is called Proof General Kit (PG Kit) [8], and aims to provide a flexible environment for managing formal proofs across their lifecycle: creation, maintenance and exploitation. The modern IDE Eclipse [46] is used as its foundation.

Eclipse is one of the most popular IDEs in use today, and a natural choice for several reasons. It is entirely open source, originally developed by IBM as a Java development environment, but it has since been extended by thousands of developers to provide an extremely rich set of IDE functionality for a wide range of languages. It exposes a customisable plugin architecture for working with an extensive spectrum of systems and projects, allowing a user easily to extend the framework by writing appropriate plugins for their domain.

Eclipse Proof General works as a plugin to Eclipse. It has so far been developed to communicate with the theorem prover Isabelle, and provides many of the IDE facilities people expect, including multi-file hierarchy, go-to-definition, show usages, hover-help, and auto-completion. See Chapter 7 for more details on this infrastructure.

6.4.3 jEdit Isabelle

Isabelle/jEdit is a younger project also aiming to improve the UI of the theorem prover Isabelle [180], building on jEdit [93]. Although jEdit is essentially a text editor and not an out-the-box IDE, it does provide a rich plugin eco-system, where a user can combine and configure plugins to make jEdit behave like an IDE. jEdit has gained some popularity as an IDE type tool as it is faster and lighter weight than many of the full-fledged IDEs such as Eclipse. However, jEdit is not as mature as Eclipse and as a result it lacks refactoring capabilities (and doesn't have access to the rich abstract syntax tree model in Eclipse). That said, however, Isabelle/jEdit does currently build a theory file dependency tree and offer go-to-definition capabilities. The big contribution of Isabelle/jEdit is that it provides a framework based on *document-oriented prover interaction*, where the PG command line mode of interaction is replaced with a continuous update model which allows the user to write their theory files without waiting for commands to process or locked regions to unprocess. Although the document-oriented model is a good idea, the actual implementation is still a work in progress, and we believe that despite it being usable, it needs significant refining before it becomes useful. In 2012, jEdit/Isabelle became the official UI for Isabelle, despite it still being in the development stage.

As a system, jEdit/Isabelle shares some of the same goals as the PG Kit project, but is more tactical in ambition and limited in scope. One of the aims of PG Kit is to to provide development environments for a whole class of interactive provers: Aspinall *et al* believe that the reason why the facilities of the IDEs took so long to be provided for TPs was due to the fragmentation of the community across so many different systems, diluting the efforts available. By investing in shared tools as much as possible — leaving only the underlying logical proof engines as separate, distinct implementations — the TP community can progress much more swiftly [8]. PG Kit also has the ambition to be a vehicle for research into the foundations of such an environments, with an explicit goal of advancing proof engineering. We feel that jEdit/Isabelle, by being more tactical in just being an editor for Isabelle, and by selecting a decidedly less popular

IDE framework, will not maximize the long-term benefit that proof engineering has the potential to bring. [5]

6.4.4 Library Lookup

Let us look in depth at how proof engineering can help with one of the most time-consuming of the tedious and disruptive tasks we noted in Chapter 5: the "library lookup" problem. Recall from Section 5.1.1 that Hales estimates that 50% of his time on the Flyspeck project is spent finding lemmas and using them appropriately. Our own experience is similar, with the knock-on effect of disrupting the flow of our proof; we would frequently be tracking several terms in a goal when we found it necessary to find a particular lemma in order to see the order of terms therein for matching. A context switch from an involved formal proof to the voluminous output from *`grep {.,*,*.*}/*.thy`* can easily shatter a hard-won understanding of the proof state.

One of the simplest ways a modern IDE helps is the "Go to Definition" feature, part of standard Eclipse and implemented for Isabelle in both Eclipse PG and jEdit. By control-clicking on a lemma, one is taken straight to where the lemma is defined. If we have a question about the order of terms, we can resolve that and be back in our proof (with one more keystroke) in less than a second. If our question regarding a lemma is more involved, we can read the comments and documentation in the containing theory file and search for other examples of where the lemma is used, all without leaving our environment. (We can even arrange the widgets so that our proof state remains visible the entire time.)

With "Hover Help" it is even easier: by placing the cursor over a lemma used in our proof, in Eclipse PG, the definition appears in a pop-up box, refreshing our memory about our own proof or helping us understand someone else's proof. "Autocomplete" is another standard IDE feature, where a keystroke (*e.g.* ctrl-space) will show us potentially applicable completions given the first few characters of a term. Where we knew the start of a lemma but were unsure about its entire name, or whether it used underscores or camel-case, this simple UI trick saves a surprising amount of time. This can also be used to disambiguate between lemmas, as its definition can appear as switch between completion proposals. This is useful where there are several lemmas with similar names, such as when describing slightly different cases. On the implemen-

[5] Since embarking on our research the PG Kit project is no longer actively maintained, but we hope it will be reignited one day for the advantages mentioned.

tation side, this functionality comes for free in the Eclipse IDE framework once the available lemmas are supplied by the Eclipse PG extension.

UI Plugins and FeaSch

Another advantage of using a modern IDE is that further UI extensions can be developed and contributed in the form of plugins. One such plugin which brings a powerful new approach to the library lookup problem is FeaSch [84]. Taking ideas from cognitive science, FeaSch is a theory and a system available for Isabelle and Eclipse ProofGeneral. *Features* — a type of tag in FeaSch capable of taking predicate-style arguments — can be defined as part of theories and then detected at runtime where those theories are used. These features are then used as the basis for automatically proposing lemmas to apply, presented in the Eclipse PG IDE alongside the proof. It can complete the instantiations automatically, saving time and distraction, and apply rules automatically, if enabled, when the feature cues are strong enough.

One domain where FeaSch has been successfully applied is integration: it can perform evaluation of integrals fully automatically, or, in semi-automated mode, it proposes familiar techniques (such as *`integrationByParts`* or *`uSubstition`*, corresponding to lemmas internally in Isabelle) based on the presence of predefined features. The feature detection process identifies arguments from the proof state which are used to instantiate the variables within the chosen techniques, allowing the user to guide the proof at a familiar level without being distracted by the minutiae of low-level steps or precise instantiations. By focusing on usability in this way, FeaSch is a good demonstration of how powerful the proof engineering approach can be.

6.5 Summary

To summarise, this chapter has described some of possible solutions for tackling many of the difficulties we described in Chapter 5. These have ranged from extending Isabelle's simplifier and classical reasoner, to writing tactics and integrating with external tools. We have also presented the emerging field of proof engineering which could help with many of the difficulties at the UI level. The table in Figure 6.1 summarises the difficulties and prospective solutions.

We can see from the table that many approaches have been tried, but nevertheless there is a long way to go. We also observe that there is some fragmentation, with some automation restricted to some systems, user interfaces, and/or domains. In particular, the automation available for Isabelle not geared towards non-linear arithmetic and geo-

Difficulties Using Isabelle	**Existing Solutions and Prospective Ideas**
Proving Minute Details	automatic tactics and techniques (`simp`, `auto`, `blast`, `arith`, `sos`), Nitpick, Quickcheck, and Sledgehammer; integration with CAS; *all however have limited applicability at present*
Library Look-up	IDE navigation; FeaSch
Entering Correct Instantiations	Isabelle tactics, FeaSch
Refactoring	very little available (can learn from software engineering and build on IDE support)
Version Incompatibilities	very little available (can learn a lot from software engineering methodologies)
Intuition Obscured by Opaque Presentation	Isar; can learn from IDE support for expanding/contracting sections
Distracting Minutiae	Isar, FeaSch; automation for proving minute details (listed above)
Expensive Mistakes caused by Obscured Intuition	Nitpick and QuickCheck providing counterexamples; *limited applicability however*

FIGURE 6.1: Approaches to improving the formal proof experience

metric problem solving, apart from `sos` which is too slow for many practical problems. If mathematicians and software developers are to embrace formal mechanised proof, we must make the developer of the proof much more productive. A lot can be learned from what IDEs have done for software engineering, in large part made possible by designing the IDE to support multiple languages. A cohesive, multi-prover system seems the most promising avenue for this work, with one of the biggest gaps being the ability to import the power of computer algebra systems. In the next chapter we look at how this could be done within a proof engineering architecture.

Chapter 7

The Prover's Palette

We believe that one major boon to developing a formal proof in today's proof assistants is to have seamless access to the power of a multitude of tools. Whilst there have been significant advances in combining provers with external tools, as described in Section 6.3, we believe the increased complexity of current proof developments places new demands and challenges on tool integrations. We propose that the proof engineering model, which learns lessons from software engineering, can yield a richer and more powerful framework for integrating systems. The key differentiator of our approach is that it is centered around the user, giving them control and visibility of many facets of the task at hand.

In this chapter we describe the design principles which we hypothesize can steer a good multi-system framework for interactive theorem proving. We then present the Prover's Palette, an architecture for such a framework, looking both at the GUI elements and the underlying implementation.

7.1 System Design

We begin with the premise that an integration's primary goal is to accelerate the proof development process. To date, this has been achieved primarily by integrations which can automatically simplify expressions and discharge subgoals [126, 175]. With more complicated verification tasks — and with more mathematicians using provers — we believe that the process of formal proof can also benefit from tool integrations which are able to enhance a user's understanding of a problem. To achieve this, it is important that integrations support multiple modes of interaction: the framework should support automatically configuring settings appropriate to a specific problem but also emphasise usability by enabling users to explore a problem domain easily, all while maintaining consistency between systems. We believe that a semi-interactive integration framework now has a vital role to play.

The design principles which govern a systems integration of this nature are presented next, followed by a description of what we believe the user's experience should look like.

7.1.1 Principles

We have taken the following short list of design principles as the overriding aims in designing this semi-interactive systems integration:

- **Automatic.** The use of an individual tool should be simple and tightly integrated with the proof being developed: this means the tool is presented through GUI components in the same development environment, with communication between the components fully automated, and commonly used capabilities available with at most a single click. A novice user can then benefit from a variety of mathematical tools, even if he is not familiar with them.
- **Interactive.** As much as possible, the full functionality of a tool should not be denied to the power user. Exploration of the problem should be facilitated, such as through suggestions for massaging the input to be more amenable to the external tool, GUI widgets allowing operating parameters to be adjusted, and finally the ability to edit the instructions sent to any tool (which can be essential for some problems, as even the best fully automated integrations cannot always tune the parameters appropriately).
- **Inspectable.** Where multiple tools are involved in the validation of a proof, the commands should be explicit and repeatable. Integrations should give careful consideration to how the output of a tool might be used, offering multiple modes in some cases, and where the output is being trusted, the full commands required to reproduce the result should be supplied.
- **Modular and extensible.** In light of the myriad systems presented in Chapter 2, any long-term viable systems integration must be designed so that new tools can be incorporated with as much ease as possible. Code should lend itself to use as a framework, where modules of functionality — whether for parsing, massaging input, or different modes of using output — can be reused.

We have been heavily influenced by two well-known sets of design principles. Schneiderman [169], looking at usability of software, says that a user interface should:

(1) Strive for consistency
(2) Cater to universal usability
(3) Offer informative feedback
(4) Design dialogs to yield closure
(5) Prevent errors

(6) Permit easy reversal of actions
(7) Support internal locus of control
(8) Reduce short-term memory load

Engelman [50], designing a computerised mathematical system, advocates the following properties (abbreviated from Section 2.1):

(1) Capable of ordinary numerical computation
(2) Support a wide spectrum of symbolic computations
(3) Simple to use for a novice
(4) Customisable by an expert
(5) Extensible
(6) Responsive

7.1.2 The User's Experience

Let us illustrate how we envision a user's experience with an integrated prover-tool development environment, in a way consistent with these design principles. We begin with the observation that proof IDEs tend to have at least two widgets (window regions, or "Views" in Eclipse terminology) already — an editor where the proof script is written, and an output pane where the proof state is shown — and sometimes other widgets for other functionality, such as a proof outline, assistance, or search. As such, we believe a View widget is a natural way to integrate an external tool inside a modern IDE in an interactive way.

Figure 7.1 illustrates how such an external tool widget could work. Like a proof state output pane, this widget is updated when the proof subgoal changes. If the widget is open (shown by the "Widget" arrow), a **Start** tab can show the current subgoal and tool options available. Alternatively, a user could run in automatic mode (shown by the "Pop-Up" arrow) where the tool widget is hidden, running in the background, and it reveals itself when the tool is able to find a useful result, showing the **Finish** tab. The various user flows through the wizard are shown by the other arrows in Figure 7.1, and will be discussed below. If a result is produced by the external tool, the widget presents the user with the choice to apply it back in the proof script.

In interactive ("Widget") mode, the **Start** tab is the entry point, displaying the current problem in the prover's language for consistency with the proof environment. If the problem is cleanly applicable to the tool, a button similar to Google's **I'm feeling lucky!** gives a one-click mechanism to send the command to the external tool and bring up the **Finish** tab (discussed below). If a problem cannot be automatically sent, the

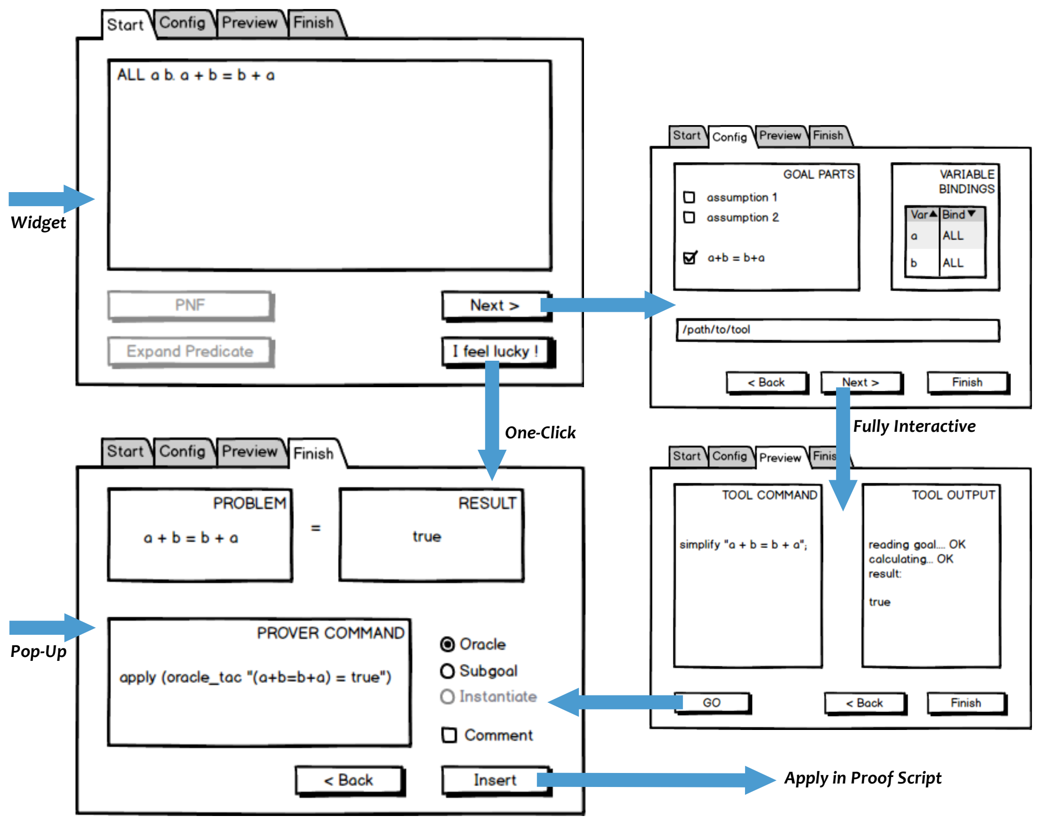


FIGURE 7.1: Sketch of tabs and user flow through them

framework will propose ways of massaging it to make it more amenable to the external tool, such as converting to prenex normal form (PNF) or expanding the predicates which are unknown to the external tool.

For richer interactivity, the user can click **Next** on the **Start** tab to advance, wizard-style, to one or more **Config** tabs where advanced integration options are available for manual adjustment. Much of the functionality here will necessarily be tool-specific, but we note there are some options which can apply to many tools, such as selecting specific parts of the proof state to work with, specifying the quantification and binding of variables, and specifying a command to run.

From the **Config** tab the user could **Finish** immediately, or proceed to a **Preview** tab displaying the I/O with the external tool. Here, the user could edit the problem or commands as sent to the tool, and click **Finish** to go to the results tab; or, in an extremely interactive mode, they could click **Go**, watch the output, change the input, and run again — repeating, with a very short round-trip time, until they have finished experimenting.

Finally, on the **Finish** tab — available from most of the other tabs in interactive usage and appearing automatically in "Pop-Up" mode — the result of the external tool's computation is displayed. This might state that a proposition is true, simplify an expression, or find a witness; options for how to use a result are available, and an **Insert** button provides a way to insert and run commands in the proof script. In other cases, the tool might indicate that a proposition is incorrect, alerting the user to halt the proof attempt, or it might provide other insight into the proof state; for these cases, no proof script commands would be shown and the **Insert** button would be disabled.

7.2 System Architecture

Having described our initial vision for a user-centric tool integration, let us now present the framework we have developed along these lines. We call this framework "The Prover's Palette", and we will outline its architecture and key parts of its implementation. The code for the tool is available online at `https://github.com/limeikle/provers-palette`.

As we have noted, the main idea behind our approach is to unify multiple tools in a cohesive, extensible UI. To this end, we expect each new tool to come as a plug-in to the IDE. The core Prover's Palette framework is a code library which facilitates development of these tool plug-ins, reducing the effort required to integrate with a new tool, and ensuring a consistent experience across tools so integrated. We have sketched an idealised user experience in the previous section; let us now turn our attention to the re-usable primary components which the framework should supply to support tool plug-ins, listed below and shown graphically in Figure 7.2.

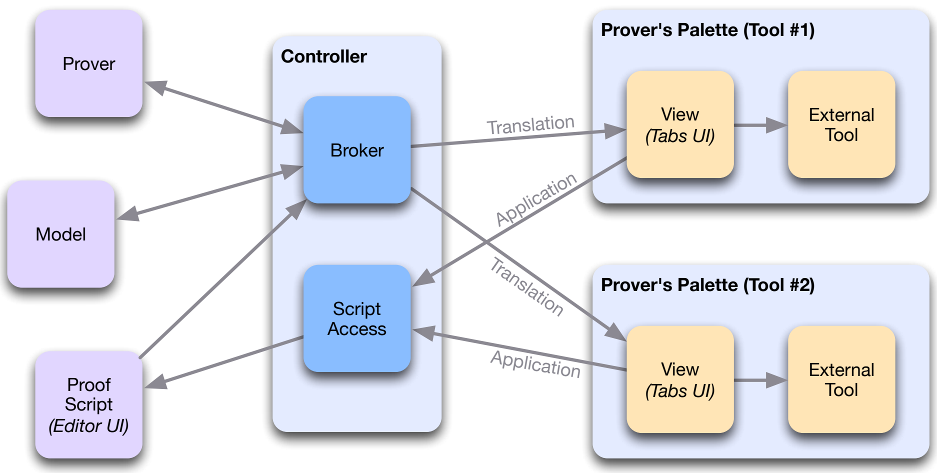


FIGURE 7.2: The Prover's Palette Architecture

GUI: Much of the process flow for a user configuring and using a tool through the GUI follows a common pattern, as we have sketched. The framework should supply abstract superclasses for the graphical View definition (the widget where a tool sits inside the IDE), for the individual tabs which comprise the View, and for many of the GUI elements which populate the tabs (*e.g.* the **Next** and **Back** buttons). This is discussed further in Section 7.2.1.

Controller: The **Start** tab must be updated when the proof state changes, and the **Finish** tab must be able to effect changes to the proof state. Therefore at the heart of the framework must be an element which can notify a tool plug-in of proof changes, modify a proof script at the request of the tool plug-in, and prompt the prover to process changes to the proof script. Let us call this element the "controller" or "broker", and record its primary responsibility to broadcast and orchestrate changes to the "model" (the proof script, in our case) among one or more "Views", in accordance with the common model-view-controller paradigm. This will be presented in Section 7.2.2.

Translation: As different tools and provers typically use different representations, the translation between them is an important aspect of the framework; the core cannot do all the tool-specific translation, of course, but it can do a lot to make the tool-specific obligations small and reliable. We describe this in Section 7.2.3.

Application: We have previously noted that there can be many different ways one might wish to use results from external tools: this is a key reason for wanting to design our framework in a user-centric way. The final major architectural concern is the encapsulation of patterns for applying results from a tool back into the proof context. This will be discussed in Section 7.2.4.

The following sub-sections describe the important implementation decisions we faced in building the Prover's Palette, split up according to these four areas. Readers may wish to skip these sub-sections if they are not interested in the underlying implementation details; it is presented here in part for completeness and as a guide for people wishing to implement their own tool integrations using our framework.

7.2.1 The GUI

We have noted that the GUI is presented as a "View" widget: specifically it is an `org.eclipse.swt.View` in the Eclipse Standard Widget Tooklit (SWT) [47]. This is a resizable, dockable, minimisable component which can contain other graphical widgets. For the Prover's Palette, the View contains the tabs we have sketched, described in the following sub-sections, and a toolbar which exposes overarching functionality, described in the final sub-section below (Section 7.2.1.5). We describe the reasons for the choice of Eclipse and Java in Section 7.2.2.

Before describing the tabs, let us reiterate that the core Prover's Palette framework supplies template implementations for each tab, as abstract classes. This is a standard way to maximize the capacity for code reuse and minimize the effort required to onboard new external tools, and one which contributes to a consistent experience across different tool implementations. Individual tool plug-ins need not use all the tabs here, and they may introduce new tabs, but we believe the set introduced here provides a good starting point.

The framework supplies the abstract class *ProversPaletteViewPartAbstract* as an entry point where tool plug-ins define the View, populating the tabs and performing tool-specific initialisation. The tabs themselves are rooted in the framework class *ProversPaletteTabCompositeAbstract*, which supplies behaviour common across nearly all tabs in the Prover's Palette:

- Creating the tab within the View
- **Next** and **Back** buttons, and their associated methods and behaviour, for "wizard" style use
- A **Finish** button on all tabs except the final tab, which encourages a design whereby the user can skip all the interactive steps and run in automatic mode, where applicable (note that the tool-specific subclasses are responsible for supplying the enablement logic)
- Listener support to tell when any fields on a tab have changed (either programmatically or manually), thus indicating when subsequent tabs require updating
- Accessing the proof context, including translators to and from the prover's notation (Section 7.2.3)

This class is the parent of the tab classes presented below, and it is recommended that this also be used as the superclass for any new tabs introduced for a tool which do not fit any of the more specific tab classes provided. As the tab-specific classes discussed below are still abstract — with tool plug-ins providing the concrete subclasses — we show a mock-up of the tab (from Figure 7.1) for an indication of how it may look.

7.2.1.1 The Start Tab

The **Start** tab (Figure 7.3) displays the problem from the proof script and the actions which are valid on that problem: the **Finish** button will typically be enabled if the problem can be sent in its entirety without any changes (as determined by calls to the translation module); the **Next** button is enabled if there is a problem; and various pre-

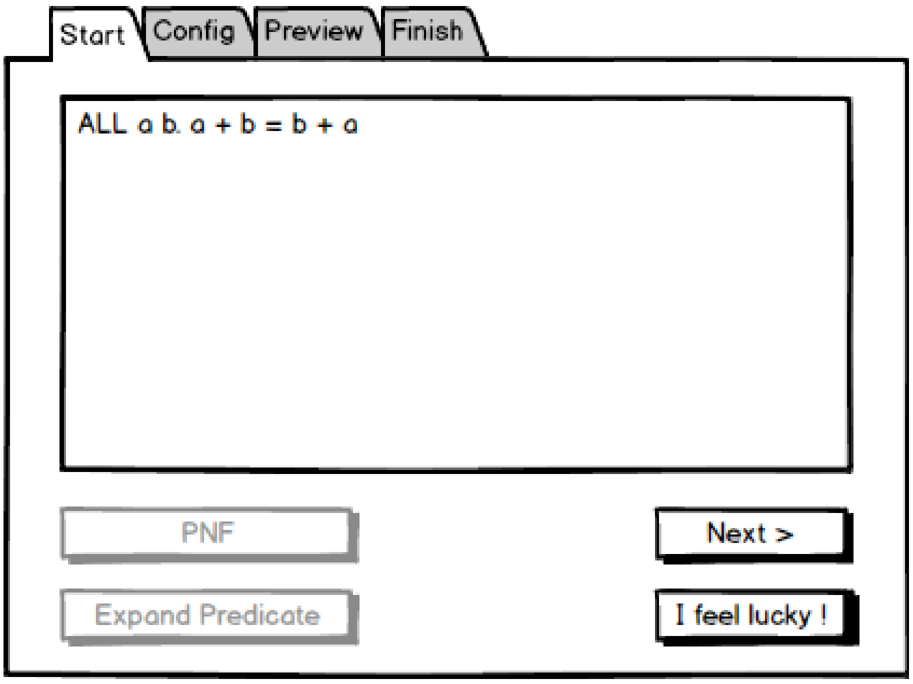


FIGURE 7.3: The Start Tab

processing actions may be exposed if the tool plug-in indicates that a problem would benefit from pre-processing.

The core framework provides the abstract class `StartTabComposite` supplying this behaviour common in this tab. Most significantly, it provides a public method `updateWithProblem` which can be invoked (by the broker, discussed in Section 7.2.2) to display the current problem whenever the proof state changes. The **Next** and **Finish** buttons are inherited from the parent class (`ProversPaletteTabCompositeAbstract`), with the expectation that concrete tool-specific subclass will supply the logic for whether **Finish** is applicable, that is where there is an obvious automated way to send the problem to the tool in a single click. For common pre-processing activities, such as converting the current problem to PNF or expanding predicates (which will be described in Section 7.2.4.1), the buttons and hooks to trigger this in the prover are also supplied here. Finally, in cases where there might be multiple input sources (provers), a dropdown `proverChoice` is supplied here allowing different input sources to be wired up.

7.2.1.2 Configuration Tabs

Due to the breadth of tools which we would like to support in the Prover's Palette, the **Config** tab shown in Figure 7.4 is a placeholder for what will typically be one or

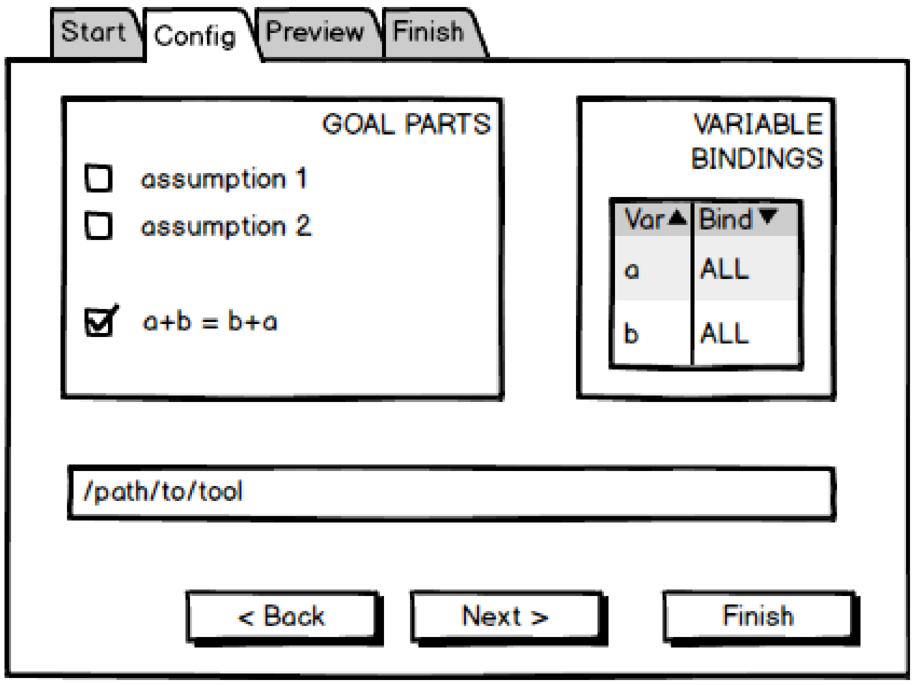


FIGURE 7.4: A Sample Config Tab

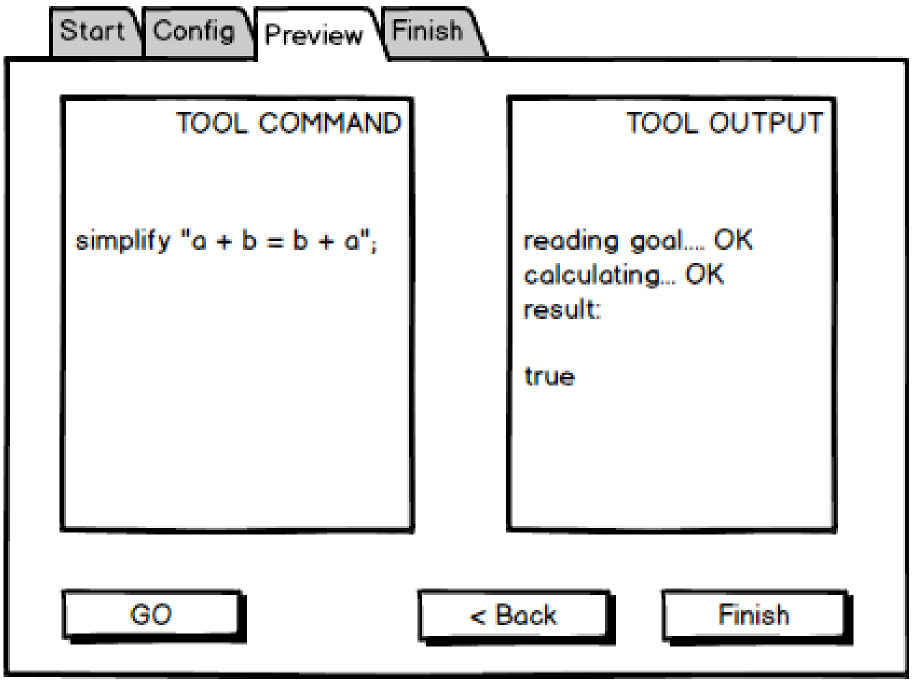
more highly tool-specific tabs where its interactive capabilities are exposed. Although a casual user might initially skip these, an important design goal is to ensure that expert users have access to many of the advanced features of a tool, even if they cannot be automated; our own experience with several such tools indicates strongly that being able to adjust the configuration is essential for exploration, and powerful in general.

The content of these tabs will vary dramatically from tool to tool, and tool plug-ins may often subclass `ProversPaletteTabCompositeAbstract` directly. We have, however, identified two types of configuration tab for which we can see general applicability, and these are provided as part of the core Prover's Palette framework.

The **Import** tab disects the current proof subgoal into its assumptions and conclusion, and allows the user to easily inspect which parts are compatible with an external tool (based on tool-specific logic provided in the translator). This tab lets a user select or deselect components of the subgoal to include in the problem sent to the external tool. This behaviour is defined in the class `ImportTabComposite`, along with code for extracting the variables from a problem, showing which variables are used in the selected parts, and editing variable quantifications and types. (In addition, the **Back/Next/Finish** buttons are inherited from the parent class; as this is the case for all tabs, we will cease mentioning them.)

The **Problem** tab swaps semantics from the prover, used in the **Start** and **Import** tabs, to that of the external tool; it presents the problem statement which will form the basis of input to the tool, based on components selected in the **Import** tab, but now translated to the language of the tool. It allows editing of the problem in case the user knows better than the automation or simply wants to play with the tool, and supplies conveniences for tool-specific subclasses to add further configuration, such as selecting a mode to run the tool or adjusting variable bindings and constraints. Its behaviour is supplied in the class `ProblemTabComposite`.

7.2.1.3 Preview Tab



```
simplify "a + b = b + a";
```

```
reading goal.... OK
calculating... OK
result:

true
```

FIGURE 7.5: The Preview Tab

The **Preview** tab (Figure 7.5), implemented in the class `PreviewTabComposite`, tracks the execution of the external tool. It displays a textbox for the input which is sent to the external tool, often a script and by default editable, and it displays a textbox for the output coming back from the tool (read-only, but supporting cut-and-paste), so that during interactive usage an advanced user can follow its activity (and *e.g.* note intermediate warnings which may be displayed). The tab includes a **Go** button to start a process, and a **Cancel** button, and the class provides logic for monitoring processes and enabling these buttons appropriately. This tab excludes the **Next** button, as typically **Finish** is the only logical next step, and includes auto-advance behaviour to switch to the final tab when the external process completes.

The class defines the method

```
    protected abstract boolean go(boolean forceInterrupt)
```

as the hook where tool-specific subclasses cause the external tool to act on the input problem. As this method is abstract (and since the core framework cannot know how to run a tool!) each plug-in must supply this method. We expect that implementations will commonly take the input from the corresponding textbox in this tab, which in turn will typically be updated using the field-change listener mechanism (provided by `ProversPaletteTabCompositeAbstract`, mentioned above), but in some situations, for instance where a tool is not script-based, the `go()` method may have other behaviour, or of course the **Preview** tab could be replaced altogether.

7.2.1.4 The Finish Tab

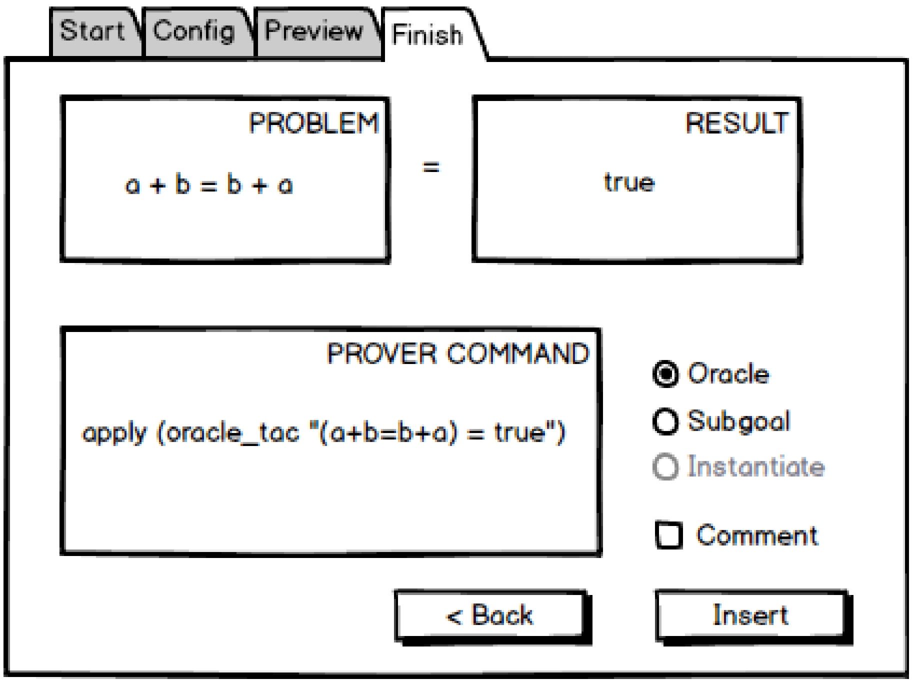


FIGURE 7.6: The Finish Tab

All the tabs mentioned so far include a **Finish** button, which results in the external tool being activated for a problem, and once a result is found the **Finish** tab is shown. This tab shows the input problem, in the tools language, and an equivalent output form, as in the mockup shown in Figure 7.6. The `FinishTabComposite` class contains stub methods where tool-specific subclasses can compute options for applying that result to present to the user: for example as a trusted oracle, as a subgoal, or as a witness (see Section 7.2.4.2 for more details). The abstract class supplied by the framework includes the input-and-output equivalence textboxes, buttons for these common application modes, and default prover command templates to effect the corresponding behaviours.

This class also defines a textbox where the prover command can be previewed and edited. If the user is happy with a command, they can select the **Insert** button which will be to cause the command to be inserted in the proof script and processed. Both this button and this default behaviour are supplied by this class — meaning a tool-specific subclass may not need to do very much at all for this tab!

The **Finish** tab also provides a **Comment** check box, which if selected will add a comment to the proof script detailing the exact command sent to the external tool. This makes it clear in the script how a result was achieved and allows the result to be reproduced subsequently without needing the Prover's Palette. Again, this functionality is provided by the abstract framework class for this tab.

7.2.1.5 The Toolbar

In addition to the tabs, the core Prover's Palette framework provides a toolbar to control common useful automatic behaviour of a tool plug-in in a consistent manner. This toolbar by default presents four actions:

- **Run Automatically**: if the current subgoal is amenable to the external tool it is translated and sent off automatically, running unobtrusively in the background
- **Show when Applicable**: if the current subgoal is amenable to the external tool then the tool's View pops up, showing the **Start** tab and alerting the user (the problem is only sent off if the user clicks **Finish**)
- **Show on Success**: if the external tool has been selected to run automatically in the background and it finds a result, then the tool's View pops up, showing the result in the **Finish** tab and allowing the user to decide if the result should be used in the proof
- **Run after Insert**: if the external tool finds a result it will automatically use this in the proof script (where the command inserted will be dependent on the specific system)

These actions are defined in the class *`ProversPaletteViewPartAbstract`*, mentioned previously in this section, in variables of the form *`action*`*. These actions exploit the framework's ability to run external tools concurrently with the prover. They are provided by the abstract class, so again, unless a plug-in needs to extend or replace this useful set of functionality, there is no need for tool-specific code. We believe these automation techniques, performing some computation in the background, will prove a useful pattern for making interactive theorem provers easier to use.

7.2.2 Controller

Recall that the controller layer must provide the mechanism by which tools in the Prover's Palette framework can:

(1) register for proof state change notifications
(2) apply changes to the proof script
(3) process the modified proof script in the prover

In Section 6.3 we reviewed several ways that TPs can interact with external systems. Based on this analysis and our survey of Proof Engineering, we identified the PG Kit (introduced in Section 6.4.2) as the most suitable choice of foundation for the controller layer. PG Kit provides a communications protocol and broker middleware for managing proofs-in-progress and mediating between components, immediately giving much of the needed functionality. Internally, PG Kit stores a model of the raw text of active proof scripts and parse-trees for these scripts (where the parsing is done by the prover, quasi-independently of applying proof steps, and used by PG Kit to determine command boundaries and theorem boundaries). It also keeps a model of the active proof state, again with a parse tree.

The fact that PG Kit is already embedded in a good IDE for Isabelle — Eclipse PG — and has access to the broader Eclipse ecosystem with its software engineering focus, made the choice of working with PG Kit in Eclipse a compelling conclusion. This of course makes Java the natural choice for implementation, and again this is a reasonable choice as Java is one of the most popular programming languages, with excellent support for concurrency particularly helpful for our purposes.

Given these capabilities already in the PG Kit, there was very little left for us to add to support our controller needs. To support (1) above, we created a lightweight eventing mechanism in *`ProversPaletteProofGeneralListener`*, building on the listeners in the PG Kit and some extensions in the FeaSch project [84]. This listener is registered with the PG Kit *`SessionManager`*, the nexus of its eventing system, as part of the initialisation done by the *`ProversPaletteViewPartAbstract`* mentioned above. To support (2) and (3), our framework provides a utility class *`ProofGeneralScriptingUtils`* which simplifies the processes of modifying the proof script and invoking prover processing, in a principled way. Thus the "Controller" box shown in Figure 7.2 is almost entirely out-of-the-box PG Kit, with a small amount of wrapping provided by us for convenience.

7.2.3 Translation

The `Listener` registered with the controller layer has access to the prover's state, represented in Proof General Markup Language (PGML). The Prover's Palette complements this by providing a rich set of routines for generating parse trees from PGML and for converting these parse trees between various systems. This section will cover many of the details of the representations and the translations; while the techniques are applicable to provers in general, we concentrate on Isabelle as that is the environment we primarily use.

To understand how the prover's state is represented in the PG Kit, consider the following lemma:

```
commute: "a + b = c ⟹ b + a = c"
```

Within the Prover's Palette, an external tool's View will have a registered listener which receives a callback containing the PGML for each new proof state. For the `commute` example, the listener receives the following object:

```
<pgip
   tag="Isabelle/Isar"
   id="/laura/442/1349731868.378"
   destid="PG-Eclipse"
   class="pg"
   refid="PG-Eclipse"
   refseq="15" seq="32">
  <normalresponse>
    <pgml area="display">
      proof (prove): step 0
      goal (1 subgoal):
      1. <atom kind="free">a</atom> +
         <atom kind="free">b</atom> =
         <atom kind="free">c</atom>
         <sym name="Longrightarrow">
           &lt;Longrightarrow&gt;</sym>
        <atom kind="free">b</atom> +
        <atom kind="free">a</atom> =
        <atom kind="free">c</atom>
      </pgml>
   </normalresponse>
</pgip>
```

This is passed to a `MathsProverTranslator` (*e.g.* for integrating externals tools with Isabelle, this is passed to the `IsabelleTranslator`) which creates a generic mathematical representation, based on that used in FeaSch [84] (which in turn is based on Isabelle notation). This generic representation defines a canonical set of mathematical symbols which are built up in a tree hierarchy, and this is used as a common intermediate language in our translation. This makes it quick to on-board a new system into the framework, as the translation need only be done to and from this common representation, with the ability to translate to and from all other existing systems following with no additional work. There are limitations with this approach, as with any approach to translation, namely that where concepts in any two systems do not align neatly either a specific system-to-system translation is required, or the common language must have an opinion on which one takes priority. Our routines follow the former strategy, and the canonical common language contains very little beyond near-universally recognised mathematics. For the arithmetic and logical operators we use notation identical to that of Isabelle's.

Translation from the prover to the common language begins with `MathsProverTranslator.preprocess` which tidies input — converting any extended characters to canonical representations[1], switching to meta-level quantification, and related activities. The current proof state then passes through `ProverTranslator.parse` which creates a parse tree based on grouping and operator precedence, yielding the common representation. For the example above, this results in the following parse tree:

```
ImplicationGroup("==>")
    OperatorGroup("=")
        OperatorGroup("+")
            Token("a")
            Token("b")
        Token("c")
    OperatorGroup("=")
        OperatorGroup("+")
            Token("b")
            Token("a")
        Token("c")
```

This parse tree is then inspected to determine whether the problem is compatible with the external tool, and the associated **Start** tab is updated as appropriate. Type information can be included, although in some environments this requires additional commands because typing is not currently a feature of PGML. (In the case of Isabelle, the user can set the flag `declare [[show_types]]` which makes the typing information explicit and available to PGML.)

When the user switches to the **Problem** tab in the View, the generic parse tree needs to be converted to the notation of the external tool.[2] This is done through the interface method `MathsSystemTranslator.fromCommon(genericParseTree)` (where `Maths`

[1] ASCII (*e.g.* ==>) replaces unicode (*e.g.* $\Longrightarrow$) as the latter continues to have portability problems and limited text file support.

[2] In fact, calls are made to generate the parse tree as early as the **Start** tab, to determine whether pre-processing is required.

`SystemTranslator` is actually a super-interface of `MathsProverTranslator`), which traverses the goal, operators, and variables, ensuring that:

- symbols are translated to their appropriate representation in the tool,
- variables are renamed where necessary,
- brackets of the appropriate type are inserted where necessary, and
- variable quantifications and bindings are extracted if appropriate

When we come to use a result from the external tool in the theorem prover (i.e. in the **Finish** tab) we need to be able to convert back from the tool's language to the prover's representation. This is done by the chain of calls in `MathsProverTranslator.fromCommon(MathsSystemTranslator.toCommon(toolExpression))`. Type information can be preserved, transformed, or inserted, depending on the capabilities of the two systems. For example, we may wish to transform the type information in the case where a prover refers to a type `nat` and an external system works with `Natural` numbers. We may wish to add type information to the results of an external tool if the tool only ever reasons over the `reals`; here we may wish to attach the type `real` to all variables passed back to the prover.

For more information on the translation, please consult the code and the unit tests.

7.2.4 Application

Let us now turn our attention to the reverse-path communication: how does activity from the tool View affect the proof? Specifically, Views need to be able to update the proof state by inserting commands into the proof script and instructing the prover to process the new commands. This is achieved by calls to `ProofGeneralScriptingUtils` as described in Section 7.2.2. The Prover's Palette includes common patterns of communication between the external tool and the prover. These patterns can be split into those which are done before the external tool is invoked, "manipulation", and those which are done by the prover in response to the output from the external tool, "using results". The following sub-sections describe these in more depth.

7.2.4.1 Manipulation

We noted earlier that `MathsProverTranslator` is a sub-interface of `MathsSystemTranslator`: this requires provers to implement a small number of additional methods for manipulating proof goals in common situations, either making them more amenable to use by a tool, or making the output from the tool more appropriate to the prover's context.

The first major area where proof states may need to be manipulated is where they contain predicate names which are foreign to the external tool. The translation routines in Prover's Palette can detect these unknown predicates; to help the user's productivity where a prover can automatically expand these predicates, the *MathsProver Translator* defines two methods. The first of these allows a prover integration to advertise this capability:

```
boolean shouldSuggestExpandUnknownPredicates(
          MathsExpression proverSubgoalText,
          Set<MathsToken> unknownPredicates);
```

When this method returns "true" for a proof state, a button labelled **Expand** is enabled in the **Start** tab. Clicking this button triggers a call to the second predicate-expansion method:

```
String getCommandForExpandingPredicates(
          Collection<String> unknownPredicates);
```

This method produces the commands which should be inserted into the proof script to expand the definitions of the unknown predicates. In the case of Isabelle, this method generates commands of the form:

```
"apply (simpl only: "+unknownToken+"_def)?"
```

In our experiments with CASs we realised that another barrier which can prevent proof goals being accepted by external tools is the presence of quantifiers within formulae: many tools require the goal to be in PNF, that is where all quantifiers are at the start of the goal. The Prover's Palette offers assistance in these situations by inspecting the parse tree to detect if a goal is not in prenex normal form. The following methods are provided by the framework:

```
boolean shouldSuggestConvertToPnf(
          MathsExpression proverSubgoalText);
String getCommandForConvertingToPnf(String text);
```

The first method is used to prompt the user that the goal is not in PNF, enabling a **PNF** button in the **Start** tab if the prover supports it. The second method produces the appropriate prover command for converting the goal to the desired form; in the case of Isabelle this is:

```
apply (atomize (full))?
apply (simp only: prenex_normal_form)
```

This ensures the representation is in the correct object logic (higher-order logic in our case, with the `atomize` step only applied if the goal is in Isabelle's meta logic format), and then uses a set of simp rules we have defined as `prenex_normal_form`[3]:

```
prenex_normal_form:
    "((∃ x.  P x) ∧ Q) = (∃ x.  P x ∧ Q)"
    "(P ∧ (∃ x.  Q x)) = (∃ x.  P ∧ Q x)"
    "((∃ x.  P x) ∨ Q) = (∃ x.  P x ∨ Q)"
    "(P ∨ (∃ x.  Q x)) = (∃ x.  P ∨ Q x)"
    "(( ∀ x.  P x) ⟶ Q) = (∃ x.  P x ⟶ Q)"
    "(P ⟶ (∃ x.  Q x)) = (∃ x.  P ⟶ Q x)"
    "(( ∀ x.  P x) ∧ Q) = (∀ x.  P x ∧ Q)"
    "(P ∧ (∀ x.  Q x)) = (∀ x.  P ∧ Q x)"
    "(( ∀ x.  P x) ∨ Q) = (∀ x.  P x ∨ Q)"
    "(P ∨ ( ∀ x.  Q x)) = (∀ x.  P ∨ Q x)"
    "(( ∃ x.  P x) ⟶ Q) = (∀ x.  P x ⟶ Q)"
    "(P ⟶ ( ∀ x.  Q x)) = (∀ x.  P ⟶ Q x)"
    "(¬ (∀ x.  P(x))) = (∃ x.¬P(x))"
    "(¬(∃ x.  P(x))) = (∀ x.¬P(x))"
by (iprover |  blast)+
```

Finally we may wish to insert type information into the result produced by an external tool, based on the domain the tool was reasoning about:

```
String annotateWithTypeInformation(String result,
        VariableBinding[] variablesAndBindings);
```

This prevents many otherwise common type errors from entering into the translation during use, as the prover will fail to be able to use a result if the types do not match, even if type information from the prover was not available or not passed to the external tool.

7.2.4.2 Using Results

There are several general ways a result from a tool might be used in a formal proof. Many of these are not specific to a single system, so it is useful to define their behaviour at the framework level.

An external tool might be used merely to explore a problem domain, in which case the result is not used in the prover. In other situations, a user may want to use a result explicitly in the proof script: for these situations, the Prover's Palette provides a number of convenience methods which present the user with the various types of commands. These commands are made appropriate to the context of the proof, respecting the quantification of variables the choice of object or meta level representation, and are expressed in the proof script syntax of the prover. The core Prover's Palette framework supplies GUI and proof-script-generation support for the following application modes:

[3]Creating this simplification set to convert problems to PNF was a pragmatic heuristic here, and it has worked well for the example problems we have encountered. However, Isabelle can apply these rules in any order and this technique has the potential to blow-up. A "stratified" approach, imposing an ordering on rule applications, would be a basis for a more reliable and efficient procedure.

Oracle: used when the result is taken on trust

Subgoal: used when the result is to be proved formally

Instantiate: used when a witness is discovered
(for an existentially quantified variable)

The methods and buttons associated with these modes are supplied in the `Finish TabComposite`. The behaviour associated with these modes can be re-used for any theorem prover, although the formulation of the specific commands is prover-dependent. For example, whenever a **Subgoal** button is enabled and activated, the following command is sent to the Isabelle proof script for processing:

```
getProverCommentForCurrentResultIfEnabled() +
        "apply (subgoal_tac \"" +
        getProverFormOfCurrentResult()+"\" )"
```

The first line checks to see if the user wishes a comment to be added to the proof script, containing the exact problem which was sent to the external tool (and thus allowing the result to be reproduced if desired).

Similarly, if the user selects the **Oracle** button, the following command is inserted into the Isabelle proof script:

```
getProverCommentForCurrentResultIfEnabled() +
        "apply (trustedtool \"" +
        getProverFormOfCurrentResult()+"\" )"
```

The `trustedtool` method has to be implemented in Isabelle. This is achieved by adding the following code to the proof script:

```
oracle trustedtool_oracle ("string") =
        {* fn thy ⇒ fn str ⇒
            HOLogic.mk_Trueprop (Sign.read_term thy str); *}

method_setup trustedtool =
        {* Method.simple_args Args.name
            (fn n ⇒ fn ctxt ⇒ Method.SIMPLE_METHOD
                (HEADGOAL
                    (Tactic.metacut_tac
                        (trustedtool_oracle
                            (ProofContext.theory_of ctxt) n))
                handle Fail _ ⇒ no_tac)) *}
        "trustedtool oracle"
```

In the case where the external tool has discovered a witness for an existentially quantified variable the `FinishTabComposite` can suggest that the user instantiate these variables back in the proof script. The Prover's Palette provides convenience methods for recursing through the parse tree in order to generate the relevant prover commands.

More illustrations of how these application modes are used in practice will be supplied in subsequent chapters of this thesis.

7.3 Conclusion

In this chapter we have introduced the Prover's Palette, a system and architecture for combining multiple mathematical software tools. By following a proof engineering methodology, where plug-ins for external tools can sit alongside the prover IDE, the Prover's Palette provides a tightly integrated and cohesive proving environment where integrations are centred around the user. Some of the main design principles highlighted in this chapter were that both novices and experts of a tool should be supported and that the core framework should be modular and easily extensible. We facilitate this last aim by providing Java superclasses for new tool integrations to extend: this is presented with regards to the four components of the framework (GUI, controller, translation and application) where common behaviour across tool plug-ins is likely.

As discussed, the Prover's Palette is implemented in Java and built upon the PG Kit which provides an Eclipse front-end to Isabelle. It should be noted, however, that development of PG Kit has since stopped and it is no longer officially supported. Despite this, it was already mature enough for our purposes when we began implementing the Prover's Palette, and in our view it remains one of the most advanced tools of its kind. It is encouraging to note that it is compatible with the current version of Isabelle (2012), a testament to the solid software engineering principles that went in to the PG Kit framework! Given the wealth of capabilities in Eclipse PG, we hold out hope — and not unreasonably we feel — that investment into PG Kit may yet resume and make it again the focal point of IDE development for Isabelle and other provers.

In the following chapters we will build upon the core Prover's Palette framework and illustrate the design principles of the system in action by presenting two concrete integrations: the first combining Isabelle with QEPCAD, and the second making some of Maple's functionality available in the Prover's Palette.

Chapter 8

QEPCAD in the Prover's Palette

Recall from Section 6.2.2 that QEPCAD [151] is a powerful implementation of the quantifier elimination decision procedure. In preliminary experiments we observed that QEPCAD was capable of solving many of the problems that arose in our proofs, particularly where Isabelle's automation is weak. Because it complements Isabelle's strengths, we chose QEPCAD as the first system to integrate into the Prover's Palette framework.

There are many ways in which QEPCAD could offer the user help when developing a formal proof. Some obvious modes of usage are:

- as a sanity check where the results simply guide the user; and
- as an oracle where the results are used in the proof, taken on trust.

As described in Section 6.3.2, in discussions with the developer of QEPCAD, we concluded that witness-generation would be extremely useful; as he has now implemented this, QEPCAD can now also be used for:

- finding a witness, which is then used to instantiate a bound variable;
- producing a counterexample for false conjectures; and
- simplifying algebra.

Additionally, we have found QEPCAD useful for:

- discovering the minimum set of assumptions required to make a conclusion hold (or, in other words, to remove superfluous assumptions); and
- experimenting with sets of the assumptions to determine whether there are any contradictory ones, thus making the conclusion true trivially.

In many domains, formal correctness requirements disallow reliance on tools such as QEPCAD. Nevertheless, it can be seen from the above lists that there are many ways QEPCAD's results can be of assistance to the user without sacrificing formal correctness, from simplifying the subgoal to finding witnesses and missing assumptions.

In this chapter we will show how the Prover's Palette framework enables each of these usage modes. We will use real-world problems to illustrate some of the ways the integration can be used, first looking at some which are fully automated and then showing how designing for interactivity allows a much wider set of problems to be solved. In both modes, we show how the integration can improve a user's understanding of their problem domain and how the automation provided by the integration reduces the context switching which can be so obstructive to one's intuition in theorem proving,

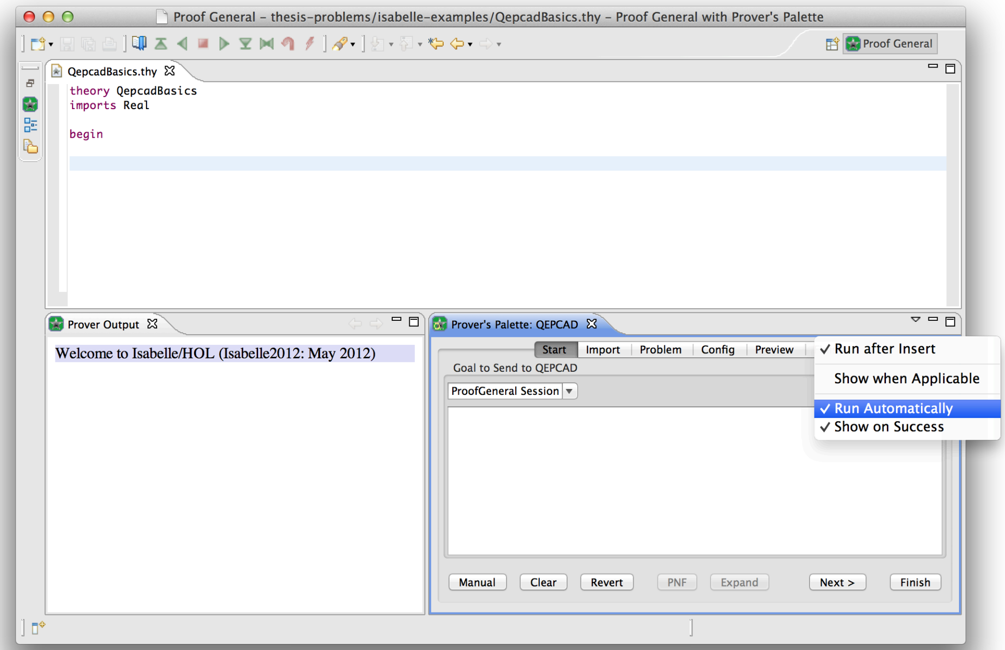
8.1 Automatic Insight

Let us begin with a look at the automation capabilities of the QEPCAD widget, where the tool can be configured to run in the background and appear only if it is able to produce a result, without any manual interaction.

8.1.1 Running QEPCAD

Figure 8.1 shows the Prover's Palette in Eclipse Proof General. The top half of the screenshot holds the Proof Script Editor, where the user writes the steps in the formal proof; as lines are sent to the prover, the current proof state is shown in the Prover Output View in the lower left; and, in the lower-right, is the Prover's Palette QEPCAD View.

Figure 8.1 also shows the dropdown toolbar menu of the QEPCAD View, where the automation behaviour of the widget can be configured. The user can tell the widget to **Run after Insert**, meaning that when a result from the tool is inserted in the script it should also be sent to the theorem prover for processing. The user can also tell the widget to **Show when Applicable**, where the widget will show itself when it detects a problem QEPCAD may be able to solve. For more automation, **Run Automatically** can be enabled to send any applicable problem to QEPCAD in the background, so that the result is there waiting if the user wants to check. A user can further enable **Show on Success**, where the QEPCAD widget will pop-up when a usable result is found by QEPCAD. With these two options enabled, the user can minimise the QEPCAD widget and work on her formal proof while the widget performs analysis unobtrusively in the background and appears whenever it has an answer.



```
theory QepcadBasics
imports Real

begin
```

```
Welcome to Isabelle/HOL (Isabelle2012: May 2012)
```

✓ Run after Insert
Show when Applicable
✓ Run Automatically
✓ Show on Success

FIGURE 8.1: The Prover's Palette with Isabelle and QEPCAD

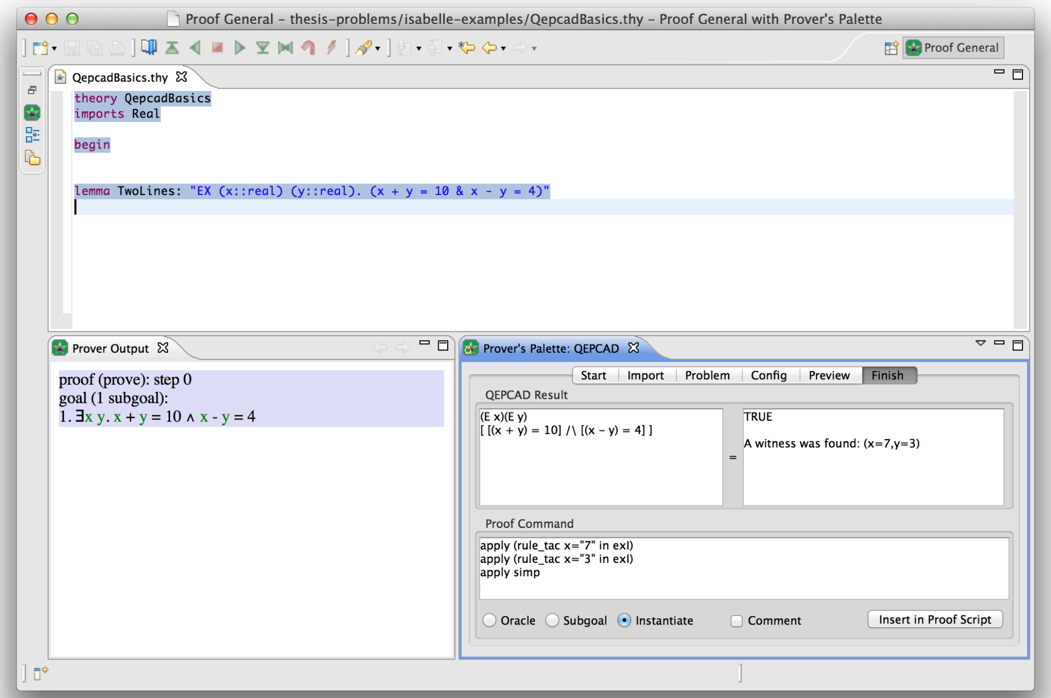
8.1.2 Using Results

To illustrate some of the ways the QEPCAD widget can assist with a formal Isabelle proof using the automation we just described, let us consider the problem of whether two lines intersect:

$$\exists\, x\, y.\ (x+y=10) \wedge (x-y=4)$$

Once this is entered into the proof script and sent to the prover (in the usual Eclipse PG manner), the resulting proof subgoal comes back to the PG broker, which dispatches it to the Prover Output View where it is displayed. With the Prover's Palette active, the broker also passes the new proof state to the QEPCAD widget. This assesses if the subgoal is amenable to QEPCAD, and, if it is, assuming the automatic modes of the widget are enabled, the problem is sent off to QEPCAD to run concurrently in the background. For this line-intersection problem, a result is found immediately and the QEPCAD widget pops-up on the screen, as shown in Figure 8.2.

The **Finish** tab is displayed, as described as part of the core Prover's Palette framework (Chapter 7), showing the key information from the QEPCAD output. The result is shown in QEPCAD's notation, for clarity about the result, with options presented to



```
theory QepcadBasics
imports Real

begin

lemma TwoLines: "EX (x::real) (y::real). (x + y = 10 & x - y = 4)"
```

```
proof (prove): step 0
goal (1 subgoal):
1. ∃x y. x + y = 10 ∧ x - y = 4
```

```
(E x)(E y)
[ [(x + y) = 10] /\ [(x - y) = 4] ]
```

=

```
TRUE

A witness was found: (x=7,y=3)
```

```
apply (rule_tac x="7" in exI)
apply (rule_tac x="3" in exI)
apply simp
```

Oracle Subgoal [✓ Instantiate] Comment

FIGURE 8.2: Using Results Found by QEPCAD

the user for how it can be applied to the proof. Here, all three of the application modes from Section 7.2.4.2 are applicable — oracle, subgoal and instantiate.

The QEPCAD widget automatically selects the mode its heuristics judge best: in this case, the **Instantiate** button is chosen as the default, as a witness has been found which can make the statement true without introducing a dependency on either QEPCAD or the Prover's Palette. The Isabelle commands associated with this option are shown, and the user can simply click **Insert in Proof Script** to automatically insert these commands into the proof script (and the commands will also automatically be processed if the **Run after Insert** toolbar option is enabled). For this example, this is sufficient to discharge the proof goal, fully formally, with just one click from the user.

There will of course be situations where witnesses are not or cannot be produced. In these instances, one may be willing to use the results of an external tool without formal proof (*i.e.* the external tool acts as a trusted oracle). This might be done for pragmatic reasons, with a clear indication of what external results are being used so that a reader can verify them to their own satisfaction (by hand or using their preferred tools), and/or with the anticipation that fully automated, formally correct methods will in time become available for proving these statements.

Although it is not the best option for the line-intersection problem, **Oracle** mode could have been selected. If that were the case, the widget would have generated the appropriate Isabelle commands for the translated QEPCAD result to be trusted by that prover:

```
apply (qepcad "(∃(x::real).  ∃(y::real).
        (((x + y) = 10) ∧ ((x - y) = 4))) = True")
apply blast
```

With these commands inserted into the proof script, the lemma is then "proved". Note that the `qepcad` oracle tactic is defined in Isabelle similarly to the `trustedTool` tactic presented in Section 7.2.4.2.

One could instead select the **Subgoal** application option, which for this problem generates the Isabelle command:

```
apply (subgoal_tac "(∃(x::real).  ∃(y::real).
        (((x + y) = 10) ∧ ((x - y) = 4))) = True")
```

This has no effect on our proof state here, but there are many situations where it can be useful: when QEPCAD returns a simplified formula rather than `True` or when the problem has been manually altered, inserting the result as a subgoal allows simplification in a fully formal way. As this is primarily applicable during interactive usage, this will be covered in more detail in the next section.

These three ways of using results are defined by the Prover's Palette core framework; but due to the variety of notations used in external systems, there are some tool-specific aspects in how results apply in different situations and with different provers. Translation will be discussed in greater detail in the next subsection, but let us briefly note some of the specifics for the commands which are generated. First, we imagine that the user has attempted to prove the line-intersection subgoal themselves in Isabelle, by first eliminating the existential quantifiers — using the command `apply (rule exI)+`. The current subgoal in Isabelle then becomes:

```
?x + ?y2 = 10 ∧ ?x - ?y2 = 4
```

The QEPCAD widget automatically detects that the variables $?x$ and $?y2$ are implicitly bound by existential quantifiers, it discovers the same witness as before, and it again makes the suggestion that the user **Instantiate** the result. However, because different commands are required for implicit and explicitly quantified variables, the integration proposes a different command to be inserted into the proof script:

```
apply (rule_tac P = "7+3=10 ∧ 7-3=4" in TrueE, rule TrueI)
apply simp
```

In addition to these three ways of applying a result to progress the proof (oracle, instantiate and witness), the user is sometimes presented with a fourth option which merely simplifies the proof state. QEPCAD may discover that certain assumptions are not needed in a proof, and it will provide an option and prover commands to "thin" (remove) those assumptions from the current goal. We will say more about this in Section 8.3.2.

Finally, recognising the importance of recording the provenance of a result, the widget offers to include an explanatory comment in the proof script. This contains all the requisite details to re-run the computation in QEPCAD without any reliance of the prover or the Prover's Palette.

8.1.3 Translation

Let us now look at the different aspects of translation, from mapping mathematical symbols between Isabelle and QEPCAD to inferring the bindings on variables and offering automated pre-processing to massage a subgoal so it may be understood by QEPCAD.

Mathematical Symbols and Logical Operators

Recall from Section 7.2.3 that the Prover's Palette Java infrastructure provides a common representation of the current subgoal, *i.e.* the parse tree generated from the PGML representation of the proof state. For a concrete tool integration, we must then describe how to translate from this common representation to the language of the external tool, in this case QEPCAD. The `QepcadTranslator` class describes this translation to and from the common notation for the mathematical operators and logical symbols (which in the Prover's Palette at present is based on Isabelle). The code uses a bi-directional map initialised as follows:

```
ISABELLE_QEPCAD_TRANSLATIONS.put("+", "+");
ISABELLE_QEPCAD_TRANSLATIONS.put("-", "-");
ISABELLE_QEPCAD_TRANSLATIONS.put("*", " ");
ISABELLE_QEPCAD_TRANSLATIONS.put("^", "^");

ISABELLE_QEPCAD_TRANSLATIONS.put("=", "=");
ISABELLE_QEPCAD_TRANSLATIONS.put("~=", "/=");
ISABELLE_QEPCAD_TRANSLATIONS.put("=/", "/=");

ISABELLE_QEPCAD_TRANSLATIONS.put(">", ">");
ISABELLE_QEPCAD_TRANSLATIONS.put("<", "<");
ISABELLE_QEPCAD_TRANSLATIONS.put("<=", "<=");
ISABELLE_QEPCAD_TRANSLATIONS.put(">=", ">=");
ISABELLE_QEPCAD_TRANSLATIONS.put("~", "~");
ISABELLE_QEPCAD_TRANSLATIONS.put("&", "/\\");
ISABELLE_QEPCAD_TRANSLATIONS.put("|", "\\/");
ISABELLE_QEPCAD_TRANSLATIONS.put("/\\", "/\\");
ISABELLE_QEPCAD_TRANSLATIONS.put("\\/", "\\/");

ISABELLE_QEPCAD_TRANSLATIONS.put("-->", "==>");
ISABELLE_QEPCAD_TRANSLATIONS.put("<--", "<==");
ISABELLE_QEPCAD_TRANSLATIONS.put("<->", "<==>");

ISABELLE_QEPCAD_TRANSLATIONS.put("==>", "==>");
ISABELLE_QEPCAD_TRANSLATIONS.put("<==", "<==");

ISABELLE_QEPCAD_TRANSLATIONS.put("True", "TRUE");
ISABELLE_QEPCAD_TRANSLATIONS.put("False", "FALSE");
```

where the first argument is in the common (Isabelle-based) notation and the second is in QEPCAD's notation. As can be seen, many of their symbols are identical, but including them in the map is an indication that the operator is supported.

The `QepcadTranslator` class also implements the translation of the correct bracketing; where Isabelle only uses parentheses around terms, QEPCAD has a slightly more complex language, requiring square brackets around logical expressions and parentheses around numeric expressions where needed to override operator precedence.

Variables and Quantifiers

If the variable bindings are explicit in a subgoal, the Prover's Palette simply needs to translate between the Isabelle notation and QEPCAD's: so `ALL` in Isabelle gets translated to `A` in QEPCAD and `EX` in Isabelle gets translated to `E` in QEPCAD. The translation is more cumbersome when the bindings of variables are implicit in the subgoal. In this situation the Prover's Palette translation has to decipher which binding to infer:

- if the subgoal states $\bigwedge$`x.` then infer `x` is universally bound;
- if the subgoal states `?x` then infer `x` is existentially bound; and
- if there is no information in the subgoal about a variable's binding then assume it is universally bound.

In addition, the Prover's Palette massages any variable names which aren't permitted by QEPCAD. For example `?x` is translated to just `x`. If there is already a variable named `x`, the `QepcadTranslator` detects the clash and reverts the binding of `x` to "unknown", with the consequence that all components which contain `x` will default to being deselected within the QEPCAD widget.

Types

The Prover's Palette stresses flexibility, ease-of-use, and safety. To this end, where types other than "real" are present in a problem statement, the QEPCAD widget allows the goal to be sent to QEPCAD but it includes the type information in resulting commands inserted into a proof. Thus the widget does not prevent potentially useful explorations, but it does prevent incorrect applications of a result in the proof script. This is the same technique used by `blast` within Isabelle.

Similarly, if there is a clash where differently-scoped variables have the same name (*i.e.* where the binding type is detected as "unknown" but a user interactively selects one or more of those components), the resulting proof commands have the variables scoped globally reflecting the computation as validated by QEPCAD. This prevents name-clash confusion from applying a result incorrectly, because the prover will refuse to unify the globally-scoped variable in the generated commands with multiple identically named variables in the proof state.

8.1.4 Generating QEPCAD Commands

A second, related phase of processing is that which generates the command script which will be sent to QEPCAD. This is performed by the `QepcadProblem` class, taking into account the components, variables, and bindings selected in the **Problem** tab. In general, this is a straightforward process, as QEPCAD's syntax is relatively simple: variables are listed first with their bindings, followed by the problem in QEPCAD notation, followed by configuration options. There are, however, some subtleties.

Selecting Applicable Parts

If all the components of a subgoal are compatible with QEPCAD, the situation is simple as the entire problem can be sent to QEPCAD. However, where one or more components are not compatible, the QEPCAD widget deselects those components by default, and the GUI allows the user to send the remainder of the subgoal to QEPCAD. For example, if a subgoal was of the form:

$$a+b=10 \wedge a < b \wedge 1 < a \wedge coprime\ a\ b \Longrightarrow a = 3$$

the widget would deselect the `coprime` component and just send off the remainder of the goal to QEPCAD. This is in keeping with the Prover's Palette philosophy that the an external tool integration should be helpful but not overly restrictive. As noted above, the result of this computation will be a valid statement, according to QEPCAD, and the application back in the proof — if permitted — will be as trustworthy as the usage mode (oracle, subgoal, instantiate) and the tool permits. The result is not guaranteed to be useful — as in the example where we removed the *coprime* fact, QEPCAD would say false — but the result is guaranteed to be safe:

$$(a+b=10 \wedge a < b \wedge 1 < a \longrightarrow a = 3) = False$$

Variable Bindings

Where a variable occurs only in deselected terms in the **Problem** tab, it is removed from the list of variables which are sent to QEPCAD. (QEPCAD will fail if a variable in the variable list does not occur in the problem or if a variable in the problem is not in the variable list.) However, if a variable occurs in both a selected term and a deselected term, the widget will initialise it as a free variable: the reason for this is that the user's intention here will usually to use QEPCAD to reduce the problem in terms of such variables, *i.e.* to eliminate all variables where all known information is included and return a result containing only those variables for which there are additional constraints not given to QEPCAD.

As before, the user can manually change these bindings to explore the problem. The result may or may not be interesting to them, and it will usually not be applicable to the proof state, but it will always be safe to attempt to apply it.

Pre-processing

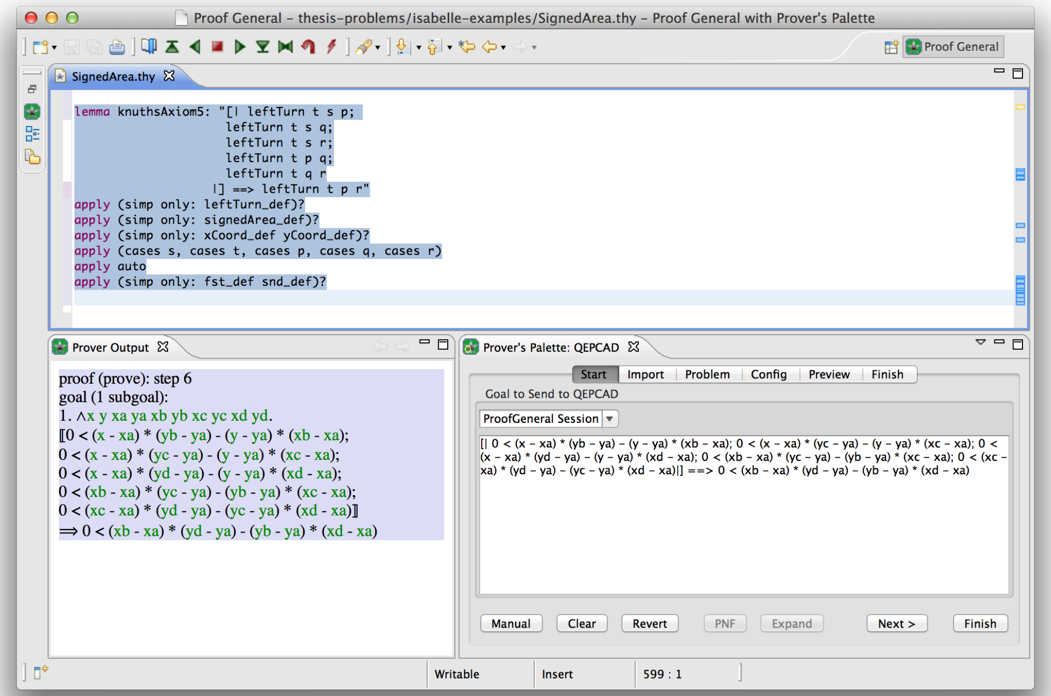
In many cases where a subgoal is not wholly compatible with QEPCAD, it can be transformed using the prover to become compatible. The two most common such transformations are converting the subgoal to prenex normal form (PNF) and expanding predicates which QEPCAD does not understand (*e.g.* defined in an Isabelle theory). Whilst massaging a goal in this way is not difficult, it can be extremely tedious, and this tedium limits the utility of the Prover's Palette. This burden is lifted from the user through simple automation which does the necessary pre-processing: two buttons — **PNF** and **Expand** — are provided, enabled when appropriate, to generate, insert and apply the commands to perform these transformations.[1]

[1]The proof commands used to perform the PNF conversion and the expansion of predicate definitions are successful in most, but not all, cases. They will, of course, never cause an unsafe transformation, but if they cannot completely convert the problem the user may, on occasion, have to manually massage the proof state to be usable in QEPCAD. This can be done either in the prover or in the QEPCAD widget.

To illustrate this, let us return to one of the lemmas introduced in Chapter 3, from our Isabelle theory of Signed Area:

```
transitivity:  leftTurn t s p  ∧ leftTurn t s q
                ∧ leftTurn t s r ∧ leftTurn t p q
                ∧ leftTurn t q r ⟹ leftTurn t p r
```

Figure 8.3 shows this problem in the Isabelle proof script. All of the commands processed after the lemma definition were inserted by the QEPCAD widget in order to massage the problem into a form compatible with QEPCAD. Here, the Prover's Palette suggests to expand `leftTurn`, then `signedArea` and `xCoord` and `yCoord`; then it detects we have to expand via a different mechanism for the cases necessary to simplify `Rep_point` (the internal representation of a tuple being a point). When all these expansions are completed we have a problem which is suitable for processing by QEPCAD.



```
lemma knuthsAxiom5: "[| leftTurn t s p;
                        leftTurn t s q;
                        leftTurn t s r;
                        leftTurn t p q;
                        leftTurn t q r
                      |] ==> leftTurn t p r"
apply (simp only: leftTurn_def)?
apply (simp only: signedArea_def)?
apply (simp only: xCoord_def yCoord_def)?
apply (cases s, cases t, cases p, cases q, cases r)
apply auto
apply (simp only: fst_def snd_def)?
```

```
proof (prove): step 6
goal (1 subgoal):
1. ⋀x y xa ya xb yb xc yc xd yd.
⟦0 < (x - xa) * (yb - ya) - (y - ya) * (xb - xa);
0 < (x - xa) * (yc - ya) - (y - ya) * (xc - xa);
0 < (x - xa) * (yd - ya) - (y - ya) * (xd - xa);
0 < (xb - xa) * (yc - ya) - (yb - ya) * (xc - xa);
0 < (xc - xa) * (yd - ya) - (yc - ya) * (xd - xa)⟧
⟹ 0 < (xb - xa) * (yd - ya) - (yb - ya) * (xd - xa)
```

```
[| 0 < (x – xa) * (yb – ya) – (y – ya) * (xb – xa); 0 < (x – xa) * (yc – ya) – (y – ya) * (xc – xa); 0 <
(x – xa) * (yd – ya) – (y – ya) * (xd – xa); 0 < (xb – xa) * (yc – ya) – (yb – ya) * (xc – xa); 0 < (xc –
xa) * (yd – ya) – (yc – ya) * (xd – xa)|] ==> 0 < (xb – xa) * (yd – ya) – (yb – ya) * (xd – xa)
```

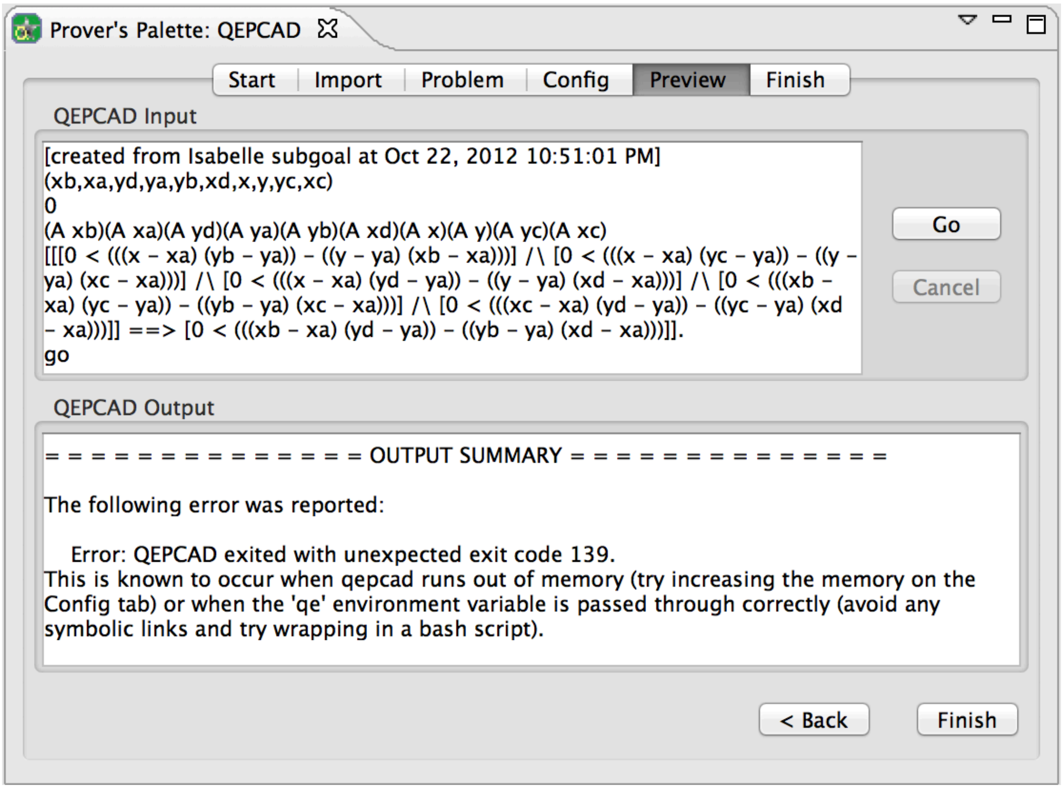
FIGURE 8.3: The Start Tab

8.2 Interactive Tabs

In addition, to be able to run QEPCAD fully automatically, one of the primary objectives of the Prover's Palette is to support interactive configuration of a tool and control over how it is used. Let us now turn our attention to the interactive tabs of QEPCAD widget which support this.

8.2.1 Configuration

For the `transitivity` problem presented in Section 8.1.4, the QEPCAD widget does not take the user to the **Finish** tab, but to the **Preview** tab, shown in Figure 8.4. This tab shows the user the output from QEPCAD and alerts them to the fact that QEPCAD encountered a problem, most likely running out of memory. This problem happens because the SACLIB library, which QEPCAD uses, initialises itself with a fixed-size heap of memory (2,000,000 cells in garbage-collection space) and fails if that is used up.

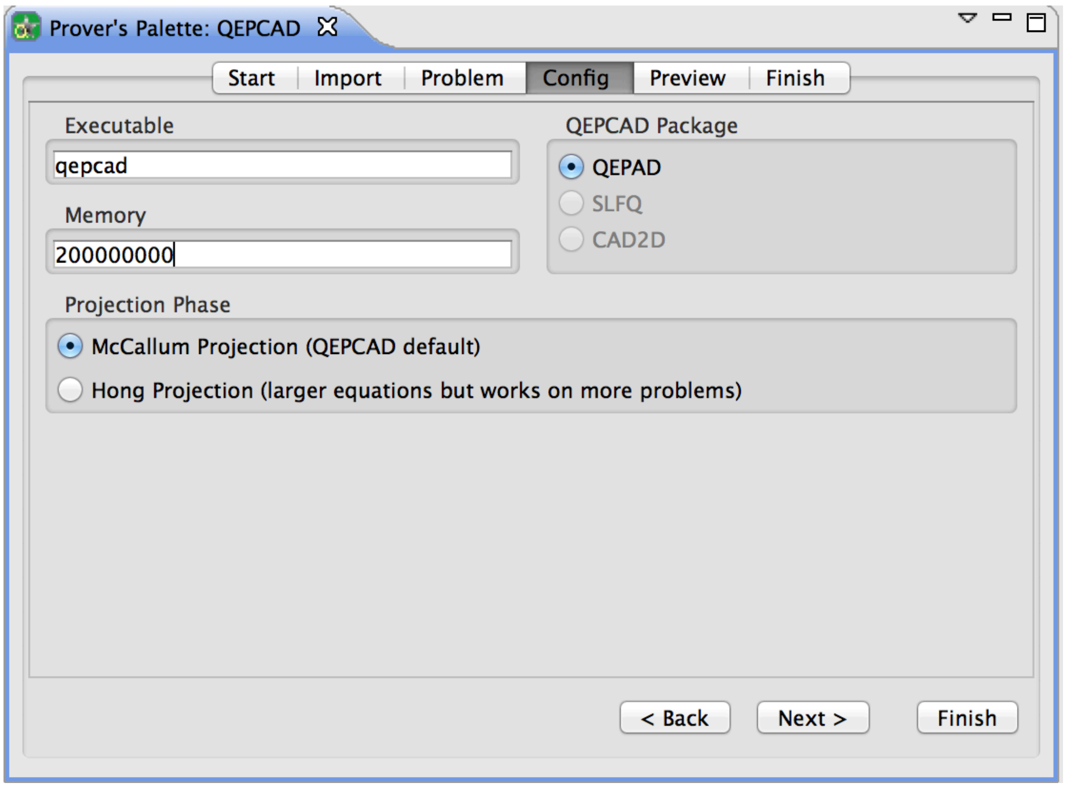


```
[created from Isabelle subgoal at Oct 22, 2012 10:51:01 PM]
(xb,xa,yd,ya,yb,xd,x,y,yc,xc)
0
(A xb)(A xa)(A yd)(A ya)(A yb)(A xd)(A x)(A y)(A yc)(A xc)
[[[0 < (((x – xa) (yb – ya)) – ((y – ya) (xb – xa)))] /\ [0 < (((x – xa) (yc – ya)) – ((y –
ya) (xc – xa)))] /\ [0 < (((x – xa) (yd – ya)) – ((y – ya) (xd – xa)))] /\ [0 < (((xb –
xa) (yc – ya)) – ((yb – ya) (xc – xa)))] /\ [0 < (((xc – xa) (yd – ya)) – ((yc – ya) (xd
– xa)))]] ==> [0 < (((xb – xa) (yd – ya)) – ((yb – ya) (xd – xa)))]].
go
```

```
= = = = = = = = = = = = = = OUTPUT SUMMARY = = = = = = = = = = = = = =

The following error was reported:

   Error: QEPCAD exited with unexpected exit code 139.
This is known to occur when qepcad runs out of memory (try increasing the memory on the
Config tab) or when the 'qe' environment variable is passed through correctly (avoid any
symbolic links and try wrapping in a bash script).
```

FIGURE 8.4: The Preview Tab

This issue can be rectified by allocating a larger amount of memory when QEPCAD is started. Our QEPCAD widget makes this possible in the **Config** tab, shown in Figure 8.5. For our transitivity problem, after two (manual) iterations of increasing this by a factor of 10, we find that with 200,000,000 cells, QEPCAD is able to solve it. However, with this increased memory QEPCAD takes 1m 1.3s on a dual-core 2.0 GHz machine. In the event that the user feels a computation is taking too long, they can always click the **Cancel** button in the **Preview** tab to interrupt QEPCAD. For this problem, he could then choose to manually apply the translation invariance technique from Section 6.3.3 to make one point the origin; the translated problem can then be sent to QEPCAD, and this, on the same machine, takes a mere 0.86s.



Executable: qepcad

Memory: 200000000

QEPCAD Package: [✓ QEPAD] SLFQ CAD2D

Projection Phase: [✓ McCallum Projection (QEPCAD default)] Hong Projection (larger equations but works on more problems)

FIGURE 8.5: The Config Tab

Other configuration settings offered by QEPCAD, such as projection type and alternate packages, are also adjustable in the **Config** tab. For a description of these modes, the reader is referred to the QEPCAD documentation [151].

8.2.2 Modifying the Problem

When a formula is sent to QEPCAD, one of three results can come back: True, False, or a simplified formula. For the third case, the simplified formula is expressed in terms of the free variables; this can be useful for reducing the number of variables in an Isabelle problem, thereby easing the formal reasoning process. This feature can be accessed in the **Import** tab, by overriding the default "solve" mode where all variables are quantified and altering the variable bindings table. It can also be triggered automatically in some cases by deselecting individual proof components, as noted above.

To illustrate this, using QEPCAD to simplify expressions, we will look at a collision problem described by Collins and Hong [40]. The problem is taken from robot motion planning and queries whether two moving objects will ever collide. Consider a moving circle (1) and a moving square (2), represented algebraically as:

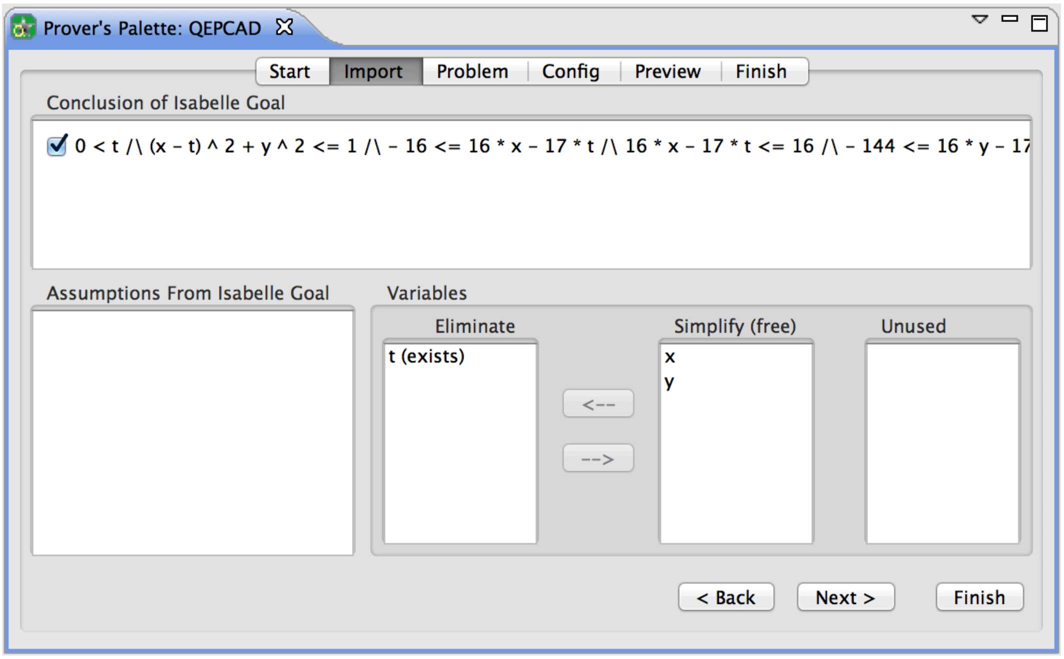
$$\begin{array}{ll} (1) & (x-t)^2 + y^2 \leq 1 \\ (2) & -1 \leq x - \frac{17}{16}t \leq 1 \wedge -9 \leq y - \frac{17}{16}t \leq -7 \end{array}$$

We wish to know whether these objects will ever collide, *i.e.* whether:

$$\exists\, t\; x\; y.\; t \geq 0 \wedge (1) \wedge (2)$$

In the default "solve" mode QEPCAD quickly confirms that this is `True`: there will be a time when the circle and square collide. In Isabelle alone, this is a difficult proof. If the user trusts the Isabelle/QEPCAD integration, of course, they can simply move on, but even when the user requires a fully formal proof the integration can be of assistance. QEPCAD can tell us that one possible solution is: $t{=}\frac{96}{17}$, $x{=}\frac{96}{17}$ and $y = -1$. Instantiating these variables in Isabelle completes the proof.

However we may prefer to simplify the algebra rather than use a witness or take the result on trust. As Harrison *et al.* have pointed out, it is sometimes easier to prove that the original problem is equivalent to a reduced formula and then prove this reduced formula than to formally prove the original [83].



Conclusion of Isabelle Goal

✓ 0 < t /\ (x - t) ^ 2 + y ^ 2 <= 1 /\ - 16 <= 16 * x - 17 * t /\ 16 * x - 17 * t <= 16 /\ - 144 <= 16 * y - 17

Assumptions From Isabelle Goal

Variables

Eliminate	Simplify (free)	Unused
t (exists)	x	
	y	

FIGURE 8.6: The Import Tab

In the collision problem, the parametric query can be transformed into the implicit representation of the problem by keeping the variables x and y free and only binding t. This can be done in the **Import** tab, as shown in Figure 8.6.

QEPCAD returns the answer:

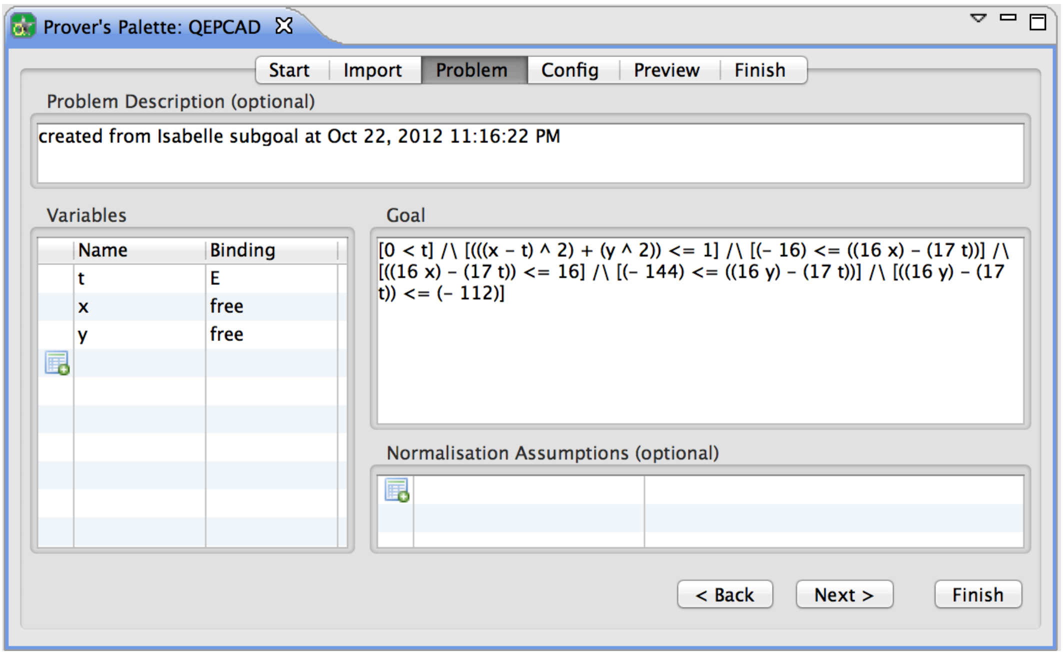
$$
\begin{aligned}
& y+1 \geq 0 \wedge y-1 \leq 0 \wedge x-y-6 \geq 0 \wedge \\
& \big[\ 289x^2-544yx-4896x+545y^2+4608y+20447 \leq 0 \vee \\
& \quad 289x^2-544yx-3808x+545y^2+3584y+12255 \leq 0 \vee \\
& \quad [17x-16y-112 > 0 \wedge 17x-16y-144 < 0]\ \big]
\end{aligned}
$$

Here, the **Subgoal** option is offered in addition to the **Oracle** option. If the Subgoal mode is selected, it introduces a new subgoal in the proof which asserts that the original Isabelle subgoal is equivalent to the reduced form found by QEPCAD (with t eliminated, in this example). In place of the original single subgoal, we now have two simpler subgoals: the equivalence of the two formulas, and the existence of an x and y satisfying the new form. Proving the equivalence may still be challenging, but in some situations it is simpler than proving the original goal.

Here, of course, the witness is much easier to work with than the two subgoals, but if we were exploring this problem or seeking to use the result in a different context, the non-parametric form of the equation could be very useful. If any of the variables are not existentially quantified, witnesses are not available, and a reduced intermediate form such shown here may be the simplest way to achieve a fully formal proof.

Instead of using the Import tab to change the bounds, we could have instead gone to the **Problem** tab (Figure 8.7). This tab describes the problem as it will be sent to QEPCAD and allows the user to edit any aspect of it. For many types of exploration, the interface on this tab is easier to work with as it represents the objects as sent to QEPCAD rather than as imported from the prover. The **Problem** tab can also be used as a graphical front-end to QEPCAD even in stand-alone mode, without any associated theorem prover.

This concludes the exploration of the tabs — but only scratches the surface of the ways the tool can be used. We will look at the other ways we have found the tool useful in the coming section.



Problem Description (optional)

created from Isabelle subgoal at Oct 22, 2012 11:16:22 PM

Variables

Name	Binding
t	E
x	free
y	free

Goal

[0 < t] /\ [(((x − t) ^ 2) + (y ^ 2)) <= 1] /\ [(− 16) <= ((16 x) − (17 t))] /\ [((16 x) − (17 t)) <= 16] /\ [(− 144) <= ((16 y) − (17 t))] /\ [((16 y) − (17 t)) <= (− 112)]

Normalisation Assumptions (optional)

FIGURE 8.7: The Problem Tab

8.3 Exploration and Discovery

As we identified in Chapter 5, one of the biggest problems encountered when developing a formal proof is the loss of intuition; ready access to powerful tools such as QEPCAD can unblock this. In this section we will explore some of the ways we have found this to be the case.

8.3.1 Counterexamples and Missing Assumptions

For complex proof developments, it is not uncommon to discover that the first attempt at formally specifying a problem is incorrect. This is especially true when verifying algorithms using Hoare logic. In this setting, as shown in Chapter 4, the user often has to provide a loop invariant (*i.e.* facts which do not change on each iteration of the loop) which is sufficient to ultimately deduce the correctness of the algorithm. Discovering a correct loop invariant is a challenging task, often requiring several rounds of refinement guided by failed proof attempts. The root cause for failure is frequently a missing assumption, but the missing assumption can be hard to identify.

With Isabelle integrated with QEPCAD in the Prover's Palette, discovering the missing assumptions of invalid theorems is simplified. We show two techniques here: a counter-example generator which identifies instances which may have been overlooked, and an interactive exploration strategy which alters variable bindings to characterise overlooked cases. This applies not just to loop invariants, but in numerous situations where the complexity of a proof state may obscure the fact that it is incorrect or incomplete. These techniques can also address some of the difficulties associated with refactoring, painful in many situations but particularly so when refining loop invariants (Section 5.1.1): the techniques help not by improving refactoring support in the IDE, but by reducing the amount of refactoring, as we can now get an early warning of a wrong definition or invariant.

As described in Section 6.3.2, Isabelle includes a feature which can automatically generate counter-examples for some problems. QEPCAD complements this capability by using different methods to identify false subgoals. The counter-example feature is applicable when a false conjecture contains only universally quantified variables. In these situations, the user can click a button which automatically translates the negated original conjecture into the equivalent existential form:

$$(\neg\, \forall\, x_1\ ...\ x_n.\ \Psi(x_1,\ldots,x_n)) = (\exists\, x_1\ ...\ x_n.\ \neg\Psi(x_1,\ldots,x_n))$$

where $\Psi(x_1,\ ...\ ,x_n)$ is a quantifier free formula. It then calls the witness function of QEPCAD to obtain a counterexample to the original conjecture.

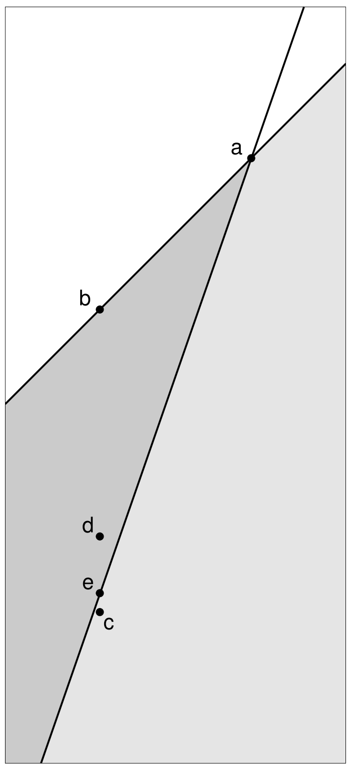
Let us demonstrate this feature through an illustrative example taken from our verification of Graham's Scan:

```
leftTurn b e a ∧ leftTurn a b d ∧
leftTurn c a b ∧ leftTurn a d e ⟹
    leftTurn a c e
```

By translating the point `a` to be the origin and then using the QEPCAD widget, we can quickly see that this lemma is false. In contrast, our original encounter with this problem — before the Prover's Palette — was but one of many instances where hours were lost because we had overlooked something small.

In this particular case, we know from the context that a subgoal like this is required, but the exact form is not clear. The counter-example generator in the Prover's Palette tells us that the following instantiations will falsify the original lemma:

$$a = (0,0) \quad b = (-1,-1) \quad c = (-1,-3) \quad d = (-1,-\frac{5}{2}) \quad e = (-1,-\frac{23}{8})$$



By drawing this particular case (manually) we gain an insight into why the conjecture is false. It is clear that all the assumptions hold, but the conclusion does not. Here, in the original context, it can be seen that *c should* be constrained to lie inside the dark grey region. One way which this constraint could be introduced would be to add an additional assumption, $\neg \circlearrowleft adc$. In the context where this problem arose, this is a valid assumption to introduce and in fact one which leads ultimately to the correct loop invariant.

Despite the counter-example generator's capacity here to give us an understanding why the conjecture is false, it is a method that is not always applicable. Existential quantifiers may be present in the conjecture or the number of variables and assumptions may be so large that drawing the situation to gain insight is not practical.

Another way the Prover's Palette can help to discover missing assumptions of a false conjecture is by allowing the user to experiment with changing the bindings of variables: setting certain variables free and leaving others bound can identify what is missing. As shown previously, this can be achieved easily in the **Import** tab. There is some art in selecting which variables should be free, but in the example just given, we reason that as variables a and b occur the most, we translate a to be the origin and prefer b kept bound. With the other variables set "free", QEPCAD returns:

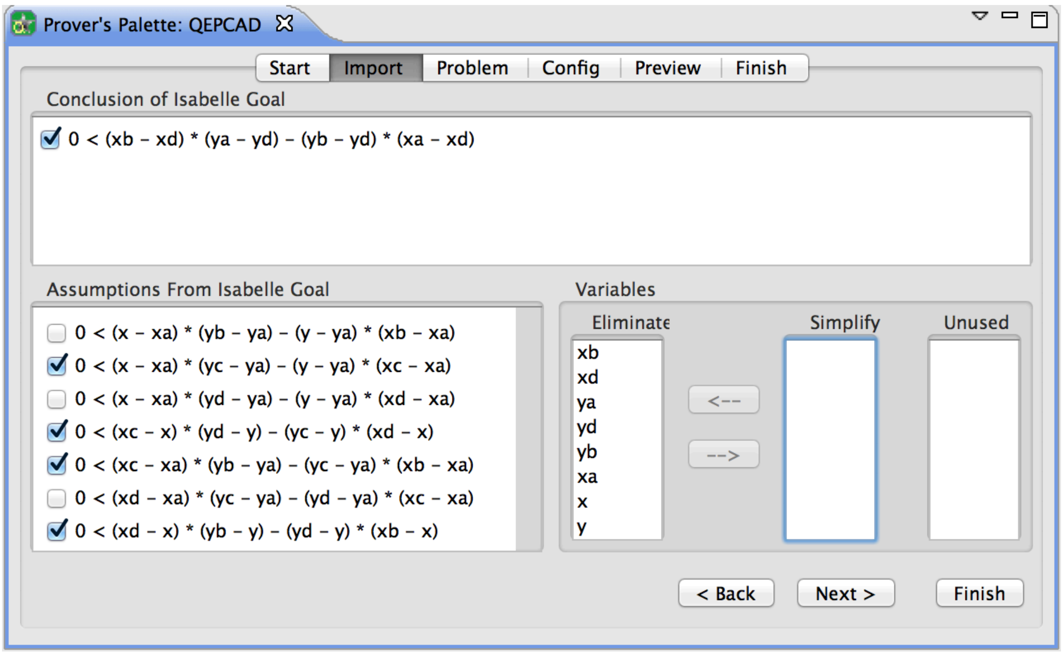
$$d_y c_x - d_x c_y \geq 0 \vee d_x e_y - d_y e_x \leq 0 \vee e_y c_x - e_x c_y > 0$$

The second and third disjuncts are unenlightening (a negated assumption and the conclusion), but the first disjunct is the hitherto missing condition to our Isabelle lemma. QEPCAD has helped guide our intuition and, recalling that a was set to be the origin, the lemma can be proven if we assume $\neg \circlearrowleft adc$. This is the same missing assumption we discovered using the counter-example generator: the Prover's Palette here would have led us to the discovery of a missing component in the loop invariant.

8.3.2 Unnecessary and Inconsistent Assumptions

We can also use the Prover's Palette to discover unnecessary and inconsistent assumptions. Consider the following problem also taken from our verification of Graham's Scan:

$$\begin{array}{l} \mathit{leftTurn}\ s\ t\ q \wedge \mathit{leftTurn}\ s\ t\ r \wedge \\ \mathit{leftTurn}\ s\ t\ p \wedge \mathit{leftTurn}\ t\ r\ p \wedge \\ \mathit{leftTurn}\ s\ r\ q \wedge \mathit{leftTurn}\ s\ p\ r \wedge \\ \mathit{leftTurn}\ t\ p\ q \wedge \neg \mathit{leftTurn}\ p\ s\ q \wedge \\ \mathit{leftTurn}\ t\ q\ r \Longrightarrow \\ \quad \mathit{leftTurn}\ p\ q\ s \end{array}$$



[✓] 0 < (xb – xd) * (ya – yd) – (yb – yd) * (xa – xd)

[ ] 0 < (x – xa) * (yb – ya) – (y – ya) * (xb – xa)
[✓] 0 < (x – xa) * (yc – ya) – (y – ya) * (xc – xa)
[ ] 0 < (x – xa) * (yd – ya) – (y – ya) * (xd – xa)
[✓] 0 < (xc – x) * (yd – y) – (yc – y) * (xd – x)
[✓] 0 < (xc – xa) * (yb – ya) – (yc – ya) * (xb – xa)
[ ] 0 < (xd – xa) * (yc – ya) – (yd – ya) * (xc – xa)
[✓] 0 < (xd – x) * (yb – y) – (yd – y) * (xb – x)

FIGURE 8.8: Thinning Assumptions

In this goal, there are superfluous assumptions obscuring the relevant facts needed for the proof. This is a common difficulty with interactive proof, and one where the Prover's Palette can help: the **Import** tab provides a way for the user to easily find a minimal set of assumptions which imply the conclusion. In this tab the user can remove assumptions and test whether the resulting statement still holds. If it does, the removed assumptions are not necessary. Following this basic strategy, we can discover (within two minutes) that the Isabelle subgoal above can be simplified as:

$$\begin{array}{l} \mathit{leftTurn}\ s\ t\ r \wedge \mathit{leftTurn}\ s\ r\ q \wedge \\ \mathit{leftTurn}\ s\ p\ r \wedge \mathit{leftTurn}\ t\ p\ q \wedge \\ \mathit{leftTurn}\ t\ q\ r \Longrightarrow \\ \quad \mathit{leftTurn}\ p\ q\ s \end{array}$$

Whenever QEPCAD returns *True* and assumptions were deselected, the widget presents an option in the **Finish** tab to **Thin** the subgoal. Selecting this option causes the redundant assumptions to be removed from the prover goal. Again, no proof dependency on QEPCAD is introduced.

In a similar way, using the **Import** tab to send only selected components of a subgoal to QEPCAD can let us discover whether there are any contradictions within the assumptions. In Isabelle, proving that assumptions are inconsistent is sometimes easier than trying to prove that the conclusion holds. In the verification of Graham's Scan, we found this to be a common situation, and one where often we did not immediately notice the obvious contradiction, especially where case splits were involved. As an example, consider the problem:

```
leftTurn s t r ∧ leftTurn t p q ∧
leftTurn t q r ∧ leftTurn s r q ∧
leftTurn s p r ∧ leftTurn t s r ⟹
    leftTurn p q s
```

Using a similar procedure to that used to find the minimum set of assumptions making the conclusion true, we can easily interact with the QEPCAD widget to discover whether a contradiction is present in the assumptions, and if so what is the minimal such contradictory set. This is achieved by deselecting the conclusion in the **Import** tab so that only the assumptions of the goal are sent to QEPCAD: if QEPCAD then returns "False", we know a contradiction is present. A simple, interactive search can derive the minimal set of contradicting assumptions. For the above example we readily find that the first and last assumptions contradict each other, and we can thin the subgoal appropriately:

```
leftTurn s t r ∧ leftTurn t s r ⟹ False
```

This is obvious in hindsight, but given a proof state with several dozen assumptions, many of which look very similar, and such a contradiction can be hard to spot. Our `simp` rules for `leftTurn` were effective in handling some of these situations, but in some cases they can cause problems, and a technique which points out the contradictory assumptions — particularly one which applies generally and does not depend on domain-specific simplification rules — can be useful to remove the cruft which obscures a proof state's essence.

8.4 Conclusion

In this chapter we have demonstrated a concrete implementation which integrates Isabelle with an external tool in the Prover's Palette, namely QEPCAD. Through illustrative examples we have shown how a user can easily interact with QEPCAD and be assisted in a number of ways while constructing their mechanical proofs.

The design criteria of the Prover's Palette also specified that it should be a modular and extensible system. The next chapter will look at whether that aim has been addressed.

Chapter 9

Maple in the Prover's Palette

One of the objectives of the Prover's Palette framework, as mentioned in Section 7.1.1, is that it should be able to incorporate new tools easily. In this chapter, we will assess the extensibility of our system, looking at the consistency of user experience given and the degree of re-use of code and functionality. This will be considered in respect of a new tool added to the framework: the computer algebra system Maple.

Maple was chosen due to its popularity and the wide breadth of functionality it offers. As QEPCAD already provided us with a powerful method for solving non-linear algebra, we decided to focus on a complementary aspect in Maple: its plotting capabilities. Although the results of plotting cannot be used directly in a proof, there are still a number potential benefits to some proof endeavours. Plots can help a user heighten their intuition about a problem domain, offer a sanity check about whether objects are defined correctly, or indicate if a theorem in not provable. Following discussion of plot support in the Prover's Palette we will present one other mode of using Maple, "assume and check", and conclude with an evaluation of the extensibility of our framework.

9.1 Plotting in the Prover's Palette

As described in Chapter 7, our implementation of the Prover's Palette supplies abstract superclasses which encapsulate common functionality, including defining the View and tabs, tracking and translating the proof state, modifying the proof state where necessary (e.g. expanding definitions, converting to PNF), and inserting results in various ways (via an oracle, by instantiation, etc). The intention was that this will minimise the additional code required to integrate a new tool such as Maple into the framework.

This integration will be explored through the collision problem posed by Collins and Hong, described in the previous chapter. Recall that this problem queries whether a moving circle (1) and a moving square (2) will ever collide, specifically asking whether $\exists\, t\, x\, y.\, t \geq 0 \wedge (1) \wedge (2)$:

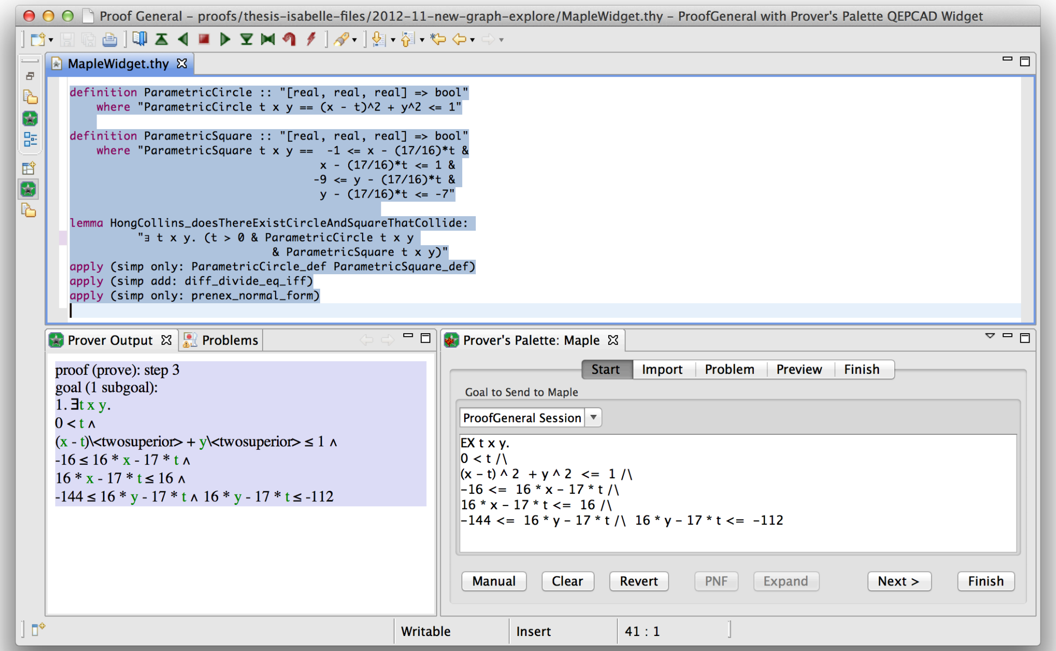
$$(1) \quad (x-t)^2 + y^2 \leq 1$$

$$(2) \quad -1 \leq x - \tfrac{17}{16}t \leq 1 \wedge -9 \leq y - \tfrac{17}{16}t \leq -7$$

As we mentioned previously, if we rely solely on a theorem prover, this can be a difficult to analyse, but QEPCAD in the Prover's Palette can provide significant assistance, as we have seen in Section 8.2.2. Plotting in Maple using our system can provide a different form of assistance. Using this example, we will highlight how the Maple integration works and call out where parts of the framework were reused and where new code was developed for this new integration.

9.1.1 Consistent User Experience

Let us begin by showing how the Maple View sits in the Eclipse PG environment. Just as with the QEPCAD widget, this View is part of an Eclipse plug-in which subscribes to PG events, and when a new problem is processed by the prover, it becomes available in the Maple View. Figure 9.1 shows this View with the Collins-Hong collision problem.



```
definition ParametricCircle :: "[real, real, real] => bool"
    where "ParametricCircle t x y == (x - t)^2 + y^2 <= 1"

definition ParametricSquare :: "[real, real, real] => bool"
    where "ParametricSquare t x y ==  -1 <= x - (17/16)*t &
                                       x - (17/16)*t <= 1 &
                                      -9 <= y - (17/16)*t &
                                       y - (17/16)*t <= -7"

lemma HongCollins_doesThereExistCircleAndSquareThatCollide:
          "∃ t x y. (t > 0 & ParametricCircle t x y
                              & ParametricSquare t x y)"
apply (simp only: ParametricCircle_def ParametricSquare_def)
apply (simp add: diff_divide_eq_iff)
apply (simp only: prenex_normal_form)
```

```
proof (prove): step 3
goal (1 subgoal):
1. ∃t x y.
0 < t ∧
(x - t)\<twosuperior> + y\<twosuperior> ≤ 1 ∧
-16 ≤ 16 * x - 17 * t ∧
16 * x - 17 * t ≤ 16 ∧
-144 ≤ 16 * y - 17 * t ∧ 16 * y - 17 * t ≤ -112
```

```
EX t x y.
0 < t /\
(x - t) ^ 2  + y ^ 2  <=  1 /\
-16 <=  16 * x - 17 * t /\
16 * x - 17 * t <=  16 /\
-144 <=  16 * y - 17 * t /\  16 * y - 17 * t <=  -112
```

FIGURE 9.1: The Prover's Palette with Isabelle and Maple

At this stage, the View is nearly identical to that of the QEPCAD widget: there are five of the same tabs, **Start**, **Import**, **Problem**, **Preview** and **Finish** (with only the **Config** tab not used in this integration), and as before, the View can be minimised so that it remains out-of-sight except when relevant. It can be restored manually or configured to display automatically when it is applicable to a proof subgoal.

This consistency ensures that a user who has become familiar with one tool in the Prover's Palette system can apply that process to a new tool. The tabs, with View behaviour, and the manner in which she can engage with the external tool do not need to change dramatically, nor is any special knowledge of a new tool required.

From the developer's perspective, additionally, we have so far not required much new code at all. The framework's abstract classes provide nearly all the functionality just described, with the only development burden here the skeleton concrete classes and the plug-in definition for the new tool.

9.1.2 Maple Running Automatically

Looking specifically at the **Start** tab, the abstract super class *`StartTabComposite`* provided us with the following:

- **PNF** button and all behaviour (converting to prenex normal form)
- **Expand** button and all behaviour (expanding predicates and functions)
- **Manual** button and all behaviour (enables manual edit mode)
- **Clear** button and all behaviour (clears the widget, for new problems)
- **Revert** button and all behaviour (resetting the widget to the current proof state)
- **Next** button and some behaviour
- **Finish** button and some behaviour

Where Maple-specific code is required for this tab is in determining which of the buttons should be enabled, based on the predicates and functions recognised by Maple, and in implementing the behaviour for some of the buttons (**Next** and **Finish**). For the most part, however, this does not require code changes in the classes associated with this tab: the behaviour of **Next** and **Finish** is defined by the invoked tabs, and enablement is computed based on calls to the interface *`MathsSystemTranslator`*. The knowledge of which proof components are valid in Maple is contained within the translation module implementing this interface. We will discuss this translation — the first major area where development was required — in Section 9.1.4; for now let us assume it is completed.

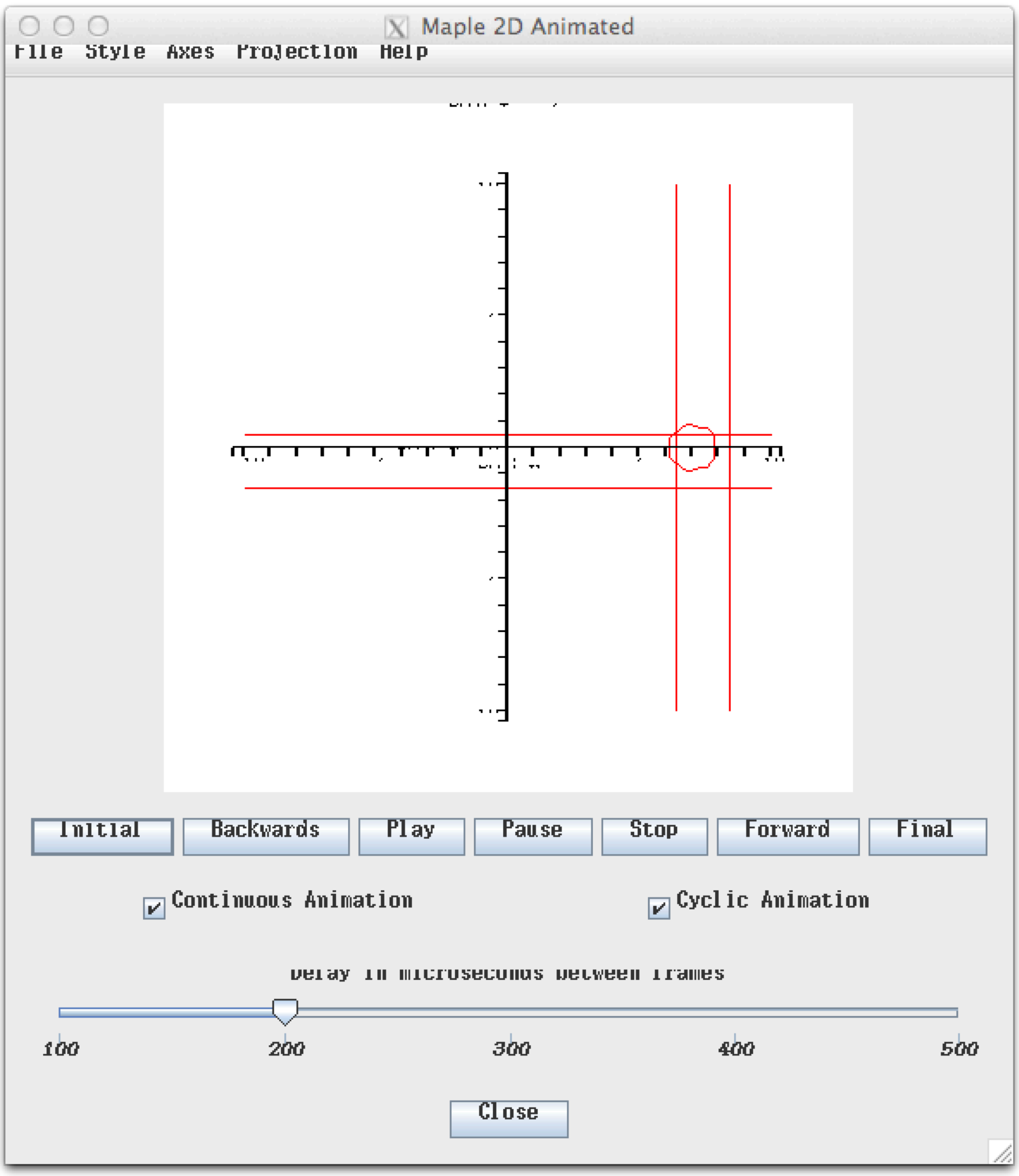


FIGURE 9.2: The Maple animation of the circle and square collision

Conveniently for the user, the Maple widget not only looks like the tab in the QEP-CAD widget, it has the same behaviour as well. For the collision problem, the **Finish** button is initially disabled and the **Expand** button is enabled, because Maple does not understand the predicates `ParametricCircle` and `ParametricSquare`. As before, clicking the **Expand** button inserts the appropriate commands into the proof script and directs Isabelle to process them; further manipulation, such as to remove division, can be done if desired; and clicking the **PNF** button converts the subgoal to prenex normal form. This brings us to the state shown in Figure 9.1, with the **Finish** button enabled and ready to send the problem off for plotting.

Upon clicking the **Finish** button, the Maple process is started and the commands for producing a plot of the current problem are sent off. This brings us to the second major area where development was required, generating the commands to be sent to this new external tool. We will discuss this in the next section, but again assuming this is working, Maple executes the command and the Prover's Palette tracks the external process. The **Finish** tab is displayed — but as there is very little to do this contains little more than a comment (and the corresponding code is mainly a "no-op" subclass of the core `FinishTabComposite`). The action comes from Maple, which generates a new window for the collision problem, containing an interactive animation of the circle and square moving over time. Figure 9.2 shows a still of this animation where the circle and square are colliding (t=5.8333). The integration lets us easily bring this up, without our having to be familiar with Maple or having to perform any manual translation activities. Without disrupting our concentration in the theorem prover, we have access to an animation which gives insight into the problem, confidence that we have defined the equations correctly (as we see the circle and square), and security that the problem is provable.

9.1.3 Tool Commands and the Preview Tab

As an alternative to selecting the **Finish** button in the **Start** tab, the user could choose to go to the **Preview** tab. As before, this tab lets the user inspect the command which will be sent to the tool; for Maple, and the collision problem just described, the script generated is shown in Figure 9.3.

Script generation to support Maple, as mentioned in the previous section, required more new development than any other single activity. As the command script is intimately tied to the way Maple is being used on a problem (explicitly shown in the **Problem** tab), the code for generating the script is in the corresponding class `ProblemTabModeImplicitPlot`. Note that this is slightly different to how it is handled in QEPCAD, where it is the responsibility of a dedicated class `QepcadProblem` (Section 8.1.4); this is because in QEPCAD, the form of the tool script is largely the same, whereas with Maple it varies widely based on the mode of problem solving selected. Section 9.1.4 describes the available problem modes and the automation therein — such as choosing the equations to be plotted and the variables to be used for each axis. Once that is clear, the construction of the script is relatively straightforward, essentially instantiating a template with the selected variables and equations. The script

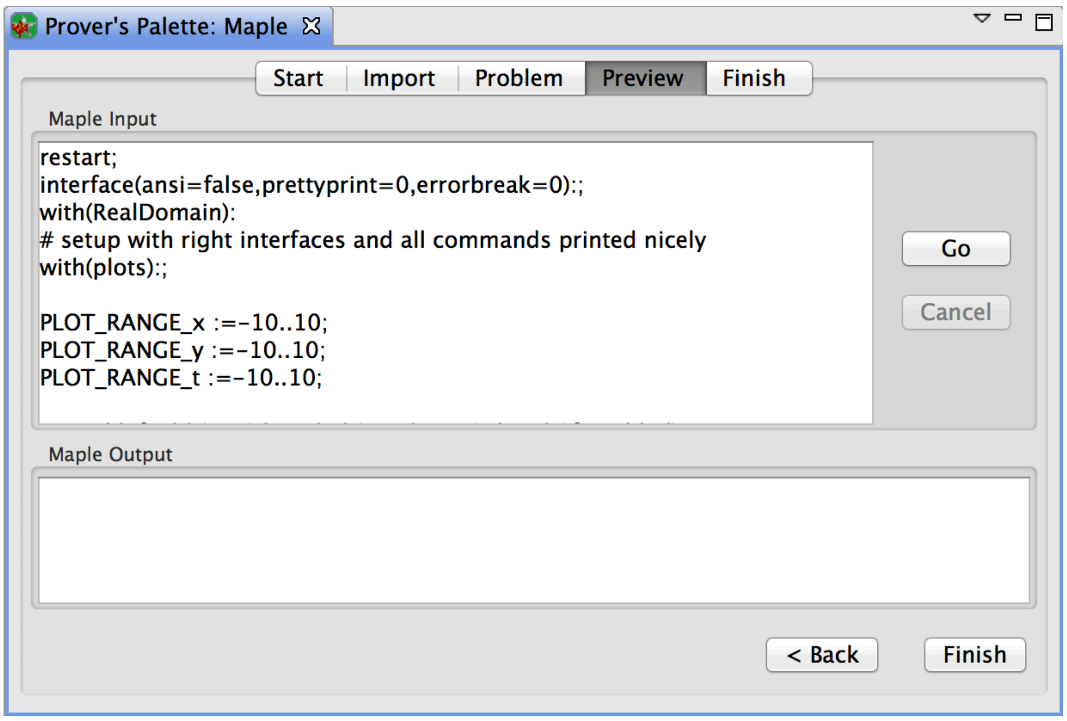
```
restart;
interface(ansi=false,prettyprint=0,errorbreak=0):;
with(RealDomain):
with(plots):;

# ranges:
PLOT_RANGE_x :=-10..10;
PLOT_RANGE_y :=-10..10;
PLOT_RANGE_t :=-10..10;

# modes: 'maplet'; 'default' (text); 'x11' (if enabled)
plotsetup(maplet):;

# internal code for simultaneous plot support
#(do not modify unless you know what you are doing)
x := PLOT_x;
y := PLOT_y;
t := PLOT_t;
interface(echo=0):; # must set echo off and wrap output
printf("BEGIN[Prover's Palette Result]\n\n");
PLOT_eq1 := animate(implicitplot,
    [ ((((x - t) ^ 2) + (y ^ 2)) = 1),
    x=PLOT_RANGE_x, y=PLOT_RANGE_y ], t=PLOT_RANGE_t):;
PLOT_eq2 := animate(implicitplot,
    [ ((- 16) = ((16 * x) - (17 * t))),
    x=PLOT_RANGE_x, y=PLOT_RANGE_y ], t=PLOT_RANGE_t):;
PLOT_eq3 := animate(implicitplot,
    [ (((16 * x) - (17 * t)) = 16),
    x=PLOT_RANGE_x, y=PLOT_RANGE_y ], t=PLOT_RANGE_t):;
PLOT_eq4 := animate(implicitplot,
    [ ((- 144) = ((16 * y) - (17 * t))),
    x=PLOT_RANGE_x, y=PLOT_RANGE_y ], t=PLOT_RANGE_t):;
PLOT_eq5 := animate(implicitplot,
    [ (((16 * y) - (17 * t)) = (- 112)),
    x=PLOT_RANGE_x, y=PLOT_RANGE_y ], t=PLOT_RANGE_t):;
printf("\nplot should appear shortly;
    \ncancel or kill maple to close\n\n");
display({PLOT_eq1,PLOT_eq2,PLOT_eq3,PLOT_eq4,PLOT_eq5});
printf("\nEND [Prover's Palette Result] \n");
```

FIGURE 9.3: The Maple script produced to plot the collision problem

generation code can be found in the method `updateGuiOnSelectionChange()` in the `ProblemTabMode` class.

The important functionality in the **Preview** tab, shown in Figure 9.4, is that allowing the user:

- to inspect the actual script which will be sent to Maple;
- to change the plot range on variables (at the top of the script);
- to change the constant values assigned to variables which are not plotted (also at the top of the script, but not shown in this example);



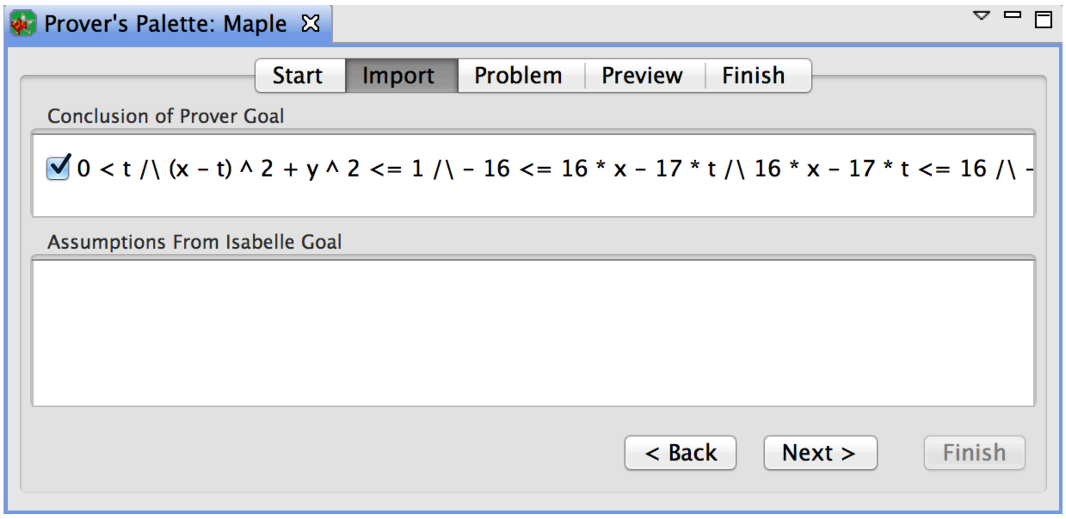
```
restart;
interface(ansi=false,prettyprint=0,errorbreak=0):;
with(RealDomain):
# setup with right interfaces and all commands printed nicely
with(plots):;

PLOT_RANGE_x :=-10..10;
PLOT_RANGE_y :=-10..10;
PLOT_RANGE_t :=-10..10;
```

FIGURE 9.4: The Preview Tab of the Maple widget

- to make further edits where desired, such as applying colours, labels, or even adding additional equations (all for the advanced Maple user); and
- to cancel the Maple process (*e.g.* if it hangs running remotely with `ssh -X`)

This is analogous to the behaviour of the tab in the QEPCAD widget, and indeed the code here re-uses much of the code from that implementation. The superclass `PreviewTabComposite` defines the tab, and we have extended that re-using code — here from the QEPCAD widget — to define the GUI text boxes within it, again with the **Maple Input** box editable so that a user familiar with Maple may edit the script. The GUI objects for the buttons (**Go** and **Cancel**, **Back** and **Finish**) were also generated by re-using code from the QEPCAD widget, rather than the core framework, whereas their behaviour (managing the external process) *was* mainly supplied by the core framework. The behaviour for monitoring the output from the process and rendering it in the **Maple Output** textbox was also supplied by the core framework.

One other small area of new code required is to define `MapleProcess`, extending `ProversPaletteAbstractExternalProcess` to declare the process to invoke (`maple`) and to describe how to analyse the output (trivial in the case of plotting). Thus, to support the fully automated integration shown so far, the areas of new development were precisely those areas unique to the tool Maple: besides housekeeping (creating the skeleton classes and plug-in project, which the IDE does automatically) and wiring



0 < t /\ (x - t) ^ 2 + y ^ 2 <= 1 /\ - 16 <= 16 * x - 17 * t /\ 16 * x - 17 * t <= 16 /\ -

FIGURE 9.5: The Import tab of the Maple widget

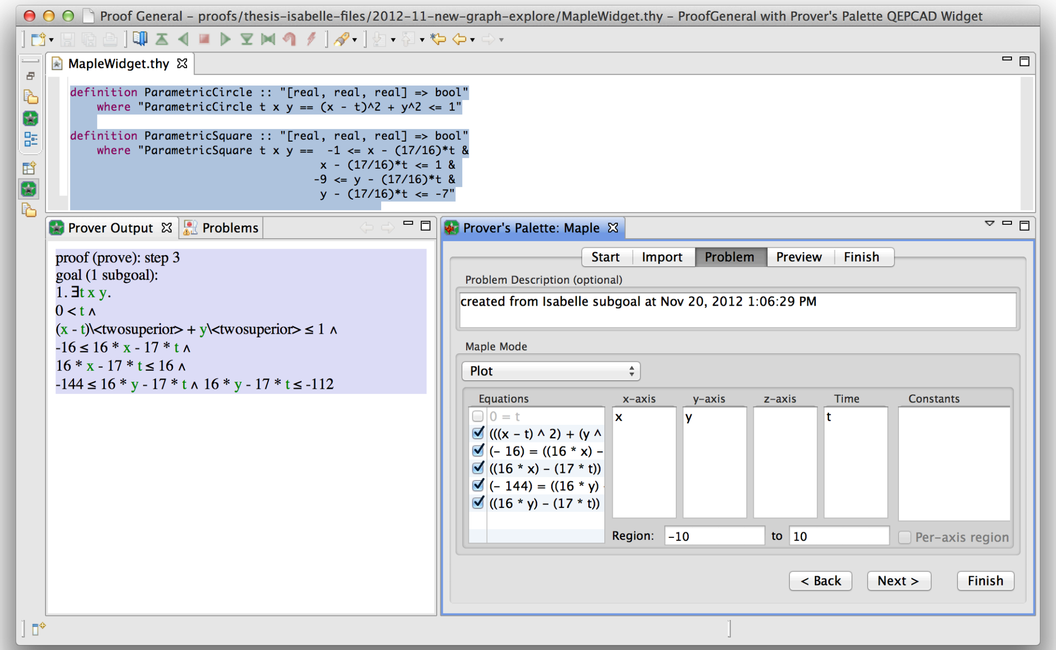
(*e.g.* specifying that the process to call is `maple`), new code is almost entirely around translation and the script generation. In fact, we were able to re-use even more GUI items than we expected from the QEPCAD widget, as much of the Maple code for the GUI shown so far was created by cut-and-paste: there is a strong case to refactor and lift this code to be part of the Prover's Palette core (abstract) framework.

This shows that we achieved a high degree of re-use, confirming our aim of making the tool extensible and modular. Let us now drill down into the new developments, translations and more advanced interactive usage and the corresponding scripts generated.

9.1.4 Translations and Interactive Usage

We will now proceed through the tabs in order to show what is supported interactively. After the **Start** tab, we first encounter the **Import** tab (Figure 9.5). As with the QEPCAD widget, this tab allows components — the conclusion and/or each of the assumptions (though there are none in this case) — to be selected for inclusion. The code for this tab (in `ImportTabComposite` in our `maple` widget Java package) is largely similar to that of the QEPCAD variant, with one significant change: the "variables" composite where bound and free variables are exchanged is not applicable has been removed. Again, the fact that we have re-used the design patterns and large portions of code shows the extensibility of the system's design, and it indicates where yet more of the work from QEPCAD could be lifted and provided as part of the shared Prover's Palette core.

Let us now turn our attention to the **Problem** tab, where the goals translated into Maple's syntax are displayed for the first time (Figure 9.6). Translation, in the class `MapleTranslator` as previously mentioned, follows the same approach as taken for QEPCAD: an internal goal state representation is constructed using the core framework, and this new module translates to and from that representation. For plotting, quantification is ignored; a variable defines the same curve whether it is existential or universal. Arithmetic and logical operators are converted to Maple syntax, and nearly all other functions are unsupported[1], so the translation is relatively simple.



Proof General – proofs/thesis-isabelle-files/2012-11-new-graph-explore/MapleWidget.thy – ProofGeneral with Prover's Palette QEPCAD Widget

MapleWidget.thy

```
definition ParametricCircle :: "[real, real, real] => bool"
    where "ParametricCircle t x y == (x - t)^2 + y^2 <= 1"

definition ParametricSquare :: "[real, real, real] => bool"
    where "ParametricSquare t x y ==  -1 <= x - (17/16)*t &
                                       x - (17/16)*t <= 1 &
                                      -9 <= y - (17/16)*t &
                                       y - (17/16)*t <= -7"
```

Prover Output

```
proof (prove): step 3
goal (1 subgoal):
1. ∃t x y.
0 < t ∧
(x - t)\<twosuperior> + y\<twosuperior> ≤ 1 ∧
-16 ≤ 16 * x - 17 * t ∧
16 * x - 17 * t ≤ 16 ∧
-144 ≤ 16 * y - 17 * t ∧ 16 * y - 17 * t ≤ -112
```

Prover's Palette: Maple

Problem Description (optional): created from Isabelle subgoal at Nov 20, 2012 1:06:29 PM

Maple Mode: Plot

Equations	x-axis	y-axis	z-axis	Time	Constants
☐ 0 = t	x	y		t	
✓ (((x – t) ^ 2) + (y ^					
✓ (– 16) = ((16 * x) –					
✓ ((16 * x) – (17 * t))					
✓ (– 144) = ((16 * y)					
✓ ((16 * y) – (17 * t))					

Region: -10 to 10 ☐ Per-axis region

FIGURE 9.6: The Problem Tab of the Maple widget

The translated subgoal can be decomposed further to make it applicable for plotting, breaking up conjuncts and converting inequalities to equalities[2], as can be seen in the **Problem** tab (the same screenshot, Figure 9.6).

The widget relies on the translation routines to identify which equations from the current subgoal are suitable for plotting, and uses heuristics — based on the presence

[1]We have included support for `sqrt`, as it is part of the commonly-used Isabelle theories. Functions from exponentiation, logarithms, and trigonometry are not translated, even though they are supported in Maple, because they are used less frequently in Isabelle and can have greater variance in their definition. The prospect of extending integrations to be aware of theory-specific semantics is discussed as potential future work in Section 11.2.4.

[2]Maple does offer a package to plot inequalities, namely `plots[inequal]`, but it only plots linear inequalities, which are of limited use, and the resulting plots are little better than the more general plots we display. For this reason we have not automated this functionality in the Maple widget.

of an `x`, `y`, `z` or `t` in variable names — to determine which ones will be plotted along the axes and which ones will be converted to constants. The user can override this, easily moving variables between columns using drag-and-drop, in order to correct mistakes in the heuristics or to explore different relationships.

Further heuristics are used to determine automatically, based on the variable assignments, which plot type and command is appropriate, choosing by default among two supported types of Maple plots:

- `ImplicitPlot` for 2D plots (two variables plotted)
- `ImplicitPlot3D` for 3D plots (three variables plotted)

As hinted by the reference to `t`, the widget also supports Maple's `animate` function to display a visualization for one additional dimension, as shown earlier in Figure 9.2.

As an illustrative example, consider the collision problem again. The Maple integration defaults to a two-dimensional animation and automatically determines the variables for each axis: `x` is plotted against the x-axis, `y` against the y-axis, and `t` represents time. The widget also deduces that five out of the six constituent equations should be plotted. In this plot configuration, the equation $t = 0$ is uninteresting and is deselected.

Once the selections are confirmed and the user proceeds to the **Preview** or **Finish** tab, the script to Maple is generated. As previously described, this is based on templated lines of Maple commands: this template is parameterised based on the selected plot type, equations, and variable-to-axis assignments. This construction is similar in approach to what the QEPCAD widget does, but the details here are necessarily different and the code in the Maple variant of this tab is largely new. Thus the two most significant areas of new code are the translation, in `MapleTranslator`, and the problem-type-specific behaviour in `ProblemTabModeImplicitPlot`.

9.2 Adding More of Maple's Functionality

We were principally interested in Maple for plotting, but of course there is a wealth of other functionality available in Maple. We have shown how the Prover's Palette supports extension to use a new external tool, but let us also assess how extensible the Maple integration itself is. Can we support other ways of using Maple which could be useful when constructing a formal proof?

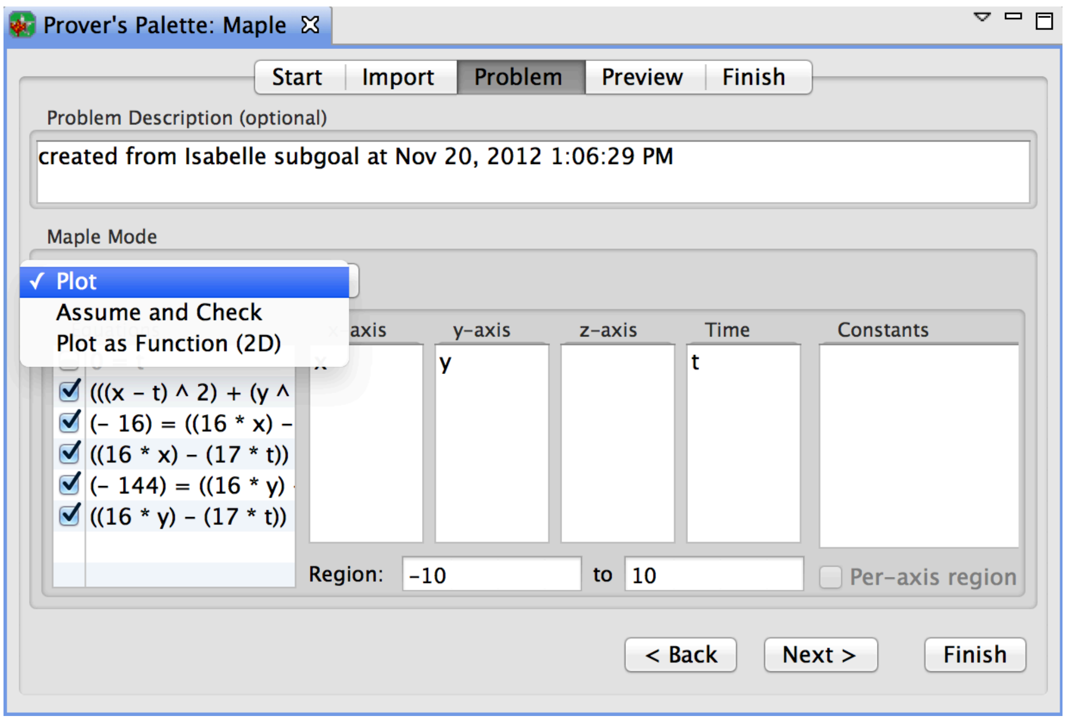
To enable extensions to the Maple integration, we have left the `ProblemTabComposite` subclass a thin shim, calling out to a `Mode` class to determine what GUI elements and behaviour should be in effect. Section 9.1.4 described the code in the class `ProblemTabModeImplicitPlot`, where nearly all the code relevant to that section is contained; the Maple `ProblemTabComposite` simply contains enough to render a description textbox (actually inherited from the superclass in the framework) and a **Maple Mode** box with a drop-down where the user can choose the desired problem mode. The general **Plot** mode described so far is the default, but this provides a hook for additional functionality. Figure 9.7 shows this dropdown and the available problem modes.

9.2.1 Explicit Plots

Sticking with plots to begin with, there is a simpler plot mode in Maple sometimes preferred by users: for two-dimensional plots, instead of using `implicitplot`, a user can select Maple's default function `plot` by choosing **Plot as Function** (Figure 9.7). This brings up a simpler GUI and generates a simpler script which can invoke Maple's `solve` routine to convert an equation to a function in a single dependent variable.

This behaviour is defined in the class `ProblemTabModePlot2d`, similarly to the implicit-plot mode class described, but with the following differences:

- different GUI components are displayed (fewer and simpler); and
- a different script is generated (again, smaller and simpler)



Problem Description (optional)
created from Isabelle subgoal at Nov 20, 2012 1:06:29 PM

✓ (((x − t) ^ 2) + (y ^
✓ (− 16) = ((16 * x) −
✓ ((16 * x) − (17 * t))
✓ (− 144) = ((16 * y)
✓ ((16 * y) − (17 * t))

x y t

Region: -10 to 10

FIGURE 9.7: The Maple modes which are supported by the widget can be selected from the dropdown menu in the Problem tab.

This shows how the Maple integration can be extended to support additional functionality, but admittedly this functionality — still plotting — is quite similar. We will describe one further problem mode which is very different in nature, to demonstrate that this extension mechanism generalises.

9.2.2 Assume and Check

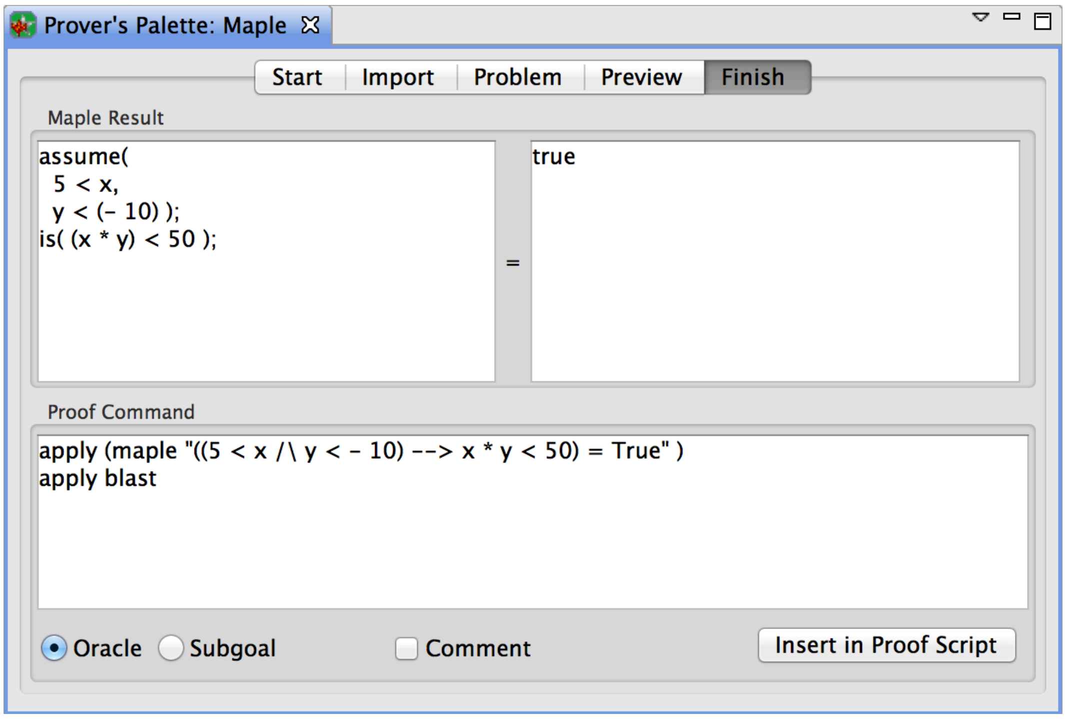
Maple includes some logical reasoning functionality where a user can supply assumptions then ask whether a conclusion is valid. This, in theory, is neatly applicable to theorem proving subgoals. In Isabelle, many subgoals are of the form:

```
[| assumption1 ; ... ; assumptionN|] ==> conclusion
```

In Maple, this can be queried using the template script:

```
assume(assumption1, ..., assumptionN);
is(conclusion);
```

Our integration exposes an **Assume and Check** mode in the **Problem** tab which, for supported assumptions and conclusions, generates a script of the form above. Figure 9.8 shows such a script for a sample problem, together with the result in the **Finish** tab.



```
assume(
 5 < x,
 y < (- 10) );
is( (x * y) < 50 );
```

```
true
```

```
apply (maple "((5 < x /\ y < - 10) --> x * y < 50) = True" )
apply blast
```

[● Oracle] ○ Subgoal ☐ Comment

FIGURE 9.8: The **Assume and Check** mode of the Maple widget

As with the QEPCAD widget, there are options to use the result in either **Oracle** or **Subgoal** mode, and the widget can be configured so that it runs in the background and pops up if and when an answer is found. The majority of this behaviour was inherited from the core framework's `FinishTabComposite`, although a small amount of change was needed from the trivial no-op subclass used for plotting to ensure that the assume-and-check result could be applied to this problem mode. The most significant such change was ensuring that the Maple script generated clearly identifiable markers in the textual output — `BEGIN[Prover's Palette Result]` as seen in Figure 9.3 — so that the result could be extracted by the Prover's Palette.

Unfortunately, although constructs of Maple's assume-and-check command are similar in theory to those in theorem proving, in practice it was not applicable because Maple's capabilities in this direction are severely limited. In our experiments, it could not handle real-world equations such as those we sent to QEPCAD, and in fact it could not manage much more complexity than the toy problem shown in Figure 9.8.[3]

9.2.3 Related Work

Despite the limitations of Maple's *assume* capabilities, that feature gives a good template for other functionality which could be added, specifically based on the facets of Maple which people find most useful. Although our method of systems integration is dramatically different to what has been done previously, some of the non-interactive integrations between Maple and theorem provers are useful to inform this selection:

- Harrison *et al.* [83] call Maple from HOL to simplify algebraic expressions and find factorisations for polynomials
- Ballarin *et al.* [13] call Maple from Isabelle to extend the capabilities of the simplifier and solve summation and induction problems

Having an interactive system could be particularly useful here, as one common challenge with such approaches is identifying which terms to simplify in the external tool. A GUI tool which allows the user to select terms within a subgoal — at a finer granularity than simply the assumptions — could be added as a library module without disrupting the design, and this strikes us as a good potential extension to the Prover's Palette.

[3]Due to the limited utility of this problem mode, and because Maple often operates over the complex numbers which are often not loaded in Isabelle theories, we have not applied the same technique of attaching the external tool types to the result script inserted in the proof. If it is possible to improve this technique, this type information could easily be inserted as the functionality is in the Prover's Palette core.

9.3 Conclusion

The aim of this phase of research was to investigate whether the approach and framework of the Prover's Palette is extensible and modular. This, we feel, has been demonstrated:

- the amount of new code for the new tool, Maple, is relatively small: 3825 lines, compared with 6374 for the core and the QEPCAD integration together[4];
- most of this code is necessarily new, corresponding to specific semantics or capabilities of the new external tool; and
- a breadth of the functionality of the new tool is made available.

One indication that the design is successfully extensible is the emergence of new capabilities which should be added to the core and used across tools, and one indication that it is modular is a clear manner in which this can be done. We observed that significant parts of the code implemented for the QEPCAD widget were applicable to the Maple widget, implying that there is even more scope for generalisation than we had first assumed, and it is clear where this should go (into the respective *`TabComposite`* superclasses in the core). Furthermore, we noted a new feature, GUI support for selecting fine-grained terms from within the subgoal, which could be useful both for Maple and QEPCAD, and again this can be cleanly added.

There were instances where the framework-supplied model was not a perfect fit, as in the use of the **Finish** tab for plotting when there are no modifications applicable to the proof script. It could be argued that the framework over-generalised in this case, but equally it could be argued that the implementation simply took the easy option: it could have omitted that tab or provided a more appropriate set of widgets. (Ideally we would have liked the plot from Maple to be embedded within this tab, but the technology behind such low-level window embedding is beyond the scope of this work!) The framework does not require that tabs be used, so in conclusion even with this critical observation, we feel we found a good balance between reusability and customisability.

Most importantly, many of the benefits of the Prover's Palette called out in Chapters 7 and 8 were found to apply to Maple: the integration allows for quick confirmation of definitions and lemma statements as well as detailed exploration of a problem domain; the integration supports fully automated processing and interactive usage; and

[4]The core at present consists of 2864 lines, and the QEPCAD plugin 3510. However approximately 800 lines of code used in both QEPCAD and Maple (cut-and-pasted) have been identified which should be removed from those projects and inserted into the core instead; this is not reflected in these line counts. Furthermore it must be noted that there are three distinct areas of functionality supported by the Maple integration.

the integration can be extended naturally to onboard new tools and new functionality of these tools. This supports our hypothesis that the user is most empowered when a palette of tightly integrated yet customisable tools is conveniently at their disposal. A successful integration framework — and on many levels the Prover's Palette can be so counted — is one which enables a deeper understanding of the proof, freeing the user from the burden of tedious, mundane proof details, but without overly restricting what is possible.

Chapter 10

Case Study: Verifying Delaunay Triangulation

In Chapters 8 and 9, we showed several individual examples where the Prover's Palette system is helpful. Let us now look at whether this cohesively integrated suite of mathematical tools is actually useful during the heat of battle. In particular, this chapter will consider if and how the availability of QEPCAD within the Prover's Palette facilitates the proof process when developing real-world theories in Isabelle. For this case study, we have began developing a formal proof for the correctness of a planar Delaunay triangulation algorithm.

10.1 Why Choose Delaunay Triangulation?

Delaunay triangulation is interesting because, like the convex hull, it is a cornerstone of computational geometry. It is widely used in animation and geographical modelling, and in a range of scientific enquiries, from astrophysics to weather prediction, where discrete point samples are taken from a continuous domain. Furthermore, given a Delaunay triangulation, it is easy to compute another important structure in the field: the Voronoi diagram, which is the dual of the Delaunay triangulation and is useful for solving the nearest-neighbour problem, with applications in biology, ecology, physics, and logistics [138].

It was anticipated that the theories necessary for developing this formalised proof would build upon much of the work done for the Graham's Scan proof, and that it would present many of the same issues described in Chapter 6. For this reason, we expected it to be a good test case for determining whether the Prover's Palette makes the interactive, formal proof process more intuitively accessible and quicker for these particularly challenging problems.

Indeed, as we shall show, the Prover's Palette is helpful throughout this process, both initially as we extend and introduce the base concepts, and as we move on to the time-consuming process of formulating the correct loop invariants and ultimately proving the necessary verification conditions. As with the Graham's Scan proof, discovering the loop invariants has been a complex, iterative process, where more and more minor details have to be incorporated as the proof progresses. Again, as the size and complexity grows, the task becomes more tedious and the problems at hand become slippery to the intuition: with the Prover's Palette in our arsenal, however, we found that we are able to quickly move past minor details and keep our intuition more closely focussed on the larger issues.

In this chapter, we demonstrate the utility of the Prover's Palette in each of the three distinct phases we have followed in producing this proof so far. We begin by reviewing and updating definitions and lemmas from Graham's Scan which we know we will need to bring in, such as signed area. The Delaunay triangulation is then defined in the second section, informally and then formally in Isabelle. The formal interpretation is accompanied by the necessary definitions, such as the notion of a circumcircle, and some related lemmas. We then present the edge-flipping algorithm for computing Delaunay triangulations and the candidate invariants we have began exploring. Each section also gives concrete examples for how the Prover's Palette has significantly aided our investigation or simplified our task.

10.2 Extending and Updating Theories

Before we began work on the Delaunay triangulation, we embarked on a round of house-keeping for our existing theories. It was clear that many of the same definitions and lemmas we developed in Isabelle for Graham's Scan would apply to this new task, but there were compelling reasons to catch up on technical debt accumulated along the way. For one thing, Isabelle had evolved significantly since we began with Graham's Scan, from version Isabelle2003 to Isabelle2012-1 and our theories needed to be updated to remain compatible[1]. Secondly, we had built up new knowledge for how the concepts could be used and felt it would be beneficial to extend the theories for broader and simpler applicability. This conclusion was reinforced as we continued to

[1]Although we had done other work with intermediate versions, much of the theory had remained in a constant version of Isabelle, Isabelle2005.

use the Prover's Palette and discovered that other variants of our geometric definitions are more amenable to tools such as QEPCAD.

When we began the process of updating to the latest Isabelle, we had no idea how substantial the effort to update our theories — `SignedArea` in particular — would be. Whilst as early as Isabelle2007 we had discovered that our proofs were no longer compatible, we did not realize how sweeping the changes to the underlying libraries were. As noted in Section 5.1.1, one major change was a restructuring of the theory `Real`, which was updated to use Isabelle's axiomatic type classes. As a consequence, many of the lemmas we had relied upon had moved, been renamed, or altered, to such an extent that a large number of proofs no longer processed. In addition, there were changes to the syntax which were relatively easy to fix and changes to underlying ML calls which were harder to fix. Finally, the default simplification rules had changed — drastically so in some of the version updates — and numerous places where `simp` and `auto` had been used in our proofs also required attention.

To show the relevance of our system, however, it would be necessary to run with a more up-to-date version[2]; and in any event it feels foolish to embark on a new project with a software version several years out of date!

Some of our old lemmas could be automatically proven by our new simp sets, and by updated simplification rules in Isabelle, but the vast majority were not. Updating the myriad details of the remaining broken proofs — where the referenced libraries have been completely overhauled — was not an appealing prospect. Consequently, we concluded that it would suffice to use QEPCAD as an oracle within the Prover's Palette, trusted to "prove" the theorems. Additionally, we were pleased to discover that Isabelle's Sledgehammer had considerably improved since we had first experimented with it, and it could now prove many of our signed area lemmas. For some of the problems it could not solve we relied on QEPCAD, and found that a happy marriage could be had between Sledgehammer and the Prover's Palette.

Unfortunately, there were several lemmas which remained tricky to prove, they are either not applicable to QEPCAD, or of too great a complexity to be solved quickly. QEPCAD showed versatility for the types of problems we were encountering, but for many of them, getting them into algebraic form was beyond the scope of the Prover's Palette automated assistance (merely expanding unknown predicates and converting to PNF), and in some cases when expanded they contained a dozen or more free variables.

[2]We have confirmed the compatibility of the Prover's Palette with Isabelle2012-1.

Theorems about the `isBetween` concept were particularly tedious. Recall the original definition:

```
definition isBetween :: "[ point, point, point] ⇒ bool"
                            ("_ isBetween _ _ " [60, 60, 60] 60)
    where "b isBetween a c ≡
            a≠c ∧ collinear a b c ∧
            (∀d. signedArea a c d ≠ 0 ⟶
                0 < signedArea a b d / signedArea a c d ∧
                signedArea a b d / signedArea a c d < 1 )"
```

The advantage of this definition is that it contains an easy to grasp geometric interpretation. However, expanding this definition results in two new quantified variables (one new point). When we proceed to massage it into its equivalent algebraic form we get a statement containing division — anathema to QEPCAD and numerous other CASs. To reduce the amount of user interaction in this massaging process, we introduced a lemma for replacing `isBetween` facts with a CAS-friendly algebraic representation[3]:

```
isBetweenAbsForQepcad: <xx,xy> isBetween <ax,ay> <bx,by> =
    ( ( (xx-ax)*(by-ay) = (bx-ax)*(xy-ay) ) ∧                  (1)
      ( (ax < xx ∧ xx < bx) ∨ (ax > xx ∧ xx > bx) ) ∧          (2)
      ( (ay < xy ∧ xy < by) ∨ (ay > xy ∧ xy > by) ) )          (3)
```

This lemma says that if a point $X = \langle x_x, x_y \rangle$ `isBetween` points $A = \langle a_x, a_y \rangle$ and $B = \langle b_x, b_y \rangle$, then we can represent this fact algebraically by saying the three points must be collinear (1), the x-coordinate of X must lie between the x-coordinates of A and B (2), and the y-coordinate of X must lie between the y-coordinates of A and B (3); this is not the algebraic expansion of the original definition but a restatement which seeks to get rid of the additional point. However, once we had massaged the `isBetween` fact into algebraic form, QEPCAD informed us that this lemma was false! It also produced a counterexample where $A = \langle -1, -1 \rangle$, $B = \langle -8, -1 \rangle$ and $X = \langle -5, -1 \rangle$. A quick illustration of this counterexample shows that our algebra had not taken into account the case when all three points lie on a horizontal line, i.e. when their y-coordinates are equal.

Similarly, we realised that we had also not included the case when the points are collinear on a line which is vertical. To explore this, we specified some variables to remain free when passed to QEPCAD, making use of the customisation afforded by the Prover's Palette, and were able to confirm that the restated problem reduced to the vertical collinearity case. Thus handling these edge cases (when the inequality is not strict) should be sufficient to make the lemma true. After correcting this oversight, we

[3]The reader will also note the switch to `<x, y>` notation for points. This was part of the housekeeping and modernization.

have the following algebraic equivalence, which this time QEPCAD is able to prove for us:

```
isBetweenAbsForQepcad: <xx,xy> isBetween <ax,ay> <bx,by> =
        ( ( (xx-ax)*(by-ay) = (bx-ax)*(xy-ay) ) ∧
          ( (ax < xx ∧ xx < bx) ∨ (ax > xx ∧ xx > bx) ∨
                (ax = xx ∧ xx = bx) ) ∧
          ( (ay < xy ∧ xy < by) ∨ (ay > xy ∧ xy > by) ∨
                (ay = xy ∧ xy = by) ) ∧
          ( ¬ (ax = bx ∧ ay = by) ) )
```

This corrected lemma then made our process of proving many of the lemmas in the `SignedArea` theory much easier.

10.3 Defining Delaunay Triangulation

With our theories all up-to-date, let us introduce the Delaunay Triangulation and its underlying concepts, before presenting our formalisation in Isabelle. As with updating our underlying theories, we will show how the Prover's Palette can be helpful.

10.3.1 A Mathematical Definition

A *triangulation* of a set of points is a maximal set of non-intersecting line segments between points in the set. In other words, a triangulation of planar point set Q is the partitioning of the region bounded by the convex hull of Q into a set of disjoint triangles whose vertices are all in Q; and where the points in Q are all collinear, no triangulation is admitted.

A *Delaunay triangulation* is a special case where the triangulation is particularly "nice"; that is, it avoids the more obtuse triangles and favours those with a maximum smallest angle (near equilateral). Figure 10.1 shows a set of points (left) and two different ways of triangulating it, one which is non-Delaunay (middle) and one which is the Delaunay triangulation (right), unique in this case.

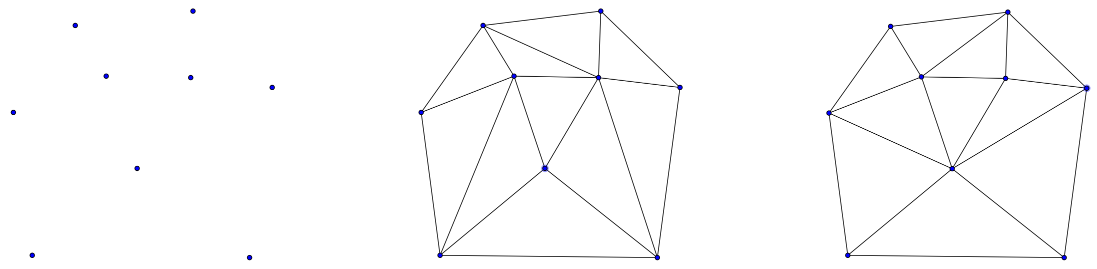


FIGURE 10.1: A point set with two triangulations, one non-Delaunay (middle) and one Delaunay (right)

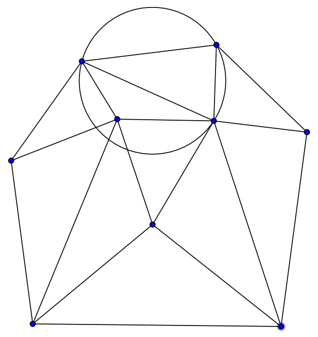


FIGURE 10.2: The Delaunay criterion is violated as there is a point lying within the circumcircle of one of the triangular faces.

Precisely speaking, a triangulation is Delaunay when no point in the set being triangulated lies within the *circumcircle* of any of the triangular faces. Figure 10.2 highlights one such violation.

This definition gives a way to construct a Delaunay triangulation, by trial-and-error. First, create any general triangulation of the point set — simply by connecting the points with non-intersecting line segments until it is complete. Next, consider each internal edge: it will necessarily have two adjacent triangles, one on each side. For each of these two triangles, take the circumcircle and check that it does not entirely contain the other triangle, or equivalently, that the unshared vertex of the other triangle is not inside the circumcircle. If both adjacent triangles satisfy this property, we say that the edge is *locally Delaunay*. Where this is not the case, the edge must be the diagonal of a quadrilateral: here, we can *flip* the shared edge and it will become locally Delaunay. This is illustrated in Figure 10.3.

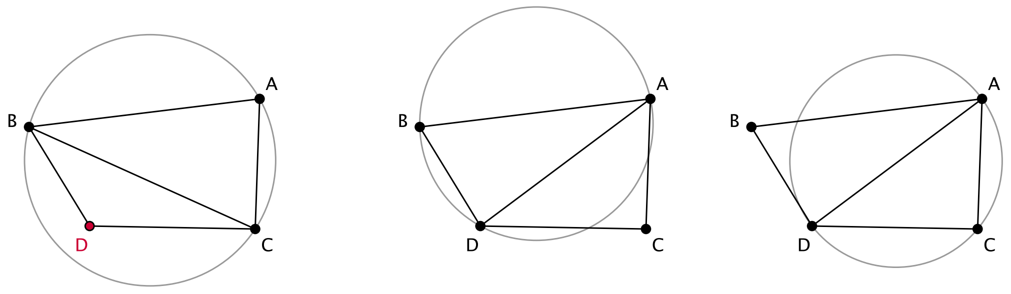


FIGURE 10.3: This diagram shows detail of the Delaunay violation (circumcircle of *ABC*, leftmost) and the resolution after the edge flip (circumcircles of *ABD* and *ADC*, rightmost)

The boundary case of the locally Delaunay test is when *four* points lie on the same circle: in this case either of the diagonals of the quadrilateral can be used to form the Delaunay triangulation. The Delaunay triangulation is therefore not guaranteed to be unique[4].

10.3.2 A Formal Definition in Isabelle

Let us now look at the formalisation of these concepts in Isabelle, building up from the concepts of point and signed area, used in Chapter 3, to define graphs, triangulations,

[4]It can be shown that the Delaunay triangulation *is* unique if no four points are concyclic.

and ultimately Delaunay triangulations. The computational geometry concept of a *planar straight-line graph* (PSLG) is particularly useful. This is a specialisation of graph theory where points (vertices) are situated in the plane with edges embedded as straight lines. *Planarity* is interpreted in the usual way, requiring that no two edges intersect (except when incident at endpoints). A *triangulation* is then any maximal PSLG, that is a planar straight line graph where no more edges can be drawn without violating planarity.

PSLGs and Triangulations

We will begin, for readability in the context of graph theory, by using Isabelle's "type synonym" mechanism so that we can refer to a `point` as a `vertex`:

```
type_synonym vertex = "point"
```

An `edge` can then be introduced as a new type defined as a set of two distinct vertices:

```
typedef edge = "{e::vertex set. ∃ a b. e = {a,b} ∧ a≠b}"
```

and we will take the usual interpretation of *intersect*, defined as follows:

```
definition straightEdgesIntersect :: "[edge, edge] ⇒ bool"
    where "straightEdgesIntersect ea eb ≡
        ∃ a1 a2 b1 b2. a1 ≠ a2 ∧ b1 ≠ b2 ∧
            ea = <a1,a2> ∧
            eb = <b1,b2> ∧
            ( ea = eb ∨
              b1 isBetween a1 a2 ∨
              b2 isBetween a1 a2 ∨
              a1 isBetween b1 b2 ∨
              a2 isBetween b1 b2 ∨
              ( leftTurn a1 a2 b1 ∧
                leftTurn a2 a1 b2 ∧
                leftTurn b1 b2 a2 ∧
                leftTurn b2 b1 a1 ) )"
```

The reader will note that we have re-used the `isBetween` and `leftTurn` predicates in this formulation. Whilst other definitions are possible (in particular, the existence of a point which lies on both edges simultaneously), this one avoids introducing new points and lends itself to the simplification rules we introduced in Chapter 6 — in both cases generally making it easier to work with. As with `point`, note that the introduction of the type `edge` causes Isabelle to create coercion functions `Abs_edge` and `Rep_edge`[5].

[5]For ease of use, we also introduced `<x,y>` as an abbreviation for `Abs_vertex(x,y)` and for `Abs_edge{x,y}`; however in practice this had limited applicability due to parsing delays. We did however, use this abbreviation in some places.

The new type `pslg` can then be formalised as:

```
typedef pslg  = "{pslg::(vertex set * edge set).
    ∃ (vs::vertex set) (es::edge set). pslg=(vs,es) ∧
        (∀ e1::edge. e1 ∈ es ⟶ (edgeIsInVertexSet e1 vs) ∧
           (∃ a b. e1 = <a,b> ⟶
              (∀ v :: vertex. v ∈ vs ⟶
                 ¬ v isBetween a b)) ∧
           (∀ e2::edge. e2 ∈ es ∧ e1 ≠ e2 ⟶
              ¬(straightEdgesIntersect e1 e2)))}"
```

where `edgeIsInVertexSet` is defined as:

```
definition edgeIsInVertexSet :: "[edge, vertex set] ⇒ bool"
    where "edgeIsInVertexSet e vs ≡ Rep_edge e ⊆ vs"
```

We are now ready to formalise the concept of a `triangulation`, simply as any maximal `pslg`:

```
definition isTriangulation :: "pslg ⇒ bool"
    where "isTriangulation g ≡
        ∀ a ∈ verticesOf g. ∀ b ∈ verticesOf g.
           a=b ∨
           <a,b> ∈ edgeSetOf g ∨
           (∃ e ∈ edgeSetOf g.
              straightEdgesIntersect <a,b> e)"
```

The functions `edgesOf` and `verticesOf` return the set of edges or vertices of a given graph respectively.

Circumcircles and the Delaunay Triangulation

As described earlier in this section, a triangulation of a graph is Delaunay if and only if no vertex is contained within the circumcircle of any triangular face in the graph. Equivalently, we can say a triangulation is Delaunay if and only if each internal edge *AB* satisfies a condition we will refer to as being *locally Delaunay*, meaning that the two triangles sharing that edge — call them *ABC* and *ABD* — each have circumcircles which do not contain the entire quadrilateral *ACBD*.

We will begin our formalisation by first defining a predicate `isMinimumTriInGraph` to tell us whether three given points form a triangular face contained in a graph:

```
definition isMinimumTriInGraph ::
        "[point, point, point, pslg] ⇒ bool"
    where "isMinimumTriInGraph a b c g ≡
        ¬(collinear a b c) ∧
        <a,b>∈(edgeSetOf g) ∧
        <a,c>∈(edgeSetOf g) ∧
        <b,c>∈(edgeSetOf g) ∧
        (∀x∈verticesOf g. x∉{a,b,c} ⟶ x outsideTri a b c)"
```

This refers to a predicate `outsideTri` with the obvious semantics, defined as being not `insideTri` and not `onBoundaryTri`, which in turn are defined in terms of `leftTurn` and `isBetween` respectively. The formal definitions are omitted for brevity.

The final element of scaffolding we require is the concept of a point being inside, outside, or on the boundary of the circumcircle of a triangle. Mathematically, for three non-collinear points $A = \langle a_x, a_y \rangle$, $B = \langle b_x, b_y \rangle$ and $C = \langle c_x, c_y \rangle$, the point $D = \langle d_x, d_y \rangle$ lies inside the circumcircle of triangle ABC if and only if the matrix determinant

$$\begin{vmatrix} a_x & a_y & a_x^2 + a_y^2 & 1 \\ b_x & b_y & b_x^2 + b_y^2 & 1 \\ c_x & c_y & c_x^2 + c_y^2 & 1 \\ d_x & d_y & d_x^2 + d_y^2 & 1 \end{vmatrix}$$

has the same sign as the signed area of the triangle; D is outside the circumcircle if the signs are opposite, and D is on the circumcircle if the determinant is zero. Thus we define the predicate `d inCircumcircle a b c` as returning true if and only if either `leftTurn a b c` and the determinant is positive or `leftTurn a c b` and the determinant is less than zero. The definition of `d outsideCircumcircle a b c` follows similarly but with the `leftTurn` conditions exchanged, and our definition of `d onCircumcircle a b c` simply asserts that the determinant evaluates to zero.[6]

We can now define `locallyDelaunay` as:

```
definition locallyDelaunay :: "[point, point, pslg] ⇒ bool"
    where "locallyDelaunay a b g ≡
        isTriangulation g ∧
        <a,b> ∈ edgeSetOf g ∧
        (∀ c ∈ verticesOf g. ∀ d ∈ verticesOf g.
            c≠d ∧
            isMinimumTriInGraph a b c g ∧
            isMinimumTriInGraph a b d g ) ⟶
                ¬d inCircumcircle a b c"
```

Finally, we define a Delaunay triangulation as a graph which is a triangulation whose every edge is locally Delaunay:

```
definition isDelaunayTriangulation :: "pslg ⇒ bool"
    where "isDelaunayTriangulation g ≡
        isTriangulation g ∧
        (∀ a b. <a,b> ∈ edgeSetOf g ⟶
            locallyDelaunay a b g)"
```

[6]The Isabelle formalisation is omitted due to its length. We further note a deviation from the terminology of Knuth [104], Guibas and Stolfi [71], and Dufourd and Bertot [45], all of whom refer to this as an `InCircle`. Our convention avoids any ambiguity with the common interpretation of "incircle" as the inscribed circle (the circle *inside* the triangle tangent to all three sides).

10.3.3 The Role of the Prover's Palette

With our new formal definition of `inCircumcircle` we set about proving some straightforward lemmas. The reason was two fold: we wanted to make sure our definitions behaved as expected and we thought many of the lemmas might be useful later in our proof endeavour. The Prover's Palette can be useful for performing a sanity check on the truth of our lemmas and it actually identified mistakes in several, including the lemma:

```
¬ d inCircumcircle a b c ⟹
        d onCircumcircle a b c ∨ d outsideCircumcircle a b c
```

QEPCAD told us that this statement was false, and it also produced the counterexample, $a = (0,1)$, $b = (-2,-1)$, $c = (-5,-4)$ and $d = (0,0)$. As shown in Figure 10.4, when we manually draw these points we notice that a, b and c are collinear [7]. Our `inCircumcircle` and `outsideCircumcircle` definitions are both false in this case, for when we unpack them we have neither `leftTurn a b c` nor `leftTurn a c b`. This matches our intuition, that *inside* and *outside* are not applicable in the degenerate case when the "circumcircle" is a line.

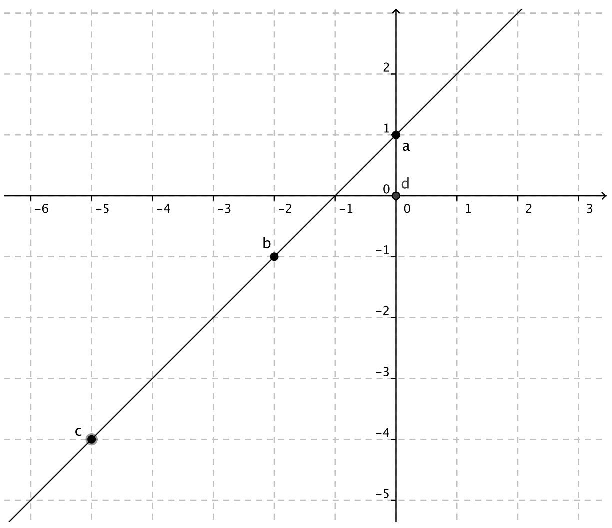


FIGURE 10.4: QEPCAD finds a counterexample for a circumcircle lemma corresponding to the degenerate collinear case.

However, it is natural to say that a point d is *on* the "circumcircle" in the collinear case when d is on the same line as a, b and c. Our definition of `onCircumcircle` matches this, simply requiring that the determinant is zero. Armed with a better understanding of the situation, we are satisfied with how we have defined the concepts, and it is merely necessary to amend the lemma:

[7] We used the GeoGebra environment to plot our counterexamples and found that a very useful tool: more is said about this in Section 11.2.3.

```
¬ collinear a b c ∧ ¬ d inCircumcircle a b c ⟹
        d onCircumcircle a b c ∨ d outsideCircumcircle a b c
```

Had we wished to, we could have excluded the collinear case from `onCircumcircle`; in this instance it makes no difference to our theory, but in other cases it might have, and it is good that we were forced to consider it explicitly early on in our theory development. Furthermore, by drawing this edge case to our attention, the Prover's Palette has been helpful in improving our intuition about the domain.

10.4 An Algorithm for Delaunay Triangulations

One algorithm for constructing a Delaunay triangulation has already been sketched, in Section 10.3.1: start with an arbitrary triangulation of a point set, and then iterate through the edges flipping those which are not locally Delaunay. This is a simple algorithm, but as Dufourd and Bertot point out, "proving its formal correctness is already a challenge" [45][8]. For this reason, we believe it is a prime candidate for evaluating the Prover's Palette.

10.4.1 Defining the Algorithm

Before we present our formalisation of the algorithm in Isabelle, let us introduce two elements of scaffolding that were required. The first of these gives us a way to iterate through the edges: a `graph` is defined in terms of an unordered *set* of edges, E, where the edge is represented as an unordered set of two distinct points; to more easily work with this structure, we introduce an ordered *list* of edges, EL, in which the edges are represented as a pair. We introduce the predicate `tupleListMatchesEdgeSet EL E` to assert that the ordered structure EL has precisely the same elements as the original unordered E.

The second element we introduced allows us to concisely formalize the edge-flipping operation. The function `replaceEdge <x,y> <a,b> G` returns the graph G with edge $\langle a,b\rangle$ replaced by $\langle x,y\rangle$ (with `replaceEdgeInSet <x,y> <a,b> E` defined similarly for a set rather than a graph).

Equipped with these concepts, we can express the algorithm in Hoare logic and assert its correctness in our main theorem. This is shown in Figure 10.5.

[8]Dufourd and Bertot's research and related work is described further in Section 11.1.1.

```
theorem DelaunayTriangulationAlgorithm: "

VARS (G::pslg) (V::point list) (E::edge set)
  (EL::(point * point) list) (a::point) (b::point)
  (i :: nat) (j :: nat)

{ distinct V ∧ isTriangulation G ∧                          (pre)
    Rep_pslg G = (set V, E) ∧
    tupleListMatchesEdgeSet EL E }

WHILE (0 < length EL)                                        (T1)
INV { (I1) }
DO
  a := fst(EL!0); b := snd(EL!0); EL := tl EL;
  IF locallyDelaunay a b G                                   (T2)
  THEN SKIP
  ELSE
    i := 0; j := 0;
    WHILE (i < length V)                                     (T3)
    INV { (I2) }
    DO
      IF (isMinimumTriInGraph a b (V!i) G)                   (T4)
      THEN
        WHILE (j < length V)                                 (T5)
        INV { (I3) }
        DO
          IF (i≠j ∧ isMinimumTriInGraph a b (V!j) G)         (T6)
          THEN
            IF ((V!j) inCircumcircle a b (V!i))              (T7)
            THEN
              E := replaceEdgeInSet <V!i,V!j> <a,b> E;
              G := replaceEdge <V!i,V!j> <a,b> G;
              EL := remdups [(a,V!i),(a,V!j),(b,V!i),
                   (b,V!j)] @ EL
            ELSE SKIP
            FI
          ELSE SKIP
          FI;
          j := j+1
        OD
      ELSE SKIP
      FI;
      i := i+1
    OD
  FI
OD

{ isDelaunayTriangulation G }"                              (post)
```

FIGURE 10.5: The Isabelle theorem which states the correctness of the algorithm for computing a Delaunay Triangulation: loop tests and if tests are labelled (T?) for reference and the loop invariants which are to be identified are labelled (I?).

The statement of correctness makes it explicit that the algorithm takes as input a set of distinct vertices and a general triangulation of them. We can see that the post condition asserts that the final graph G should be a Delaunay triangulation.

The reader will note that *three* loops are used for this algorithm. The outer loop recurses through the edges in the list EL. We note that for each edge ab there must be at most two minimum triangles which contain it. The next loop cycles through the points in V searching for the V_i which makes a minimum triangle with ab. Then, in the third (inner-most) loop we cycle through points in V again, this time looking to see if there is a V_j distinct from V_i which makes another minimum triangle with ab. There will only be a V_j if ab is an *internal* edge. If a V_i and V_j are found, we determine whether $\langle a,b\rangle$ is locally Delaunay, and if it is not the edge is flipped, or in other words `replaceEdge <V!i,V!j> <a,b> G`. With this flipped edge it may be necessary to revisit the four edges of the quadrilateral $a\ V_i\ b\ V_j$, so these edges are added to EL.

The components of the loop invariants are referenced as (I1), (I2), and (I3) in Figure 10.5. Discovering what these components should be is one of the main tasks in proving this theorem, and our work in this area will be presented next.

10.4.2 Proving the Algorithm

We now introduce some of the candidate loop invariants we have explored in proving the correctness of this algorithm. We will not give a full proof, but rather aim to give the reader enough understanding of the process — and of the specifics pertaining to the Delaunay triangulation algorithm used here — to enable an appreciation of the utility of the Prover's Palette.

Compared with Graham's Scan, the Delaunay algorithm in Figure 10.5 is not significantly longer or more complicated to understand. However, the formal proof does become more extensive: recall from Chapter 3 that each loop requires identifying an invariant and showing two verification conditions (truth of the invariant when entering the associated loop and preservation of the invariant at the end of each loop iteration). With three loops in the Delaunay algorithm, there are three invariants to identify and seven verification conditions to prove. The VCs are shown in Figure 10.6[9].

From our analysis of the Graham's Scan proof in Chapter 5, we remarked that formulating the correct loop invariant was a non-trivial task, requiring several iterations before the necessary information to prove the truth of the VCs could be established.

[9]The VCs are presented for completeness and for the interested reader, but a full understanding of them is not essential for appreciating the remainder of this chapter.

VC1	*Invariant at Loop 1 Start*
	$(\text{pre}) \Longrightarrow (\text{I1})$
VC2	*Invariant at Loop 2 Start (or Loop 1 End if Loop 2 skipped)*
	$\big((\text{I1}) \wedge (\text{T1})\big) \Longrightarrow \big((\text{T2}) \rightarrow (\text{I1}')\big) \wedge \big(\neg(\text{T2}) \rightarrow (\text{I2})\big)$
VC3	*Invariant at Loop 3 Start (or Loop 2 End if Loop 3 skipped)*
	$\big((\text{I2}) \wedge (\text{T3})\big) \Longrightarrow \big((\text{T4}) \rightarrow (\text{I3})\big) \wedge \big(\neg(\text{T4}) \rightarrow (\text{I2}')\big)$
VC4	*Invariant at Loop 3 End*
	$\big((\text{I3}) \wedge (\text{T5})\big) \Longrightarrow \big((\text{T6}) \rightarrow \big((\text{T7}) \rightarrow (\text{I3}')\big) \wedge \big(\neg(\text{T7}) \rightarrow (\text{I3})\big)\big) \wedge$ $\big(\neg(\text{T6}) \rightarrow (\text{I3})\big)$
VC5	*Invariant at Loop 2 End (after Loop 3)*
	$\big((\text{I3}) \wedge \neg(\text{T5})\big) \Longrightarrow (\text{I2})$
VC6	*Invariant at Loop 1 End (after Loop 2)*
	$\big((\text{I2}) \wedge \neg(\text{T3})\big) \Longrightarrow (\text{I1})$
VC7	*Program End*
	$\big((\text{I1}) \wedge \neg(\text{T1})\big) \Longrightarrow (\text{post})$

FIGURE 10.6: The set of verification conditions required to prove the correctness of the Delaunay Algorithm, where (I?$'$) is used to indicate invariant (I?) evaluated with the new values associated with the updated variables at the end of that loop.

This iterative process of repairing the invariant to prove one VC correct, could have the consequence that other VCs would change, sometimes becoming unprovable. This sensitive dependency between the VCs and the invariants, coupled with a large number of minor-but-necessary facts being carried around, can make it extremely hard to focus on the essence of a proof. It is against this backdrop that we found the Prover's Palette particularly useful.

10.4.3 Prover's Palette: Identifying Missing Components in the Loop Invariant

We will examine in depth one example of where the Prover's Palette helped to focus our intuition by revealing overlooked case splits in point configurations. This ultimately helped us identify a missing component in the loop invariant I3, and as a consequence this change filtered back through the proof indicating that I2 and I1 needed updating also.

In proving VC4, we had to show that invariant (I3), possibly updated, remained true at the end of every iteration of the third loop. Inside that loop, if V_j is in the circumcircle of a b and V_i, then we flip the edge ab to become the edge V_iV_j, and we must show that the invariant (I3) is true for the updated graph (the first conjunct in the conclusion of VC4); and if V_j is not in the circumcircle, no changes are made to the graph (the latter conjunct in VC4).

The latter case is straightforward, but the former one is quite involved. One of the lemmas we were faced with proving was that the new graph remains a triangulation; in particular, that no more edges can be added to the new graph without intersecting the edges in it. Since the new graph has removed the edge *ab*, the proof involves showing that there must now exist some edge in the graph which intersects it. Our intuition tells us that *ab* would intersect newly added V_iV_j, so we set forth to prove this. Along the way we encountered case splits, one of which was to prove:

```
⟦ ¬ straightEdgesIntersect <b,vj> <a,vi> ∧
  ¬ straightEdgesIntersect <a,vj> <b,vi> ∧
  vi ≠ vj ∧ ¬ collinear a b vi ∧ leftTurn a b vj
⟧ ⟹
  straightEdgesIntersect <vi,vj> <a,b>"
```

As this lemma could be expressed in purely algebraic form it was a perfect candidate for QEPCAD to solve. Using the Prover's Palette, we sent the lemma off[10], fully expecting QEPCAD to return the answer true and allowing us to progress with formulating the main structure of the proof. Surprisingly it returned false. For a greater appreciation of why this lemma didn't hold, we asked QEPCAD to provide a counterexample: it informed us that a violation could be found when $a = (0,0)$, $b = (1,0)$, $V_i = (-1,1/4)$ and $V_j = (-3,-1)$. This can be seen in Figure 10.7.

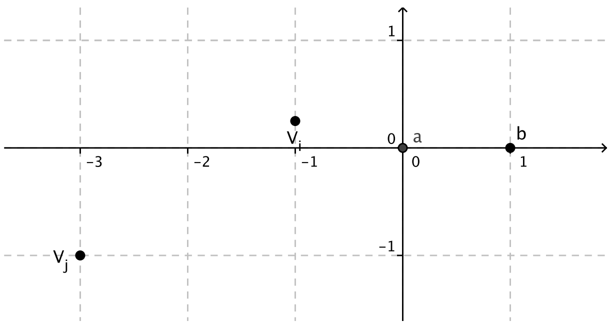


FIGURE 10.7: QEPCAD identifies a counterexample to the proposition that V_iV_j intersects *ab*.

By drawing out the counterexample QEPCAD provided, it was immediately apparent to us what information was missing in the lemma. From the context of the proof we knew that we were reasoning about the case where the edge *ab* was illegal and needed flipping, thus V_j should be lying within the circumcircle of the triangle abV_i. However, Figure 10.8 clearly shows that the assumptions of our lemma are not restricting V_j to this position. This was easily corrected by updating the lemma to include the assump-

[10]QEPCAD struggled to reason about this lemma until we manually performed geometric transformations on the points, translating $a = (0,0)$ and $b = (1,0)$.

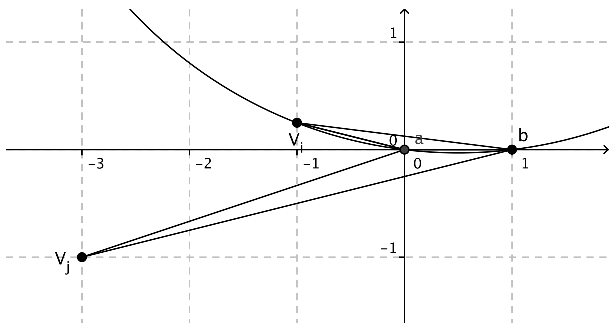


FIGURE 10.8: The counterexample QEPCAD finds can be eliminated by bringing in the fact that V_j is known to lie inside the circumcircle of $a\ b\ V_i$.

tion `vj inCircumcircle a b vi` (fortunately a fact already present in the subgoal of VC4 which uses this lemma).

However, sending the updated lemma to QEPCAD[11] still returned the answer false. The counterexample produced this time was $a = (0,0)$, $b = (1,0)$, $V_i = (-3,-2)$ and $V_j = (-35/16,-3/2)$, shown in Figure 10.9.

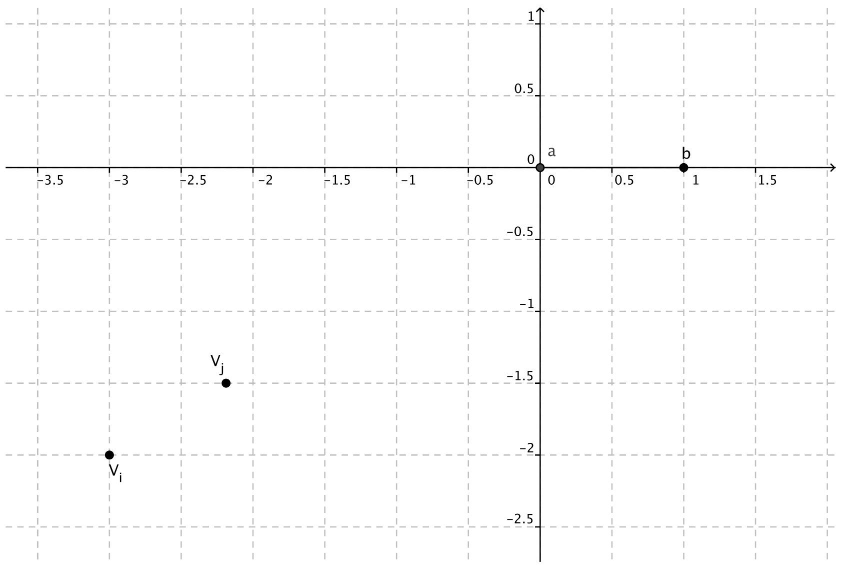


FIGURE 10.9: QEPCAD identifies a further counterexample even when the circumcircle constraint is in place.

We know by (T4) that `isMinimumTriInGraph a b V!i G` holds when we enter the loop. If a, b and V_i make a minimum triangle, then it is not possible for V_j to lie in its interior, and thus we have a contradiction, as shown in Figure 10.10. Unfortunately, we cannot merely pull `isMinimumTriInGraph a b V!i G` into (I3), because this will invalidate (I3$'$): in the case where ab is flipped, we no longer have the minimum triangle

[11]The updated lemma contained the fact `V!j inCircumcircle a b V!i` which meant we no longer needed the explicit assumptions that `leftTurn a b V!i` and `V!i` $\neq$`V!j`.

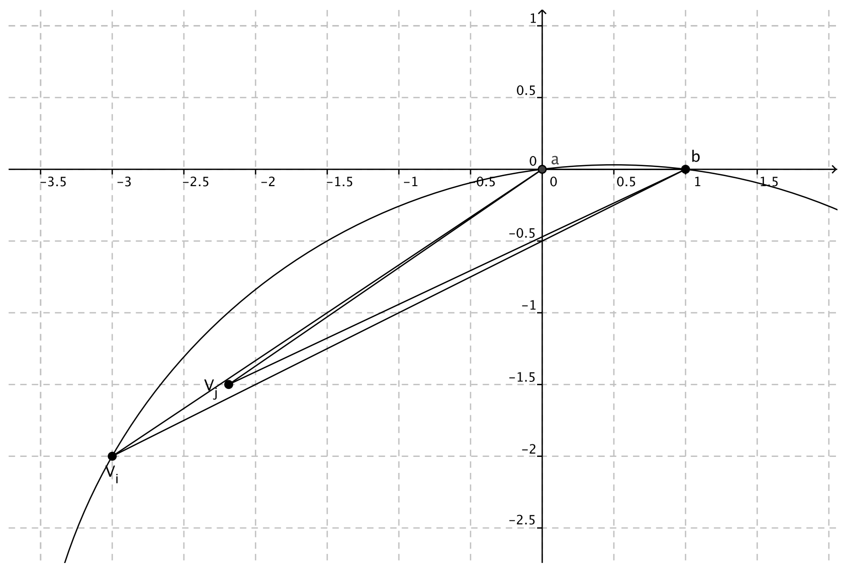


FIGURE 10.10: The counter-example QEPCAD provides tells us that we need the further fact that V_j cannot lie inside the $a\ b\ V_i$.

abV_i in G. One option we explored was to introduce an additional flag into the algorithm so that the invariant could assert conditionally that the `isMinimumTriInGraph` property holds before an edge flip, but this poses the additional non-trivial effort to show that there is at most one j which could trigger an edge flip for any i.

A simpler possibility, revealed by our explorations in the Prover's Palette QEPCAD widget, is to claim the following as an invariant:

```
∀x∈verticesOf g. x∉{a,b,V!i} ⟶ x outsideTri a b V!i
```

It is easy to show that this is implied by the `isMinimumTriInGraph` precondition, but this weaker form is independent of G and so can safely be added to (I3). Returning to the original lemma, some of our previous assumptions are now redundant, so we remove them, and this time, using Prover's Palette, we are able to have QEPCAD confirm that the patched lemma is true:

```
⟦ ¬ straightEdgesIntersect <b,vj> <a,vi> ∧
  ¬ straightEdgesIntersect <a,vj> <b,vi> ∧
  vj inCircumcircle a b vi ∧
  vj outsideTri a b vi
⟧ ⟹
  straightEdgesIntersect <vi,vj> <a,b>"
```

10.5 Evaluation

At this stage we believed that we had collected sufficient examples to show that the Prover's Palette system does indeed provide a powerful way of integrating tools. We had a good understanding of how it improves the user experience of constructing mechanical verifications and of its limitations. For this reason we decided that — whilst it would be a useful result in its own right — completing the proof of the Delaunay algorithm would not shed more light on our evaluation exercise. Our future plans include completing this proof and we estimate that it will take no longer than a month of work. A significant portion of the proof has been developed and the loop invariants appear to be converging to a stable set.

We estimate that QEPCAD has been invoked 50 times to date during the proof construction. As we described in this chapter, we found it extremely helpful in a number of ways: to prove broken geometric proofs from the Graham's Scan development; to check definitions; to sanity check that we were trying to prove a valid theorem; and sometimes in the situations where we were going down a wrong path we used it as a guide providing us with invaluable counterexamples, which in turn helped us in formulating the correct loop invariants. We believe that each of these ways address many of the main unnecessary difficulties we previously encountered when undertaking a large-scale formal proof. Recall that these difficulties were called out in Section 5.1. Figure 10.11 recalls this list of impediments to productivity and shows how our new approach to integrating systems is able to assist with each.

In Chapter 5 we called out two categories of difficulties with formal proof: those which contributed to the "black holes of time" and those which obscured our intuition. We found that the Prover's Palette greatly alleviated the first category of difficulties by reducing the time spent on reasoning about small details and time spent searching for applicable lemmas in the library. Of course, our integrations only helped with these issues in some areas; there is much more that can be done, and we will describe these shortly. We note however that these improvements generally required using QEPCAD in "oracle" mode, thus trusting its results and diminishing confidence in the resulting proof.

For the second category of difficulties, where intuition is obscured during the formal proof process, our approach is able to assist with all the noted difficulties *without* introducing any proof dependency on the external systems. Frequently in our Delaunay case study, the Prover's Palette was used as a guide, either to sanity check new formulations or alert us to mis-steps along the way. Having explicit counterexamples was particularly useful to penetrate the sometimes densely obfuscated expressions, and being able to change bound- and free- variables and isolate expressions — using the tools interactively — further enriched our understanding of proof goals. By eliminating

Difficulties Using Isabelle	**How the Prover's Palette Helps**
Proving Minute Details	oracle mode discharges many algebraic goals; subgoal mode sometimes introduces `simp`lifiable goals
Library Look-up	oracle mode reduces the need for looking up lemmas; subgoal mode sometimes introduces `simp`lifiable goals
Entering Correct Instantiations	QEPCAD provides witnesses; QEPCAD can identify superfluous assumptions, facilitating automatic instantiation
Refactoring	sanity checks permit faster iteration, reducing need for refactoring
Version Incompatibilities	oracle mode gives assurances for proofs which break across version changes
Intuition Obscured by Opaque Presentation	QEPCAD can identify superfluous assumptions, clarifying relevant lemmas
Distracting Minutiae	QEPCAD allows skipping or deferring intricate proofs
Expensive Mistakes caused by Obscured Intuition	mistaken lemmas and definitions identified sooner using QEPCAD

FIGURE 10.11: Summary of the ways the Prover's Palette addresses the obstructions to interactive formal proof listed in Figure 6.1

unnecessary assumptions, we could simplify many of the expressions more quickly; however, we did not utilise this feature of the Prover's Palette as much as may have been useful as the interactive selecting/deselecting of assumptions was somewhat tedious. In all these ways, having tightly integrated access to external systems improved our ability to concentrate our intuition on the essential proof argument, accelerating development without compromising confidence.

Despite the improvements the Prover's Palette brought to the proof process, there are a number areas where the tool integrations could perform better. Dealing with formulae involving division was frequently cumbersome, and adjusting definitions for usability, due to the simplistic treatment of internal predicates (expanding their definitions), is both artificial and cumbersome. One of the most frustrating situations was when QEPCAD could not produce an answer within reasonable time or space bounds. In these situations, as with division, manual manipulation of the goal state is needed to see whether the goal complexity can be reduced, again rendering the integration less smooth than we would like. Finally, there is the challenge of reconciling expanded definitions with the original statement, as human-readable semantics vanish when predicate expressions are replaced by algebra and variable names are changed in unintuitive ways. Some ideas for tackling these limitations will be reviewed in the next chapter, when we look at potential future work in Section 11.2.

One of the areas we found most exciting is the automation possible around the time-consuming task of formulating the correct loop invariants. Taking as an example the "missing invariant component" from the Delaunay proof (described in Section 10.4.3), we have shown how our QEPCAD integration can help determine what is logically missing; however, a more subtle understanding of the flow of the proof is needed to identify what facts from the environment can be pulled in to correct an incomplete loop invariant. Addressing the limitations described in the previous paragraph may make this easier, but already, with the Prover's Palette we were able to take advantage of the potential of tools to remove much of the tedium. More importantly, by relieving the human of much of the cognitive burden and distraction, our system allowed us to focus on the structure of the argument, seeing the big picture, and figuring out and expressing the more interesting steps of the proof.

Chapter 11

Conclusions

Our thesis began by describing the role intuition plays in formulating a mathematical proof; whilst one has to be mindful of its fallibility, it would be wrong to underestimate its importance (Chapter 1). We then drew a comparison between current mathematical software packages, observing that, in general, mathematicians have happily embraced CASs for exploring problem spaces efficiently, but they have been hesitant to adopt theorem provers, despite the guarantee that proofs constructed in them are formally correct (Chapter 2). Our case study of the Graham's Scan algorithm in the theorem prover Isabelle (Chapter 4) highlighted that this disparity could be attributed to the fact that theorem provers can inhibit intuition (Chapter 5). It was our belief that a happy marriage could be achieved between the two types of systems, culminating in a more powerful proving environment (Chapter 6). Hypothesising that it would be *possible* to make CASs readily accessible within the same IDE as a theorem prover, we set about creating the Prover's Palette, a system inspired by the proof engineering paradigm which enables integrations of mathematical tools within the same IDE (Chapter 7). The concrete integration of Isabelle and QEPCAD within the Prover's Palette was then presented (Chapter 8). Furthermore, we demonstrated that the design of our system was modular and extensible by showing how Maple's plotting functionality could be accessed whilst constructing a formal proof in Isabelle (Chapter 9). Finally, we revealed how the Prover's Palette could be of assistance during the course of complex proofs, focussing on the formal verification of another fundamental algorithm from computational geometry, Delaunay triangulation (Chapter 10).

This chapter aims to make clear the contributions of our research and put them into context within the larger picture of mechanical theorem proving. Our belief that intuition is paramount when constructing intricate formal proofs is re-iterated at the end.

11.1 Contributions of this Work

This thesis is the culmination of many years of research, and we are gratified to list the main ways we feel it contributes to the field of mechanical theorem proving:

- formal proofs for algorithms in computational geometry;
- a new geometry library for Isabelle;
- a new planar graph library for Isabelle;
- identified inaccuracies in two written proofs for the Graham's Scan algorithm;
- the conceptual idea of integrating mathematical systems in a user-centric way;
- the implementation of a framework following this approach in a popular IDE;
- an integration of QEPCAD with Isabelle in this framework;
- an integration of Maple with Isabelle in this framework;
- case studies highlighting the value of this approach and framework;
- support for the important emerging idea of proof engineering;
- suggesting improvements to QEPCAD (witnesses and bug-fixes, now available).

The following subsections elaborate on these points, highlighting the value they bring in a wider context.

11.1.1 Proofs in Computational Geometry

A prominent standalone contribution of our efforts is the formalisation of key concepts from computational geometry: we constructed the first formal proof of the Graham's Scan algorithm for computing convex hulls and made substantial progress towards a formalisation of a Delaunay triangulation algorithm. Furthermore, our formal verification of Graham's Scan revealed flaws in the written arguments justifying the correctness of the algorithm in two mainstream textbooks, thus highlighting the value of a mechanised proof.

When we embarked on our research, there was only one article in the area of formalising geometric algorithms: Pichardie & Bertot had proven two algorithms for finding planar convex hulls, but restricted their domain to non-collinear cases [143] (as discussed in Section 5.2.2). Their work was encouraging and impressive, and it inspired us to aim for a mechanical proof which was not so constrained [120]. Whereas Pichardie & Bertot adopted Knuth's axiomatic approach, assuming that no three points are collinear (and asserting only informally that this could be repaired either by new axioms or by a perturbation argument), we opted to build upon a different foundation, involving coordinate geometry in the real plane. This made our proof endeavour more difficult initially, but it provided a sound foundation with the benefits of addressing collinear points and bringing extensive automation once the foundation was laid.

While we were in the midst of our second case study, proving the Delaunay algorithm, Dufourd & Bertot announced a formal proof for the construction of such triangulations [45]. Their work, based on intuitionistic type theory, takes a very mathematical view of the problem and their whole approach is quite remote from computer science. They build on a theory of hypermaps, a powerful and useful concept but not one typically used in software engineering, and they start with a mathematical equivalence several steps removed from the code that a computer might execute. Their work has an elegance which ours sorely lacks, but our work addresses a different problem: it shows that existing tools *can* tackle mainstream algorithms in computational geometry, it shows *how* this can be done, and it shows *improvements* to the process enabled by our focus on tooling.

11.1.2 Theories in Isabelle

In the course of proving the geometric algorithms we have naturally built up a large body of formalised mathematics in Isabelle. A theory of planar geometry has been developed, expressed in terms of signed areas of triangles, and proving a wide range of useful and common lemmas. Beyond this, we provide definitions and results pertaining to circumcircles, graphs, convex hulls, triangulations and Delaunay triangulations. These are necessary not only for our own work, but for any work in Isabelle addressing mainstream computational geometry themes. The automation we have supplied in the form of carefully selected additions to Isabelle's simplifier is a further important aspect of the theories we have developed.

11.1.3 The Prover's Palette

The Prover's Palette as an out-of-the-box ready-to-use system is another contribution of our research, allowing Isabelle users to easily call upon the evaluation functionality of QEPCAD and the graphing capabilities of Maple whilst working on their formal proofs. This system, as we have shown, enables versatile tool integrations, where the user can customise how the external tool is used and how the result is applied in the theorem prover, without disruption to the user's process. This approach — whereby an external tool is effectively invisible except when useful, or when manual control is desired — has been a major assistance to our efforts, and we are confident that it will be beneficial in the many other mathematical domains where QEPCAD and Maple can be used.

The extensible, open-source design also makes it easy for other developers to supply integrations to additional systems. By lifting much of the functionality to abstract framework classes, we have made it so that comparatively few additions are required to onboard new external CASs or solvers, or other theorem provers. We have made it easy to see where the additional code for new integrations would be needed. This will, we hope, allow the Prover's Palette to become a platform for others to share integrations to yet more tools and new techniques.

Both the approach of the Prover's Palette and the system itself have been validated by our case studies. The previous three chapters have given numerous concrete instances where it has been useful. Specifically, the Prover's Palette has been of assistance in:

- discovering missing loop invariants;
- facilitating the exploration of our problem domains;
- proving algebraic statements;
- indicating if lemmas were unprovable (and in some occasions providing counterexamples);
- helping to identify the minimum set of assumptions which entail a conclusion.

As a result, less time is spent proving minute and distracting details, looking up the library and finding the right instantiations for lemma applications. Mistaken definitions and incomplete assumptions are flagged much sooner. This was useful in many situations we encountered, from the initial modelling of a theory through to the construction of proofs therein; most helpfully for us, it kept us on track during the hunt for loop invariants. This improvement to productivity is beneficial on its own, but it also reduces the frequency that one's intuition is disrupted. With intuition so improved, and further improved by the exploration capabilities, one enjoys a more natural formal proof process and a clearer understanding of the formalisation task: one is closer both to the underlying mathematical truth and to its celebrated beauty.

11.1.4 Proof Engineering and User-Centric Provers

It is manifest that the many mathematical software packages, and the myriad ways they are used, have evolved for good reasons. But the only way to make these accessible in their variety and their fullness of functionality, within a theorem proving context, is to design the integration from the outset for interactive use. It must be done in such a way that the user need not be an expert in each external tool, nor must he interrupt the flow of his activity — with cumbersome translations and context switches — to use it. There has been a large body of prior work integrating systems (Section 6.3), but none fully address the problem of enabling full-featured, low-overhead interactive use.

Our biggest contribution, we feel, is the approach we have taken, showing how this problem can be resolved. We were powerfully influenced by developments in software engineering, and by the creative idea of proof engineering, as described in Section 6.4. This doctrine introduces a markedly different emphasis to prior work in theorem prover design, placing the user at the centre of the process and focussing on the richness of the development environment where they operate. In doing this, systems such as Eclipse Proof General opened an effective and promising new avenue for theorem proving, down which we were tempted as we composed our framework for integrating systems, with compelling results. The usability, utility, and acceleration enabled by our approach make a strong case for this style of combining tools, keeping the cognitive interruption low and letting the user focus his intuition on the essential aspects of the proof. In achieving these benefits, we are indebted to the idea of proof engineering, and it is rewarding to see them provide a ringing endorsement for this burgeoning paradigm.

11.1.5 Reflections

Reflecting on our research methodology, we acknowledge that several lessons can be learned from our experience. We share them here as we feel that they may be of interest and help to others in the community.

Without doubt, we deliberated over constructing the formal proof of Graham's Scan for far too long. Early into the proof endeavour we recognised the difficulties we were encountering and appreciated which improvements could and should be made to the interaction process with Isabelle. However, instead of interrupting the proof and turning our attention to enhancing the tool, we trudged onwards with the – at times painful — verification. With hindsight, proving and improving should be given equal importance and ideally the two should be interleaved.

Future proof developments will also benefit from writing more in depth comments in the theory files. We learnt the hard way that complex proofs which involve many case splits can be difficult to write about when one has not looked at the proof for a considerable amount of time. This is particularly true when the proof has been constructed using Isabelle's procedural style.

Another lesson has been extrapolated from comments several researchers have made at conferences, enquiring as to why we integrated Maple and not Mathematica into the Prover's Palette. Our answer was simply that it was the software package we had the most experience with, so it seemed like a natural choice. In retrospect it would be wise to seek feedback early on from the community as to which mathematical tools they use most frequently and use this to influence future integrations.

Finally, we would like to add that we wish we had been more vocal in promoting the Eclipse Proof General project when it was in its infancy. By championing it, we may have helped to bring more users and developers to the system and it may have been actively supported today. We have learnt the importance of speaking out at conferences, writing to mailing lists and being active in advocating the use of a system.

11.2 The Bigger Picture

Let us now turn our attention to where our research fits in to the wider picture of mechanical theorem proving, and in particular identify some of the most exciting avenues of future research which we believe hold promise for the field.

11.2.1 Program Verification

Returning to the concrete proofs we have undertaken in computational geometry, one obvious continuation is to expand the coverage of algorithms in this field, beginning of course by completing our work on the Delaunay triangulation algorithm (as discussed in Section 10.3), and proceeding to look at Voronoi diagrams, more efficient algorithms, and at some of the applications where they are used. Space exploration, for example, is one domain where geometric reasoning is widespread and the strong guarantee of formal correctness is highly valued. Crucial to the verification of such algorithms would be the modelling of any advanced data-structures used in computational geometry, as well as more low level constructs such trees and pointers.

While correctness at the algorithmic level is required if we are to provide guarantees for a live system, it is not sufficient. It would be necessary — and interesting — to look at verifying program code for these algorithms (*e.g.* in C or Java) and implementations running on specific hardware (*e.g.* ARM or Intel). This is a very active area of research, from the hardware level [79, 101] to specific programming contexts, either using custom proof languages [189] or geared around well-known languages

[110, 54, 16]. Logically connecting our algorithmic proofs to this chain would afford an unprecedented level of confidence in the results.

Whether working with algorithms or executable code, the challenge of formulating the right loop invariant remains when using Hoare logic. As we have described (Section 5.1), this is often a taxing and time-consuming part of the process. In recent years there has been lots of attempts at automation in this area, and an entire workshop on invariant generation (WING) has been devoted to this challenge [187]. Whilst we do not think there is a general solution to this problem, there are instances where it is possible to automatically discover the loop invariants; more interestingly, it might be possible to make some of the automation in this area more amenable to human guidance to give the best of both worlds.

11.2.2 Extending the Functionality of Existing Tool Integrations

Continuing our investigations as described above will give more insight into how the Prover's Palette tool integration could be made more useful, but already — through our case studies, and through conferences, workshops and countless discussions — we have identified a number of compelling enhancements. We will start, in this section, with those potential improvements relating to the tools currently integrated within the Prover's Palette.

Supporting Division in QEPCAD

One of the most frustrating difficulties we encountered with the existing tool integrations is the inability of QEPCAD to reason about division (Section 10.2). At present, any proof goal containing division must be manually rewritten in terms of multiplication in order to be sent to QEPCAD. Isabelle's *field_simps* help in simple situations, but usually require explicit non-degeneracy preconditions or case splits, meaning that the user must enter a tedious sequence of simplifications and rule applications.

To help with this problem, we designed a simplification set for removing division in Isabelle, shown in Figure 11.1. This set addresses many more of the formulae we encountered. However, it is not complete, and it fails for some of our more intricate statements involving division (such as most of those where sums of fractions are present), so more work to devise a reliable procedure for removing division would be a valuable undertaking.

```
div_simp_eq_l:    (a/b = (c::real)) =
  ((b≠0 ⟶ a=c·b) ∧ (b=0 ⟶ c=0))
div_simp_eq_r:    ((c::real) = a/b) =
  ((b≠0 ⟶ c·b=a) ∧ (b = 0 ⟶ c = 0))
div_simp_neq_l:   (a/b ≠ (c::real)) =
  ((b≠0 ⟶ a ≠ c·b) ∧ (b=0 ⟶ c≠0))
div_simp_neq_r:   ((c::real) ≠ a/b) =
  ((b≠0 ⟶ c·b ≠ a) ∧ (b=0 ⟶ c≠0))
div_simp_less_l:  (a/b < (c::real)) =
  ((b>0 ⟶ a < c·b) ∧ (b<0 ⟶ a > c·b) ∧ (b=0 ⟶ 0<c))
div_simp_less_r:  ((c::real) < a/b) =
  ((b>0 ⟶ c·b < a) ∧ (b<0 ⟶ c·b > a) ∧ (b=0 ⟶ c<0))
div_simp_gr_l:    (a/b > (c::real)) =
  ((b>0 ⟶ a > c·b) ∧ (b<0 ⟶ a < c·b) ∧ (b=0 ⟶ 0>c))
div_simp_gr_r:    ((c::real) > a/b) =
  ((b>0 ⟶ c·b > a) ∧ (b<0 ⟶ c·b < a) ∧ (b=0 ⟶ c>0))
```

FIGURE 11.1: A set of Isabelle simp rules for removing division

Supporting Transcendental Functions in QEPCAD

In addition to supporting formulae which involve division, there is ample scope to increase the coverage of QEPCAD for other functions and domains. Statements involving absolute value and rational powers could be made amenable to QEPCAD with a small amount of automated pre-processing.

Furthermore, a technique for reasoning about transcendental functions (*e.g.* sin, log, e^x) in QEPCAD has recently been demonstrated by Akbarpour *et al.*, replacing occurrences of such functions with polynomial upper and lower bounds [1]. Combining their "bounds" approach with our work could provide an extremely powerful integration, resolving some of the speed and coverage issues with their system on its own, as well as some crucial accuracy issues: in preliminary experiments using our QEPCAD integration with their bounds, the Prover's Palette has highlighted that some of their reported bounds are in fact too weak. We presume that including more terms in the Taylor expansion would correct this, and this could be an interesting future investigation.

Reducing the Search Space: Translation Invariance

One other area where our QEPCAD integration is limited, as we discovered, is the frequency with which problems which are *theoretically* solvable could not be solved within reasonable time/space bounds. There are some concrete improvements which can be done to alleviate this. The first of these is to apply more techniques for reducing the number of variables and hence reducing the complexity of problems sent to QEPCAD. For geometric problems, there are some well known techniques which we touched upon in Section 6.3. We introduced lemmas showing translation invariance allowing us to fix one point as the origin, removing two variables from the problem sent to QEPCAD and in several cases making the problem tractable. Our lemmas, however, are not always easy to apply, and we do not begin to tackle other invariants such as scaling and rotation which could provide a way to eliminate further variables. A formal theory of geometric invariance has been developed [112] and it would be rewarding to build on this, instead of our simple lemmas, so that we can simultaneously extend and simplify this way of reducing problem complexity.

Reducing the Search Space: QEPCAD's Special Quantifiers

Another concrete improvement would be to utilise QEPCAD's special quantifiers, `F` ("for infinitely many") and `G` ("for all but finitely many"), as weakened alternatives to `E` ("for at least one", or "there exists") and `A` ("for all"). This enables QEPCAD to bypass a large amount of computation, "and that can make a huge difference!" [151]. Knowing when to use these special operators is more difficult. The interactive nature of the Prover's Palette allows a sophisticated user to manually select these when appropriate, which is a good start, but we can go further. Using Isabelle's classes of `open` and `closed` sets, it could be possible to detect and formally prove the sufficiency of a weaker quantifier (such as "for all but finitely many" in place of "for all"), and of course to send the weakened form for proof in QEPCAD. A good concrete example is given on the QEPCAD website [151], taking a problem which Hong *et al.* believed to be intractable in QEPCAD. The example shows how it can be solved (in two seconds!) using special quantifiers, justifying the use of these quantifiers with the following generalised argument: if the space of solutions can be shown to be closed (due to the inequalities used, in the case of the problem he is discussing), then the complement of this set, the counter-examples, is an open set; therefore, if there is one counter-example, there must be an infinite number of counter-examples, or in other words the original statement in terms of "for all" is equivalent to the statement quantified instead by "for all but finitely many". The QEPCAD site concludes:

> The moral of the story is this: "for all but finitely many" can be decided using CAD faster than "for all" (which makes sense, since finitely many points can be ignored), so you should use it if you can.

Such an approach could have widespread applicability, using the strength of the theorem prover to express topological arguments to guide the search space employed by many algebraic decision procedures.[1]

Further Automation of QEPCAD

In addition to the increased breadth of coverage in QEPCAD, there are a number of configuration options and modes of using the tool we would like to better support. As mentioned in Section 8.2.1, it is sometimes necessary to increase the amount of memory available to the tool; while this can be done manually by the user, it would be a simple and useful extension to automate this process. The same is possible for other configuration options also, such as the type of projection used and normalisation assumptions, although these may require more sophisticated automation.

A second area which has been mentioned (Section 8.3.2) is that of finding a minimal set of assumptions needed for a statement to be true. We manually did this by selecting various sets of assumptions interactively, to find which assumptions could be thinned out (removed) to make the subgoal easier to comprehend (and in some cases filtered out the conclusion, as we realised truth was due to a contradiction in the assumptions), but as we described, this was somewhat cumbersome. Automating this process — both the discovery of unnecessary components, and the removal of these components — would be an effective way to use the computational power of QEPCAD, in a fully formal way, to simplify the proof state, making it easier for our intuition to grasp.

Expanding Coverage of Functionality in Maple

Finally let us briefly discuss the integration with Maple. This exercise was done primarily to demonstrate modularity and extensibility in the Prover's Palette, picking two of the ways we were familiar with the tool — plotting equations and solving boolean statements — and complementary to our use of QEPCAD. Maple, of course, has a vast library of techniques and is routinely used in far more ways than we have catered for in our GUI. One can manually enter any command, cutting-and-pasting relevant proof terms from the automatic translation to Maple's syntax, but a tighter integration covering more of its functionality would have many benefits.

Simplifying and factorising are an obvious strength of Maple which could be added

[1]As another example, such a theory could potentially be used to formalise Pichardie & Bertot's suggestion [143] to use a perturbation argument to make their convex hull algorithm proof applicable to points in general position, including collinearity.

to the Prover's Palette. As described in Section 9.2.3, other integrations have concentrated in this area. It would be attractive to build on this work, leveraging the highly interactive nature of our approach, to make this capability available, easily and on a fine-grained basis. A GUI module which facilitates the selection of sub-components would be an important addition, not just here, but in many areas where we see that being useful, from reducing algebra in QEPCAD to plotting equations.

Another simple extension which could be applicable to many cases, including Akbarpour *et al.*'s work already mentioned, would be to generate power-series expansions of transcendental functions using Maple. Not only would this facilitate automatic computation of candidate bounds to be sent to QEPCAD, it could also allow the user to control the number of terms desired in the expansion from Maple, and thereby control the precision of the estimate (trading off against the desire to keep the algebra simple and possibly preventing errors such as the one we noted in their work).

These extensions would continue the trend of expanding the set of the functionality exposed by the Prover's Palette from Maple. This puts a much richer set of automation at the proof author's fingertips, accelerating the proof process and leaving it to the author's discretion to judge both how the tool is used and how much the tool is trusted. However rather than base what new functionality would be most useful on our own limited experience, it would be helpful to investigate how mathematicians and computer scientists use the tool at present, and, as described in the next section, to look at related systems so that the Prover's Palette can implement category extensions of functionality across multiple tools.

11.2.3 Integrating More Systems

A natural direction to take our work would be to integrate more systems into the Prover's Palette. One of the most popular tools used by mathematicians is Mathematica [113], and so it would be a great addition to the suite of tools currently supported. Indeed there are many other CASs which could be beneficial to integrate, even if some of their functionalities have a lot of overlap with other tools which have been added. Our approach in the Prover's Palette architecture for using abstract classes to provide common functionality and common GUI affordances could be extended to introduce a library of widgets applicable to many of the CAS functionalities. The user could then select their favourite external tool for a task such as factorising algebra; tools vary in their support for different tasks, of course, so the user might make this selection based on how well tools perform particular tasks, or instead choose to let several run automatically to make use of the fastest or best result possible.

We believe there is scope to incorporate many other categories of tools into the Prover's Palette, such as model checkers, discrete algebra and group theory systems and statistics packages. We would like to investigate how mathematicians and other users of these tools currently employ them.

GeoGebra [64] is another system we are eager to make available within the Prover's Palette. We found ourselves frequently using it over the course of our research to graphically explore the counterexamples QEPCAD provided when we were attempting to prove false conjectures, and in fact many of the diagrams we have included in this thesis were produced using the tool. Currently, one has to interact with GeoGebra separately and explicitly tell it what sort of object to draw and any constraints which it should adhere to. It would be useful to have this process automated for some problems, especially those where QEPCAD provides a counterexample which could be given a geometric interpretation. This would be a non-trivial process as semantics would need to be attached to the variables, *e.g.* allowing the Prover's Palette to infer that two variables make up the co-ordinates of a point, and that the abstract sequence of characters `inCircumcircle` refers to a circle in GeoGebra. We will discuss this more in Section 11.2.4.

Extending the Prover's Palette to work with more theorem provers is also a future goal. One natural extension would be to link more interactive theorem provers with the PG Kit — an ambition of that project — then incorporate these into the Prover's Palette. Of course, this task would be made much easier if theorem provers were to communicate in a standard form, such as OpenMath [137] or OMSCS [29]. Alternatively, if the PGIP standard is extended to give richer structural details of proof states in a standard form, then a PGIP-compatible prover would be straight-forward to integrate with the tools supported in the Prover's Palette. Prover-specific customisation would only be needed for the optional (though useful) preprocessing and result application steps.

Whilst supporting different interactive theorem provers each talking to multiple mathematical tools is useful, a loftier goal would be to design the Prover's Palette to permit the communication between multiple interactive theorem provers and other tools in unison. We note that the FlySpeck project ignited with many researchers carrying out their designated proof on their chosen theorem prover (be it Isabelle, Coq or HOL-Light) with the goal to then integrate the parts of the proof into a coherent whole. This approach could be adopted in the Prover's Palette — using the results from individual theorem provers in one argument. However, we note that it may not be very satisfactory to do this in general, especially if different foundational structures have been used to define objects. Translating results from one prover to another is a complex project, but would be an interesting direction to take the Prover's Palette.

Of course, linking in more provers to the Palette does not need to involve exclusively those of the interactive variety. Incorporating first-order, automated provers would be tremendously beneficial. As mentioned in the Chapters 3 and 10, Isabelle's Sledgehammer tool already does a wonderful job at utilising such external provers but it does not always succeed at finding a solution. In these cases it may be possible for a user to guide an external, automated prover to find a solution (perhaps by selecting fewer lemmas for it to use when searching for a proof). Experimenting with this idea could yield some interesting results.

Once we begin to expand the suite of tools in the Prover's Palette we will undoubtedly have to consider the complexity of the cooperations. A number of projects, including KOMET [29], PROSPER [43], Logic Broker [3], and PG Kit, could assist in such multi-system integrations: the Prover's Palette could offer a "meta tool" widget, recommending systems to use at crucial points in the proof development (and, where possible, automatically opening the corresponding widget). This converges to one of the core ideas of software engineering and proof engineering, that one should be pragmatic about getting the job done, making use the best tool for the job, with the community as a whole concentrating on tools working well together. The next section will look more broadly at how proof engineering suggests exciting possible continuations of this research.

11.2.4 More Proof Engineering

One area of future work raised in the previous section is that of customizing the Prover's Palette's behaviour for specific theories, *e.g.* telling an external tool like GeoGebra how to interpret `inCircumcircle` (previous section), or supporting translation of transcendental functions to Maple (Section 9.1.4). To do this neatly requires new machinery on top of the current architecture, for presently the obvious implementation would consist of a new Eclipse plug-in extending the Prover's Palette. This is the wrong place for theory-specific metadata to sit, as it limits the modularity and general applicability of the tool. What is needed is a way to provide usage hints, for other systems and potentially for human users, as a natural part of the theory itself.

One technique for doing this has been pioneered in the FeaSch system [84]. FeaSch provides a mechanism for encoding within a theory the control logic for detecting relevant "features" (semantic tags), and the application logic for activating lemmas associatively linked to these features. FeaSch is written atop Eclipse Proof General, so one quick benefit is to use theory-specific features to suggest which tools are most relevant. Leveraging a theory metadata mechanism such as this could also provide a clean way for theories to extend the Prover's Palette integrations. This would be very powerful: any new theory could include domain-specific enrichments making external tools more relevant.

Our hypothesis in Chapter 7 is that external tools can be integrated in ways relevant to their context and accessible to the user. We have shown that this can be done and that this can facilitate the user's understanding of a proof, with the biggest benefit in our view being when this engages the user — and her unique capability, intuition — in the best possible way. Proof Engineering provides a good model for how this can be done, based in large part on how it *has* been done elsewhere. Some of the areas where we feel theorem provers could benefit the most from a proof engineering approach are listed below.

Proof Term Metadata

In long proof goals, even with mathematical rendering it can be hard to identify repeated terms and learn the shape of the proof. By permitting metadata for visualization to be attached to terms — for instance highlighting a fact such as `P inCircumcircle A B C` in blue — it can be made much easier to recognise relationships between statements, such as its presence in the assumption of one implication and in the conclusion of a different implication. Propagating this from one proof step to another could take this further, giving continuity to a user's mental model of a proof by preserving colour even as the components of a proof state might change. These colours could also be used in the interactions with other systems, such as in a diagram of multiple circumcircles produced by GeoGebra or Maple.

Multiple Proof States

The modern software developer takes "lightweight branching" for granted: a simple `git checkout branch2` can replace a large codebase within a fraction of a second, and IDEs with their incremental build support can recompile and regenerate dependencies with only slightly more overhead (and even that software engineers complain about). If anything, theorem provers should be leading the way in this capacity, given the common practice in mathematical proof to "hill-climb", that is to switch among a set of active proof attempts or even definitional foundations, whilst hunting for a solution; proof planning provides one technique for managing this [155], while IsaPlanner gives a pragmatic approach for Isabelle [44]. Instead, we relied on low-level version control (git) to maintain alternate proof approaches, with the penalty of a punitive delay when switching between proofs while the prover reprocessed the entire theory.

Refactoring

Refactoring code — that is, changing a name or arguments to a function — is one of the core capabilities expected of IDEs. When viewed from a proof engineering angle, it almost beggars belief that these capabilities are almost entirely absent in theorem prover IDEs. [2] This is painful enough whenever we wish to rationalize lemma names, and we do an old-fashioned "search-and-replace" then suffer the punitive delays just mentioned, waiting for the proof to be reprocessed; but it is agonising when, time after time, we are making small changes to a loop invariant as it evolves, then regenerating the verification conditions, repairing those proofs, and then some hours later celebrating the small step forward in the proof which engendered the loop invariant change (usually with some damage to the intuition and flow which gave us the insight to make the change!). The Prover's Palette lessens the frequency with which such changes are needed, but it does not increase the "round-trip" time to assay the real effect of a change. Addressing this seemingly small problem would pay big dividends!

Research into this area is however progressing and we are encouraged by Bourke *et al.*'s Levity tool for Isabelle/HOL [22], which has began addressing some of the refactoring issues when lemmas are relocated; the tool automatically moves lemmas upward as far as possible in the theory dependency graph, increasing their potential reuse. Despite the tool occasionally moving lemmas into unnatural locations, the work is a step in the right direction. Recent work by Whiteside [183] is also encouraging. In his PhD thesis he constructs a proof language framework called Hiscript, which provides a minimal proof language, its formal semantics and a notion of statement preservation. Proof refactoring is then defined using the Hiscript framework, with over

[2] There are some efforts in this direction, but we are unaware of any which leverage the powerful libraries available (such as Eclipse's JDT), and as a result the functionality which we have seen has tended to be disappointing.

thirty refactorings being formally specified and proven to preserve the semantics of the framework.

Collaboration

Collaboration in general is another area where exciting things are happening in software engineering, not just at the code level, but in the plethora of "Cloud IDEs" where developers can work on a live, shared body of code simultaneously. In theorem proving this could be very useful, as lemmas could be assumed in one part of a proof even while someone else is proving them; or, envisaging a "TPaaS" (theorem prover as a service), a live communal prover could track the lemmas, definitions, and proofs built by anybody, on other foundations, and inform a user when a proof is already completed or suggest work which might be relevant. Proofs that are the most interesting could be "liked", inspiring the most useful prospective theories to be evolved by other people, or by software tools, or both, perhaps using the Prover's Palette approach [3].

Presentation

Presenting and communicating a result is as important as conceiving and developing it, but in this respect formal theorem proving is particularly weak. Software code is often judged for its elegance, and mathematicians' proofs even more so. In contrast, formal proofs still read like telephone directories (Section 2.4). The aim of creating a proof meant for a human is very different to the aim of creating a proof meant for a computer: they need not be incompatible, but so far, we feel they have been too closely intertwined.

We have shown many ways that our approach, and proof engineering in general, can improve the interleaved processes of modelling a theory and constructing proofs within it, but we have done little to address the impenetrable presentation of complex formal proofs. There have been attempts in this direction — supporting mathematical notation, "readable" languages such as Isar, and LaTeX-formatted PDF output — but these fall far short of the standards expected for human consumption.

Software engineering has given us the WYSIWYG principle ("what you see is what you get"), which transformed word processing, and the fourth-generation languages (visual programming). Is it time for a meta-language or visual paradigm on top

[3]We salute early efforts from in this direction undertaken in MathWeb [114], and lament their prematurity. The technology landscape has changed, with radical advances in availability of cloud resources, web API's, understanding of collaborative working, richer metadata, and — not least — faster internet access to the human users. As a result, we'd like to see their ambition revived.

of the prover's syntax? Do we need a new, machine-centric file format where multiple perspectives can be saved? We don't know the answer, but we have noticed that whenever a development environment has become too unwieldy, from the tedium of writing assembly code to the irreconcilable tensions of a format meant for both machines and people, software has responded with innovations. LCF (Section 3.2.2) was an important such innovation in mechanised theorem proving, but we are overdue for another: proof engineering may lead us to the next evolution.

The Changing Nature of Theorem Proving

Some of these ideas challenge the fundamental approach of existing theorem provers, requiring complex meta-models and new user interfaces, but this should not be surprising. The landscape of software has changed altogether in the few years since we began this research, not due to the web so much as "Web 2.0": witness distributed version control (*e.g.* Github) and online multi-user office suites (*e.g.* Google Docs) supplanting the traditional tools despite their over thirty years of development. Dramatic changes in how people do mathematics are also occurring, from crowd-sourcing and rapidly iterating proof ideas back and forth [147] to formalising proofs because they are too complex to believe them otherwise [57]. Theorem proving systems may not be recognizable in ten years' time; in fact, we hope that they are not, for there is such great scope for evolution and innovation. It is now clear that proof engineering is a more radical idea than it first appeared, but one whose time has come, as Lüth notes:

> The overall challenge in user interfaces is to leverage the underlying technology to an extent which makes it easier to do proofs in a computer than with pen and paper. Presently, this is not the case. Theorem provers tend to get in the way more often than they are helpful, and even though that is in part their duty as proof checkers, the preferable role model of a theorem prover should be that of a helpful co-author gently pointing out errors and suggesting improvements, rather than a stubborn civil servant refusing to accept the blindingly obvious because of some formality [109].

However, the major shifts in emphasis and investment needed for such an endeavour will demand a strong body of evidence as justification. We hope this research can help make that case, illustrating just some of the ways that paradigm changes elsewhere can be models for making formal theorem proving more intuitive and productive.

11.3 The Future Will Not Be Automated

When I applied for university in 1997, I chose Artificial Intelligence amidst a heady excitement with headlines championing Deep Blue's victory over Kasparov [90], a consummation of decades of research into computer chess:

> a rare, pivotal watershed beyond all other triumphs: Orville Wright's first flight, NASA's landing on the moon ... [132]

In nearly every domain I encountered, computers had skills which people could not rival. But the converse was also manifest: people have skills which computers cannot touch, and even as technology advances apace this fact stays constant. The game Go remains out of reach of our automation abilities, and technology for all we depend on it, ultimately is a mere assistant to the humans who design it.

So it is in theorem proving. When I entered my studies 15 years ago, automated techniques were routinely performing feats no human could match. This is even more the case today, as evidenced by tools such as Sledgehammer and QEPCAD, and by the scope and precision of theories captured in modern provers. But these skills are merely a part of what is required for the verification tasks we have looked at. The extensive fully formal proofs being produced today were never possible until the advent of modern computers, but equally their construction can only proceed with the domain knowledge and intuition of the human operator directing it. In our case studies, months of such human direction is required, and other proofs have taken years of such guidance [57].

This research has shown how, by study and development of theorem provers as engineering tools rather than automation systems, the interaction between various software systems and users can be made more efficient. Different computerized mathematical tools have different strengths, and by integrating them in a way which involves the human user, it is possible to make better use of each and, more importantly in our view, the user's intuition can be brought closer to the problem domain.

Given the necessity of that human interaction, and the unlikelihood of that necessity going away, the proof engineering line of research seems very promising. We are hopeful we have encouraged others to come this way, and we are sure that with their skills and intuition — as different again to our own as the computer tools are different to each other — that other researchers will further improve how these plethora of systems can collaborate.

We are reminded of another, more recent, chess milestone, less celebrated than

Deep Blue's victory but a far stronger and better inspiration. In 2005, a "freestyle" contest was held by playchess.com. This was not about man versus machine, but about teams, and the results were fascinating:

> Lured by the substantial prize money, several groups of strong grandmasters working with several computers at the same time entered the competition. At first, the results seemed predictable. The teams of human plus machine dominated even the strongest computers. The chess machine Hydra, which is a chess-specific supercomputer like Deep Blue, was no match for a strong human player using a relatively weak laptop. Human strategic guidance combined with the tactical acuity of a computer was overwhelming.
>
> The surprise came at the conclusion of the event. The winner was revealed to be not a grandmaster with a state-of-the-art PC but a pair of amateur American chess players using three computers at the same time. Their skill at manipulating and "coaching" their computers to look very deeply into positions effectively counteracted the superior chess understanding of their grandmaster opponents and the greater computational power of other participants. Weak human + machine + better process was superior to a strong computer alone and, more remarkably, superior to a strong human + machine + inferior process [97].

Chess is a metaphor for so many other domains: brute-force can be defeated by collaboration. The burgeoning field of proof engineering lets us bring this lesson to theorem proving. For in the end, the greatest success depends not on automation but on a system's capacity to augment and ultimately be guided by that uniquely human ingredient — intuition — in all its undiluted potency.

Bibliography

[1] B. Akbarpour and L. C. Paulson. *Extending a Resolution Prover for Inequalities on Elementary Functions.* Logic for Programming, Artificial Intelligence and Reasoning (LPAR), LNCS 4790: 47-61, Springer, 2007.

[2] K. Appel and W. Haken. *Solution of the Four Color Map Problem.* Scientific American, 237 (4): 108-121, Oct 1997.

[3] A. Armando and D. Zini, *Towards Interoperable Mechanized Reasoning Systems: the Logic Broker Architecture*, AI*IA, 3: 70-75, 2000.

[4] D. S. Arnon. *Geometric reasoning with logic and algebra.* Artificial Intelligence, 37: 37-60, 1988.

[5] M. Aschbacher. *The Status of the Classification of the Finite Simple Groups.* Notices of the American Mathematical Society, 51 (7): 736-740.

[6] A. Asperti. *A survey on Interactive Theorem Proving.* 2009. `http://www.cs.unibo.it/~asperti/SLIDES/itp.pdf`

[7] D. Aspinall. *Proof General: A Generic Tool for Proof Development.* Tools and Algorithms for the Construction and Analysis of Systems, Proc TACAS 2000, LNCS 1785, Springer, 2000.

[8] D. Aspinall, C. Lüth and D. Winterstein. *A Framework for Interactive Proof.* Calculemus/MKM, LNCS 4573: 161-175, Springer, 2007.

[9] J. Avigad. *Understanding, Formal Verification and the Philosophy of Mathematics.* 2010. `http://www.andrew.cmu.edu/user/avigad/Papers/understanding2.pdf`

[10] J. Avigad, K. Donnelly, D. Gray and P. Raff. *A formally verified proof of the prime number theorem.* *arXiv:cs.AI/0509025*, 2005.

[11] J. W. Backus. *The history of FORTRAN I, II and III.* Proceedings First ACM SIGPLAN conference on History of programming languages 1978, History of Programming Languages, ACM Monograph Series, Academic Press, 1981.

[12] W. W. Rouse Ball. *Mathematical Recreations and Essays.* 1st Edition, MacMillan and Co., 1892.

[13] C. Ballarin, K. Homann and J. Calmet. *Theorems and algorithms: an interface between Isabelle and Maple*. ISSAC '95, 150-157, 1995.

[14] C. Ballarin and L. Paulson. *A Pragmatic Approach to Extending Provers by Computer Algebra: with Applications to Coding Theory*. Fundamenta Informaticae, 3, IOS Press, 1999.

[15] L. J. Bass and S. R. Schubert. *On finding the disc of minimum radius containing a given set of points*, Math. Comut., 12: 712-714, 1967.

[16] B. Beckert, R. Hähnle and P. H. Schmitt. *Verification of Object-Oriented Software: The KeY Approach*, LNCS 4334, Springer, 2007.

[17] S. Berghofer and T. Nipkow. *Random testing in Isabelle/HOL.* 2nd IEEE International Conference on Software Engineering and Formal Methods (SEFM 2004), IEEE Computer Society Press, 2004.

[18] Y. Bertot. *The CtCoq system: design and architecture.* Formal Aspects of Computing, 11: 225-243, 1999.

[19] Y. Bertot, N. Magaud and P. Zimmermann. *A proof of GMP square root.* Journal of Automated Reasoning, 29: 225-252, 2002.

[20] J. C. Blanchette. *Hammering Away: A User's Guide to Sledgehammer for Isabelle/HOL.* 2013. `http://isabelle.in.tum.de/doc/sledgehammer.pdf`

[21] J. C. Blanchette and T. Nipkow. *Nitpick: A Counterexample Generator for Higher-order Logic based on a Relational Model Finder.* First International Conference on Interactive Theorem Proving, LNCS 6172: 131-146, Springer, 2010.

[22] T. Bourke, M. Daum, G. Klein and R. Kolanski. *Challenges and Experiences in Managing Large-Scale Proofs.* AISC/MKM/Calculemus, LNCS, 7362: 32-48, Springer 2012.

[23] J. Brillhart, J. Tonascia and P. Winberger. *On the Fermat Quotient.* Computers and Number Theory, New York: Academic Press, 213-222, 1971.

[24] C. W. Brown. *QEPCAD B: a program for computing with semi-algebraic sets using CADs.* ACM SIGSAM Bulletin, 37 (4), 2003.

[25] C. Brun, J. Dufourd and N. Magaud. *Designing and proving correct a convex hull algorithm with hypermaps in Coq.* Journal of Computational Geometry, 45: 436-457, 2012.

[26] B. Buchberger, A. Crăciun, T. Jebelean, L. Kovács, T. Kutsia, K. Nakagawa, F. Piroi, N. Popov, J. Robu, M. Rosenkranz and W. Windsteiger. *Theorema: Towards computer-aided mathematical theory exploration.* Journal of Applied Logic, 2005.

[27] J. Buhler, R. Crandell, R. Ernvall, T. Metsänkylä. *Irregular primes and cyclotomic invariants to four million.* Math. Comp. (AMS) 61 (203): 151-153, 1993.

[28] A. Bundy. *The automation of proof by mathematical induction.* In A. Robinson and A. Voronkov, *Handbook of automated reasoning, 1*. Amsterdam: Elsevier, 845-911, 2001.

[29] J. Calmet, C. Ballarin and P. Kullmann, *Integration of Deduction and Computation*. Applications of Computer Algebra, 2001.

[30] University of Cambridge Computer Laboratory, Automated Reasoning Group. *The HOL System.* `http://www.cl.cam.ac.uk/research/hvg/HOL/`

[31] J. E. Caplan and M. T. Harandi. *A logical framework for software proof re-use.* Proceedings of the 1995 Symposium on Software Reusability, 106-113, 1995.

[32] L. Carroll. *The Mathematical Recreations of Lewis Carroll: Pillow Problems and A Tangled Tale.* Pillow Problems Problem 5, pp 2,

[33] S. C. Chou, X. S. Gao and J. Z. Zhang. *Automated generation of readable proofs with geometric invariants, I. multiple and shortest proof generation.* Journal of Automated Reasoning, 17: 325-347, 1996.

[34] S. C. Chou, X. S. Gao and J. Z. Zhang. *Automated generation of readable proofs with geometric invariants, II. theorem proving with full angles.* Journal of Automated Reasoning, 17: 349-370, 1996.

[35] A. Church. *A formulation of the simple theory of types.* Journal of Symbolic Logic, 5: 56-68, 1940.

[36] E. M. Clarke and X. Zhao. *Analytica — A Theorem Prover in Mathematica*. Proceedings of the 11th International Conference on Automated Deduction: Automated Deduction, CADE-11, LNCS 607: 761-765, Springer, 1992.

[37] P. J. Cohen. *Skolem and pessimism about proof in mathematics*. Philisophical Transactions of the Royal Society A, 2005.

[38] H. Coelho and L. M. Pereira. *Automated reasoning in geometry theorem proving with Prolog.* Journal of Automated Reasoning 2(4), 329-390, 1986.

[39] G. E. Collins. *Quantifier elimination for real closed fields by cylindrical algebraic decompostion.* Automata Theory and Formal Languages 2nd GI Conference, LNCS 33: 134-183, Springer, 1975.

[40] G. E. Collins and H. Hong. *Partial cylindrical algebraic decomposition for quantifier elimination.* Journal of Symbolic Computation, 12 (3): 299-328, 1991.

[41] T. H. Cormen, C. E. Leiserson and R. L. Rivest. *Introduction to Algorithms*. MIT Electrical Engineering and Computer Science Series, MIT PRESS, 1998.

[42] The Coq Development Team. *The Coq proof assistant reference manual: Version 8.0*, LogiCal Project, 2004, `http://coq.inria.fr`

[43] L. A. Dennis, G. Collins, M. Norrish, R. Boulton, K. Slind and T. Melham. *The PROSPER Toolkit.* Int. J. Software Tools for Technology Transfer., 4 (2): 189-210, 2003.

[44] L. Dixon and J. D. Fleuriot. *A Proof-Centric approach to Mathematical Assistants.* Journal of Applied Logic: Special Issue on Mathematics Assistance Systems, 2004.

[45] J. Dufourd and Y. Bertot. *Formal study of plane Delaunay triangulation.* Interactive Theorem Proving, LNCS 6172: 211-226, Springer, 2010.

[46] Eclipse Foundation. *Eclipse*. `http://www.eclipse.org`

[47] Eclipse Foundation. *SWT: Standard Widget Toolkit*. `http://www.eclipse.org/swt`

[48] E. W. Elcock. *Representation of knowledge in geometry machine.* Machine Intelligence, 8, 11-29, 1977.

[49] F. Engel and M. Dehn, *Moritz Pasch. Jahresbericht der Deutschen Mathematiker Vereinigung*, 44 (5/8): 133, 1934.

[50] C. Engelman. *MATHLAB: a program for on-line machine assistance in symbolic computations.* Proceedings AFIPS '65, Fall, part II, 117-126, New York: ACM, 1965.

[51] C. Engelman. *The Legacy of MATHLAB 68.* Proc 2nd Symp on Symbolic and Algebraic Manip, ACM, Mar 1971.

[52] D. Fearnley-Sander. *The idea of a diagram.* Resolution of equations in algebraic structures, 1: 127-150, 1989.

[53] S. Feferman. *Mathematical Intuition vs. Mathematical Monsters.* Synthese, 125: 317-332, 2000.

[54] J. Filliâtre and A. Paskevich. *Why3 – Where Programs Meet Provers*. ESOP'13 22nd European Symposium on Programming, LNCS 7792, Springer, 2013.

[55] L. Fisher. *Crashes, Crises, and Calamities: How We Can Use Science to Read the Early-Warning Signs.* Basic Books: 2011.

[56] J. Fleuriot. *A Combination of Geometry Theorem Proving and Nonstandard Analysis with Applications to Newton's Principia.* Springer, 2001.

[57] The Flyspeck Project. `https://code.google.com/p/flyspeck`

[58] L. Fousse, G. Hanrot, V. Lefèvre, P. Pélissier and Paul Zimmermann. *MPFR: A multiple-precision binary floating-point with correct rounding.* ACM Trans. Math. Softw., 33, 2007.

[59] G. Frege. *Begriffsschrift: eine der arithmetischen nachgebildete Formelsprache des reinen Denkens.* Halle, 1879.

[60] H. Freudenthal. *The main trends in the foundations of geometry in the 19th century.* Logic, Methodology, and Philosophy of Science, 613 - 621, Stanford, CA: Stanford University Press, 1962.

[61] L. Fuchs and L. Théry. *A formalization of Grassmann-Cayley algebra in COQ and its application to theorem proving in projective geometry.* ADG 2010, Springer, 2011.

[62] A. Garnham. *Artificial Intelligence, An Introduction*. Routledge and Kegan Paul, 116-118, 1988.

[63] H. Gelernter, *Realization of a geometry-theorem proving machine.* Computers and thought, 134-152, MIT Press, 1995.

[64] GeoGebra. `http://www.geogebra.org/cms/en`

[65] K. Gödel. *Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme*. Monatshefte für Mathematik und Physik, 38: 173-98, 1931. English translation: `http://www.research.ibm.com/people/h/hirzel/papers/canon00-goedel.pdf`

[66] G. Gonthier. *The four colour theorem: Engineering of a formal proof.* Computer. Mathematics: 8th Asian Symposium, ASCM 2007, LNCS 5081: 333, Springer, 2008.

[67] M. Gordon and T. Melham. *Introduction to HOL: A theorem proving environment for Higher Order Logic.* Cambridge University Press, 1993.

[68] F. Q. Gouvêa, *p-adic Numbers: An Introduction*, Springer, 1997.

[69] R. L. Graham. *An efficient algorithm for determining the convex hull of a finite planar set*. Info. Proc. Lett. 1, 132-133, 1972.

[70] J. Greeno, M. E. Magone, S. Chaiklin. *Theory of constructions and set in problem solving.* Memory and Cognition 7 (6): 445-461, 1979.

[71] L. Guibas and J. Stolfi. *Primitives for the Manipulation of General Subdivisions and the Computation of Voronoi Diagrams.* ACM TOG, 4 (2): 74-123, 1985.

[72] J. Hadamard. *The Mathematician's Mind: The Psychology of Invention in the Mathematical Field*. Princeton Science Library, 1945.

[73] H. Hahn. *The crisis in intuition.* In [74], 73-102, 1933.

[74] H. Hahn. *Empiricism, Logic and Mathematics.* Dordrecht: Reidel, 1980.

[75] T. C. Hales. *A proof of the Kepler conjecture.* Annals of Mathematics, Second Series 162 (3): 1065-1185, 2005.

[76] T. C. Hales. *The Jordan curve theorem, formally and informally.* American Mathematical Monthly, 114 (10): 882-894, 2007.

[77] T. C. Hales. Personal correspondence, 2012.

[78] R. Hardy. *Formal Methods for Control Engineering: A Validated Decision Procedure for Nichols Plot Analysis.* PhD thesis, St. Andrews University, 2006.

[79] J. Harrison. *Floating-Point Verification using Theorem Proving*. Proceedings of SFM 2006, the 6th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, LNCS 2965: 211-242, Springer, 2006.

[80] J. Harrison. *Formalizing an analytic proof of the Prime Number Theorem.* Journal of Automated Reasoning, 43: 243-261, 2009.

[81] J. Harrison. *Without Loss of Generality*, TPHOLs 2009, LNCS 5674: 43-59, Springer, 2009.

[82] J. Harrison. *The HOL Light theorem prover.* `http://www.cl.cam.ac.uk/~jrh13/hol-light/index.html`

[83] J. Harrison and L. Théry. A Skeptic's Approach to Combining HOL and Maple, *Journal of Automated Reasoning*, 1997.

[84] A. Heneveld. *Using Features for Automated Problem Solving.* PhD Thesis, Univeristy of Edinburgh, 2007.

[85] D. Hilbert. *The Foundations of Geometry*. The Open Court Company, 11th edition, 2001. Translation by Leo Unger.

[86] D. Hilbert. *Über die Grundlagen der Geometrie*, 1902. In M. Hallet and U. Majer, Lectures on the Foundations of Geometry 1891 - 1902, Springer, 2004.

[87] C. A. R. Hoare. *An axiomatic basis for computer programming.* Communications of the ACM, 12 (10): 576-580, 1969.

[88] T. Hobbes. *Leviathan.* 1651.

[89] L. Hongbo and C. Minteh. *Clifford Algebraic Reduction Method for Automated Theorem Proving in Differential Geometry.* Journal of Automated Reasoning, 21 (1): 1-21, 1998.

[90] IBM. *Deep Blue.* `http://www-03.ibm.com/ibm/history/ibm100/us/en/icons/deepblue/`

[91] P. Janičić, J. Narboux and P. Quaresma. *The Area Method : a Recapitulation*, Journal of Automated Reasoning, 48: 489-532, 2012.

[92] P. Janičić and P. Quaresma, P. *System Description: GCLCprover + GeoThms.* Automated Reasoning, Lecture Notes in Artificial Intelligence, 4130: 145-150. Springer-Verlag, 2006.

[93] jEdit. `http://www.jedit.org`

[94] C. Kaliszyk and F. Wiedijk. *Certified Computer Algebra on top of an Interactive Theorem Prover.* Proceedings of the 14th symposium on Towards Mechanized Mathematical Assistants: 6th International Conference (Calculemus '07 / MKM '07), LNCS 4573: 94-105, Springer, 2007.

[95] D. Kapur. *Using Gröbner bases to reason about geometry problems.* Journal of Symbolic Computation, 2: 399-408, 1986.

[96] D. Kapur. *A refutational approach to geometry theorem proving.* Artificial Intelligence, 37: 61-93, 1988.

[97] G. Kasparov. *The Chess Master and the Computer.* The New York Review of Books, 2010.

[98] M. Kaufmann and J. S. Moore. *ACL2.* `http://www.cs.utexas.edu/~moore/acl2/`

[99] M. Kerber, M. Kohlhase and V. Sorge. *Integrating Computer Algebra into Proof Planning*. Journal of Automated Reasoning, 21 (3): 327-355, 1998.

[100] F. Klein. *Elementary Mathematics from an Advanced Standpoint: Geometry.* Dover, 2004. Reprint of edition first published by Springer, 1928.

[101] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch and S. Winwood. *seL4: Formal Verification of an OS Kernel*, ACM Symposium on Operating Systems Principles, 207-220, ACM, 2009.

[102] M. Kline. *Mathematics: The Loss of Certainty*. New York: Oxford University Press, 1980.

[103] K. Koffka. *Principles of Gestalt Psychology.* 1935. (Routledge, 1999.)

[104] D. E. Knuth. *Axioms and Hulls.* Lecture Notes in Computer Science, Volume 606. Springer, 1992.

[105] B. Kutzler and S. Stifter. *On the application of Buchberger's algorithm to automated geometry theorem proving.* Journal of Symbolic Computation, 2: 389-397, 1986.

[106] J. H. Laning and N. Zierler. *A Program For Translation of Mathematical Equations for Whirlwind I.* Engineering Memorandum E-364, Instrumentation Laboratory, Massachusetts Institute of Technology, 1954.

[107] J. E. Littlewood. *The Riemann hypothesis*. In *The scientist speculates: an anthology of partly baked idea*. Basic Books, New York, 1962.

[108] Z. Luo. *Developing reuse technology in proof engineering.* AISB95, Workshop on Automated Reasoning: bridging the gap between theory and practice, Sheffield, UK, 1995.

[109] C. Lüth. *User Interfaces for Theorem Provers: Necessary Nuisance or Unexplored Potential?* AVOCS-09. `http://journal.ub.tu-berlin.de/eceasst/article/view/322/305`

[110] C. Lüth and D. Walter. *Certifiable specification and verification of C programs.* FM 2009, LNCS 5850, Springer, 2009.

[111] K. Maling. *The LISP Differentiation Demonstration Program.* MIT AI Memo 10, Cambridge, MA, 1959.

[112] F. Marić, I. Petrović, D. Petrović and P. Janičić. *Formalization and Implementation of Algebraic Methods in Geometry.* Proceedings of First Workshop on CTP Components for Educational Software (THedu'11), Electronic Proceedings in Theoretical Computer Science, 79: 63-81, 2011.

[113] Mathematica. `http://www.wolfram.com/mathematica`

[114] MathWeb. `http://www.mathweb.org`

[115] T. Matsuyama and T. Nitta. *Geometric theorem proving by integrated logical and algebraic reasoning.* Artificial Intelligence, 75: 93-113, 1995.

[116] J. McCarthy. *History of LISP*. Newsletter, ACM SIGPLAN Notices, 13 (8): 217-223, 1978.

[117] W. McCune. *OTTER 3.3 Reference Manual.* CoRR, cs.SC/0310056, 2003.

[118] S. McLaughlin and J. Harrison. A Proof-Producing Decision Procedure for Real Arithmetic. *20th International Conference on Automated Deduction*. LNCS 3632, Springer, 2005.

[119] L. Meikle and J. Fleuriot. *Formalizing Hilbert's Grundlagen in Isabelle/Isar*. TPHOLs 2003, LNCS 2758: 319-334, Springer, 2003.

[120] L. Meikle and J. Fleuriot. *Mechanical Theorem Proving in Computational Geometry.* Automated Deduction in Geometry 2004. LNCS 3763: 1-18, 2006.

[121] L. Meikle and J. Fleuriot. *Prover's Palette: A User-Centric Approach to Verification with Isabelle and QEPCAD-B.* CAV. LNCS 5123: 309-313, Springer, 2008.

[122] L. Meikle and J. Fleuriot. *Combining Isabelle and QEPCAD-B in the Prover's Palette.* AISC/MKM/Calculemus. LNCS 5144: 315-330, Springer, 2008.

[123] L. Meikle and J. Fleuriot. *Automation for Geometry in Isabelle/HOL* PAAR-2010: Proceedings of the 2nd Workshop on Practical Aspects of Automated Reasoning. 84-94, 2012.

[124] L. Meikle and J. Fleuriot. *Integrating Systems around the User: Combining Isabelle, Maple and QEPCAD in the Prover's Palette.* Electronic Notes in Theoretical Computer Science, 285: 115-119, 2012.

[125] G. Melquiond and S. Pion. *Formally certified floating-point filters for Homogeneous Geometric Predicates.* Theoretical Informatics and Applications, 41: 57-69, 2007.

[126] J. Meng, C. Quigley and L. C. Paulson, *Automation for interactive proof: first prototype.* Inf. Comput., 204 (10): 1575-1596, 2006.

[127] A. Mulhern, C. Fischer and B. Liblit. *Tool Support for Proof Engineering*. Proceedings from UITP. 2006.

[128] J. Mumma. *Intuition Formalized: Ancient and Modern Methods of Proof in Elementary Euclidean Geometry.* PhD Dissertation, Carnegie Mellon University, 2006. http://www.andrew.cmu.edu/jmumma

[129] J. Narboux. *A decision procedure for geometry in Coq.* TPHOLs, LNCS 3223: 225-240, Springer, 2004.

[130] A. Nevis. *Plane geometry theorem proving using forward chaining.* Artificial Intelligence, 6 (1): 1-23, 1975.

[131] L. P. Nieto and T. Nipkow. Hoare Logic (Isabele Theory). http://www.cl.cam.ac.uk/research/hvg/Isabelle/dist/library/HOL/HOL-Hoare/Hoare_Logic.html

[132] M. Newborn. *Deep Blue — an artificial intelligence milestone.* Springer, 2003.

[133] T. Nipkow. *Structured Proofs in Isar/HOL.* Types for Proofs and Programs (TYPES 2002), LNCS 2646: 259-279, Springer, 2003.

[134] T. Nipkow, L. Paulson and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, LNCS 2283, Springer, 2002.

[135] Nova (documentary series). *The Proof*, 1997. http://www.pbs.org/wgbh/nova/proof/

[136] Nova (web site). *The Theory Behind the Equation.* http://www.pbs.org/wgbh/nova/physics/theory-behind-equation.html

[137] The OpenMath Society. http://www.openmath.org

[138] J. O'Rourke. *Computational Geometry in C.* Cambridge University Press, 1994.

[139] Oxford Dictionaries (web site). http://oxforddictionaries.com/

[140] M. Pasch. *Vorlesungen uber Neuere Geometrie.* Teubner, Leipzig, Germany, 1882.

[141] L. C. Paulson and K. W. Susanto. *Source-level proof reconstruction for interactive theorem proving.* Theorem Proving in Higher Order Logics: TPHOLs, LNCS 4732: 232-245, Springer, 2007.

[142] G. Perelman. *Finite extinction time for the solutions to the Ricci flow on certain three-manifolds.* `arXiv:math.DG/0307245`, 2003.

[143] D. Pichardie and Y. Bertot. *Formalizing Convex Hull Algorithms.* TPHOLs, LNCS 2152: 346-361, Springer, 2001.

[144] H. Poincaré. *Intuition and Logic in Mathematics.* Published as part of 'La valeur de la science' in 1905. Translated into English by G B Halsted, published in The Value of Science, 1907.

[145] R. Pollack. *How to Believe a Machine-Checked Proof.* Twenty Five Years of Constructive Type Theory, Oxford Univiversity Press, 1998.

[146] G. Pólya. *How to Solve It.* Princeton University Press, 1945.

[147] The polymath blog. `http://polymathprojects.org/`

[148] F. P. Preparata and M. I. Shamos. *Computational Geometry: An Introduction.* Springer, 1985.

[149] D. Priest. *Algorithms for Arbitrary Precision Floating Point Arithmetic.* Tenth Symposium on Computer Arithmetic, IEEE, 132-143, 1991.

[150] Princeton University, Institue for Advanced Study. Hermann Weyl, (biographical site). `http://www.ias.edu/people/weyl/legacy`

[151] QEPCAD. `http://www.usna.edu/CS/~qepcad/B/QEPCAD.html`

[152] W. V. O. Quine. *Paradox.* Scientific American, , 206 (4): 84-96, 1962.

[153] M. Raussen and C. Skau. *Interview with Michael Atiyah and Isadore Singer.* European Mathematical Society Newsletter, 24-30, 2004.

[154] A. Riazanov and A. Voronkov. *Vampire 1.1 (system description).* In R. Gore, A. Leitsch, and T. Nipkow, eds, Automated Reasoning, First International Joint Conference, IJCAR 2001, LNAI 2083: 376-380, Springer, 2001.

[155] J. Richardson. *A Semantics for Proof Plans with Applications to Interactive Proof Planning.* LPAR '02, 337-351, Springer, 2002.

[156] R. H. Risch. *The solution of the problem of integration in finite terms.* Bulletin of the American Mathematical Society, 76 (3): 605-608, 1970.

[157] J. F. Ritt. *Differential Algebra.* American Mathematical Society, New York, 1950.

[158] N. Robertson, D. P. Sanders, P. Seymour, R. Thomas. *The Four-Colour Theorem.* J. Combin. Theory Ser. B, 70 (1): 2-44, 1997.

[159] A. Robinson. *Non-standard Analysis.* North-Holland, 1980.

[160] J. Robu. *Geometry theorem proving in the frame of the Theorema project.* Ph.D. thesis, Johannes Kepler Universitat, Linz, 2002.

[161] Royal Society. *The Nature of Proof*, meeting Oct 2004.

[162] P. Rudnicki. *An overview of the Mizar project.* In Workshop on Types for Proofs and Programs, Chalmers University of Technology, 1992.

[163] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach.* New Jersey: Prentice Hall, 1995.

[164] B. Samples. *Metaphoric Mind: A Celebration of Creative Consciousness.* Addison Wesley Longman, 1976.

[165] D. Schlimm. *Pasch's Philosophy of Mathematics.* The Review Of Symbolic Logic, 3 (1): 2010.

[166] S. Schulz. *System description: E 0.81.* In D. Basin and M. Rusinowitch, editors, Automated Reasoning — Second International Joint Conference, IJCAR 2004, LNAI 3097: 223-228, Springer, 2004.

[167] A. Seidenberg. *A new decision method for elementary algebra.* Annals of Mathematics, 60: 365-374, 1954.

[168] N. Shankar. *Metamathematics, Machines and Gödel's Proof.* Cambridge tracts in theoretical computer science, 38, Cambridge University, 1994.

[169] B. Shneiderman and C. Plaisant. *Designing the User Interface.* Addison-Wesley, 2009.

[170] J. Slagle. *A Heuristic Program that Solves Symbolic Integration Problems in Freshman Calculus.* New York: Journal of the ACM, 10(4): 507-520, Oct 1963.

[171] SRI. *PVS Specification and Verification System.* `http://pvs.csl.sri.com/`

[172] J. Stark and A. Ireland. *Invariant Discovery via Failed Proof Attempts.* LNCS 1559, 271, Springer, 1998.

[173] L. Théry, Y. Bertot and G. Kahn. *Real theorem provers deserve real user-interfaces.* SDE 5: Proceedings of the Fifth ACM SIGSOFT Symposium on Software Development Environments, 120-129, 1992.

[174] P. Thompson. *The Nature and Role of Intuition in Mathematical Epistimology*, Philosophia, 26: 279-319, 1998.

[175] A. Tiwari. *PVS-QEPCAD.* `http://www.csl.sri.com/users/tiwari/qepcad.html`

[176] C. Urban, L. Paulson, and M. Norrish. *The Isabelle Cookbook.* `http://www.inf.kcl.ac.uk/staff/urbanc/Cookbook`

[177] M. J. G. Veltman. *Schoonschip, a program for symbol handling.* 1964 / 2001. `http://www-personal.umich.edu/~williams/archive/schoonschip/schipman.pdf`

[178] S. S. Wagstaff, Jr. *The irregular primes to 125000.* Math. Comp. AMS, 32(142): 583-591, 1978.

[179] C. Weidenbach, D. Dimova, A. Fietzke, R. Kumar, M. Suda and P. Wischnewski. *SPASS Version 3.5.* In *Twenty-second International Conference on Automated Deduction*, CADE 2009, LNCS 5663: 140-145, Springer, 2009.

[180] M. Wenzel. *Isabelle/jEdit: a prover IDE within the PIDE framework.* Conference on Intelligent Computer Mathematics, LNCS 7362: 468-471, Springer, 2012.

[181] H. Weyl. *Mathematics and logic.* American Mathematics Monthly, 53 (13), 1946.

[182] I. Whiteside, D. Aspinall, L. Dixon, and G. Grov. *Towards Formal Proof Script Refactoring*. Calculemus/MKM, LNCS 6824: 260-275, Springer, 2011.

[183] I. Whiteside. *Refactoring Proofs.* PhD Thesis, Univeristy of Edinburgh, 2013.

[184] F. Wiedijk. *The Seventeen Provers of the World.* `http://www.cs.ru.nl/~freek/comparison/comparison.pdf`

[185] S. Wilson and J. D. Fleuriot. *Geometry Explorer: Combining dynamic geometry, automated geometry theorem proving and diagrammatic proofs.* Electronic Notes in Theoretical Computer Science (in press), 2006.

[186] L. Wittgenstein. *Philosophical Grammar.* Blackwell, Oxford, 1978.

[187] Workshop on Invariant Generation. `http://cs.nyu.edu/acsys/wing2012`

[188] W. Wu. *Basic principles of mechanical theorem proving in elementary geometries.* Journal of Automated Reasoning, 2: 221-252, 1986.

[189] K. Zee, V. Kuncak and M. Rinard. *An Integrated Proof Language for Imperative Programs*. ACM SIGPLAN Conf. Programming Language Design and Implementation (PLDI), SIGPLAN Not. 44.6: 338-351, 2009.