Automation for Geometry in Isabelle/HOL

Laura I. Meikle
School of Informatics,
University of Edinburgh,
Edinburgh, U.K.
lauram@dai.ed.ac.uk
and Jacques D. Fleuriot
School of Informatics,
University of Edinburgh,
Edinburgh, U.K.
jdf@inf.ed.ac.uk

Abstract

We present an implementation of a theory of two-dimensional geometry based on the signed area of triangles, together with a collection of automatic and interactive techniques we have developed to assist in our formal verification of geometric algorithms.

1 Introduction

Despite the successes of current state-of-the-art, fully automatic theorem provers, their results are far from replicating the expertise of human mathematicians. The search space for complex problems tends to explode very quickly, rendering them impractical for many verification tasks. As a result, much work in mechanised reasoning has focussed on the usability of systems for interactive theorem proving, where a human user guides the proof attempt. The end product of this process is a body of formalised mathematics which has the correctness guarantees that machines can provide but which is much more sophisticated than current systems could produce automatically.

Unfortunately, this approach requires a great deal of effort from the human user; not only do they require a good insight into the direction a proof should take, but they are often burdened with the tedium of proving enormous amounts of low level detail, which can detract from the main proof endeavour. For these reasons, there is a growing interest in semi-automated theorem proving, where a system incorporates automation techniques within an interactive theorem proving environment. These systems provide the best of both worlds: a user can observe an automated proof attempt, using its results when applicable, or instead manually guide the system when the automation flounders or when they have a good sense of how to proceed.

One such system is the generic theorem prover Isabelle [11], which has an extensive library of theories and some automatic proof methods which combine simplification and classical reasoning. Our interests have concentrated on using Isabelle for geometry theorem proving, and more recently for proving the correctness of geometric algorithms using the notion of signed areas. This interim paper describes parts of our Isabelle theory and the framework that we have developed for proving geometric algorithms, with a particular focus on the automation, systems integrations and related techniques that we have built in the course of this work.

2 A Theory of Planar Geometry Using Signed Area

The algorithms that we have been formally verifying are taken from the field of computational geometry. Specifically, we have been reasoning about convex hulls and Delaunay triangulations. Our formalisations have focussed on two-dimensional problems as these provide a clear appreciation of the difficulties that run at the core of many of the algorithms and we believe the lessons learnt from these proofs will scale easily when dealing with real-world problems that may involve higher dimensions. The algorithms share the common feature that they compute the positions of a set of points relative to each other. This led us to conclude that the concept of signed area would be a suitable representation.

We began by defining the type `point` as any 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 verifications relied upon reasoning about the relative positions of points, we needed to formlise this notion. For this we used the *signed area* of a triangle; with the convention being that if the points are ordered anti-clockwise, the area is positive, and if the points are ordered clockwise, the area is negative. In our theory this was formalised as [1]:

```
constdefs signedArea :: "[point, point, point] ⇒ real"
      "signedArea a b c ≡ (xCoord b - xCoord a)*(yCoord c - yCoord a)
                        - (yCoord b - yCoord a)*(xCoord c - xCoord a)"
```

where the predicates `xCoord` and `yCoord` were formally represented by:

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

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

Using this definition it was then easy to formally represent the orientation of points; we say that three points a, b and c make a *left turn* if they make an anti-clockwise cycle:

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

We deviate from many geometry theories by including so-called *degenerate* cases where the points may be collinear, or equivalently, the area of the triangle they define is zero:

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

~~A consequence of permitting collinea~~rity is that an *ordering* on points along a line must be

[1] As the magnitude of the area was irrelevant in our proofs we simplified the usual definition by omitting the factor of $\frac{1}{2}$.

established. We achieved this by defining the concept of between. For collinear points a, b and c, we represent and define b lying between a and c as follows:

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

We have chosen this definition in preference to more succinct variations because the existence clause facilitates instantiating witnesses. Our theory also includes the obvious lemmas concerning these predicates, omitted here for brevity (although some relevant ones are introduced in in the subsequent section).

3 Extending Isabelle's Simplifier and Classical Reasoner

To a human mathematician the statement that three points are collinear is interpreted in only one way. However, to a computer the terms `collinear` a b c and `collinear` b a c are symbolically evaluated and interpreted differently. 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.

Isabelle makes it easy for a user to encode this type of automation by enabling them to extend its simplifier and classical reasoner on demand. In the following subsections we present the rules which have been added to automate much of the geometric reasoning often required in verification tasks.

3.1 Simplification Rules

The standard simplification tactic—`simp`—is one of the single most powerful automation tools in Isabelle. 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, and even in some cases entire decision procedures can be encoded and goals proven automatically.

The following set of simplification rules allows 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 a proof, omitted here in the interest of space), 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, our theory adds 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"
notBetweenSelf2 [simp]: "¬ b 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 between 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. We have found it difficult to understand why this looping occurs in certain situations, even after inspecting the trace output. 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.

For our purposes though, another means of easily automating the "conditional rewrites" in Isabelle does exist. One can extend the "classical reasoner" rather than the simplifier. This approach shall be looked at next.

3.2 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 [11]. 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". To add to confusion Isabelle distinguishes between two types of elimination rules: if information is lost then the rule is called a destruction rule (although in the usual natural deduction sense it would just be called an elimination 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 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"
```

3.3 Limitations of Adding Automatic Rules

The simplification, introduction and elimination rules above 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 are two specific limitations we wish to point out.

The first of these is that 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.

Secondly, if a user wants to add more sophisticated automation they have to create their own tactics. 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 “Isar”.

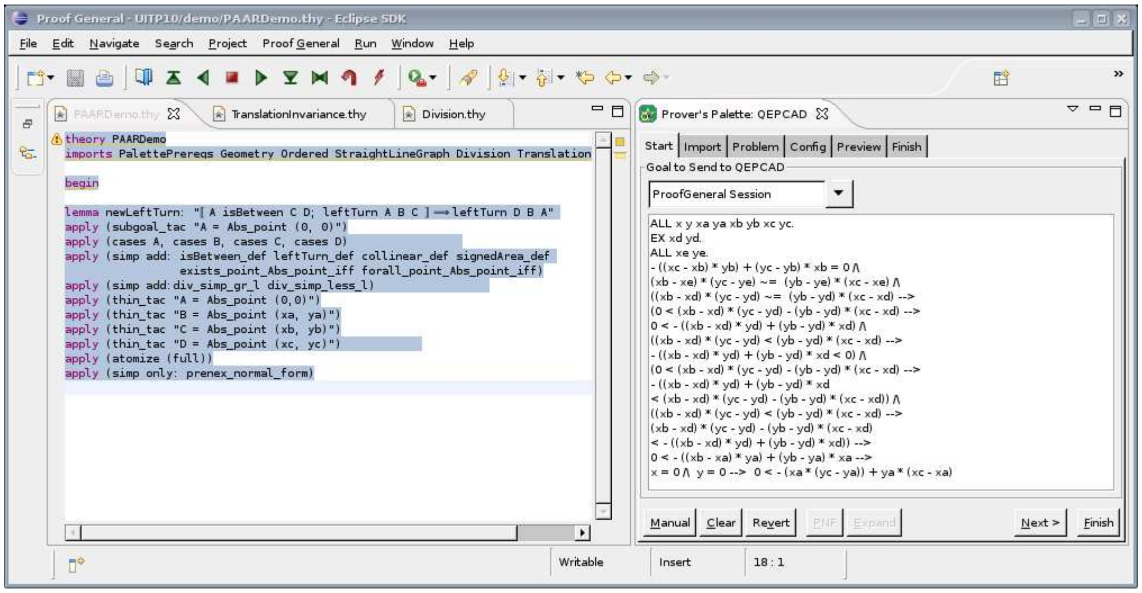
4 Integrating with QEPCAD

The simplification rules in the previous section can automatically handle some of the simplest problems we encountered in our verification, but they do not automatically solve any of what we consider our non-trivial lemmas. They assist by removing a great deal of the tedious manipulations—which is what they were intended for. For deeper problems, we have turned to some of the existing techniques for automatic geometry theorem proving (GTP).

Extensive research into mechanical GTP has been carried out over the past 40 years, and as a result it is a mature field. It has split into two main paradigms: the algebraic methods and the synthetic (or coordinate free) techniques. The latter have the advantage that they can often produce short, readable and intuitive proofs which are usually based on geometrically meaningful notions such as full-angles between lines [4] or signed areas of triangles [3]. This signed area method may seem like an ideal candidate for automatically discharging many of our proof obligations, and has recently been implemented in Coq [8]. Unfortunately the basic method does not apply to inequalities, making it inapplicable to our lemmas about betweenness, and the variants which do handle inequalities resort to algebraic methods such as cylindrical algebraic decomposition. This has led us to adopt the powerful and complete algebraic approach of Hong and Collins, based on partial cylindrical algebraic decomposition [5]. This provides a decision procedure for quantifier elimination over real closed fields. Another benefit afforded by this approach is that many other formal reasoning tasks will benefit from this automation in Isabelle, for example the verification of control and hybrid systems.

Rather than implement this decision procedure ourselves, we have decided to integrate Isabelle with the external tool QEPCAD-B which implements cylindrical algebraic decomposition efficiently in C [2] As we believe other other external tools can also provide great assistance in the proof development process, we have sought to build a general framework that can simplify and automate the ways that existing best-of-breed tools can be accessed. Of course, where it is feasible in terms of development time and execution time, decision procedures embedded within Isabelle are clearly preferable because of the soundness guarantees of both the methods and the integration. However our focus has been primarily on integrating with external tools due to the wide variety and coverage of such tools. We note that in some cases these tools may generate information which can guide or generate pure-Isabelle proofs which do not rely on the external systems; our integration with QEPCAD can for example produce witnesses and counterexamples in some situtions In some instances though, we have been willing to use the results of these external tools without formal proof (i.e. the external tools act as oracles). We have done the latter 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 with the anticipation that fully automated, formally correct methods will ultimately be available for proving these statements.

Our integration with QEPCAD has been done in a framework we have developed called the Prover's Palette (Figure 1). This framework is built upon the recently developed Eclipse Proof General [1], and takes advantage of its broker middleware. The integration and the framework are described in detail elsewhere [9]; however there is a large amount of automation in both the integration framework and the QEPCAD integration built on this framework, much of it developed more recently. This section focuses not on the integration framework so much as on the automation and other techniques which are applicable more widely, to theories of planar geometry or to others developing automation tools.



```
theory PAARDemo
imports PalettePrereqs Geometry Ordered StraightLineGraph Division Translation

begin

lemma newLeftTurn: "⟦ A isBetween C D; leftTurn A B C ⟧ ⟹ leftTurn D B A"
apply (subgoal_tac "A = Abs_point (0, 0)")
apply (cases A, cases B, cases C, cases D)
apply (simp add: isBetween_def leftTurn_def collinear_def signedArea_def
                 exists_point_Abs_point_iff forall_point_Abs_point_iff)
apply (simp add: div_simp_gr_l div_simp_less_l)
apply (thin_tac "A = Abs_point (0,0)")
apply (thin_tac "B = Abs_point (xa, ya)")
apply (thin_tac "C = Abs_point (xb, yb)")
apply (thin_tac "D = Abs_point (xc, yc)")
apply (atomize (full))
apply (simp only: prenex_normal_form)
```

```
ALL x y xa ya xb yb xc yc.
EX xd yd.
ALL xe ye.
- ((xc - xb) * yb) + (yc - yb) * xb = 0 /\
(xb - xe) * (yc - ye) ~=  (yb - ye) * (xc - xe) /\
((xb - xd) * (yc - yd) ~=  (yb - yd) * (xc - xd) -->
(0 < (xb - xd) * (yc - yd) - (yb - yd) * (xc - xd) -->
0 < - ((xb - xd) * yd) + (yb - yd) * xd) /\
((xb - xd) * (yc - yd) < (yb - yd) * (xc - xd) -->
- ((xb - xd) * yd) + (yb - yd) * xd < 0) /\
(0 < (xb - xd) * (yc - yd) - (yb - yd) * (xc - xd) -->
- ((xb - xd) * yd) + (yb - yd) * xd
< (xb - xd) * (yc - yd) - (yb - yd) * (xc - xd)) /\
((xb - xd) * (yc - yd) < (yb - yd) * (xc - xd) -->
(xb - xd) * (yc - yd) - (yb - yd) * (xc - xd)
< - ((xb - xd) * yd) + (yb - yd) * xd)) -->
0 < - ((xb - xa) * ya) + (yb - ya) * xa -->
x = 0 /\  y = 0 -->  0 < - (xa * (yc - ya)) + ya * (xc - xa)
```

Figure 1: The Prover's Palette with Isabelle and QEPCAD

4.1 Automatic Pop-up for Useful Results

One feature of the integration framework is the ability to exploit concurrency. An external system can run automatically in the background, appearing only when it is able to solve a subgoal. Since the initial development, this has turned out to be exceedingly useful, not for the cases we expected, but when we have made a mistake in our proof: several times, we found the QEPCAD integration popping up to tell us that our current goal is false. This was not due to extreme negligence on our part (at least not the majority of times!), but due to the complexity of verifying algorithms, especially when they contain loops. We chose to formally represent our algorithms using Isabelle's development of Floyd-Hoare logic as this allowed the formal specifications of the geometric algorithms to closely resemble their implementations [7]. As a result of using this logic, our verifications relied upon us discovering the correct loop invariants (the facts which hold true on each iteration of the loops) This discovery process was often one of iterative refinement. Thus, by alerting us when a statement was flawed, the external tool accelerated the feedback cycle. This vastly reduced the time which could have been wasted attempting to prove verification conditions with incomplete loop invariants.

We believe this automation technique, performing some computation in the background, will be one of the most important mechanisms by which interactive theorem provers become widely usable. Isabelle now 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 ability to combine multiple automation techniques in a single proving environment, where the automation is effectively invisible except when useful, has been a major assistance to our efforts.

4.2 Expansion and Prenex Normal Form

One requirement of QEPCAD is that the subgoal be expressed in prenex normal form (PNF) without reference to predicates defined elsewhere in the Isabelle theory. To minimise the effort required by the end-user, the Prover's Palette framework offers support for detecting whether these conditions (or others) are met and for automatically generating the appropriate commands for converting a subgoal to the desired form.

When some of a subgoal is detected as incompatible, the QEPCAD integration automatically disables *those parts* of the subgoal, but the integration GUI allows the user to use the remainder of the subgoal with QEPCAD. This is in keeping with the Prover's Palette philosophy that the an external tool integration should be helpful but not overly restrictive. In addition, however, the integration presents the user with a button which will automatically generate, insert and apply the commands to convert the subgoal to the required form where appropriate.

Specifically, if there are definitions that QEPCAD will not understand (*e.g.* `collinear`), the QEPCAD integration GUI gives the user the option to automatically expand them (*e.g.* `apply (simp only: collinear_def)`): the user clicks one button and the subgoal is transformed.

Additionally, where a problem is not in PNF, the integration can insert the correct Isabelle commands into the theory file to convert the problem into PNF. The Prover's Palette can also allow a user to specify that these conversions always be made automatically.

While none of these items is mathematically difficult, they are extremely tedious and this tedium limits the utility of the dependent technique. This burden is removed from the user through simple automation which does the necessary pre-processing,

4.3 Removing Division

Another obstruction to the use of QEPCAD is its inability to reason about division. All statements containing division must be rewritten in terms of multiplication. To simplify this process, we have produced the following set of rewrite rules:

```
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))
```

For the statements encountered in our proofs, these rules are sufficient to remove the division and allow the goals to be sent to QEPCAD. However, they will not work for more intricate statements involving division, such as those containing a sum of fractions. Also, it is worth noting that in Isabelle, dividing by zero equals zero; this is a design choice to ensure that division is total and we have fewer conditional rewrite rules.

5 Translation Invariance

While QEPCAD is theoretically a complete decision procedure, some of the problems 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 these problems is:

```
lemma segExtensionStillIntersects:  "[|X isBetween A B;
        straightEdgesIntersect e {X,B} |] ==>
          straightEdgesIntersect e {A, B}"
```

where `straightEdgesIntersect` is defined as:

```
types edge = "point set"
constdefs straightEdgesIntersect ::  "[edge, edge] => bool"
      "straightEdgesIntersect ea eb ≡
         EX 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. In the QEPCAD integration GUI, we can manually change one pair of (x, y) co-ordinates to be $(0, 0)$. Sending the revised problem in 8 variables, instead of 10, yields a result from QEPCAD 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 [6]. 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 involving a point (x, y), are equivalent to the same proposition translated by $(-x, -y)$; simplification rules then yield the proposition in terms of the origin and one fewer point.

The process is not fully automated, but it is very easy using the Prover's Palette for a user to test whether translation is worth doing, and then fairly quick to perform the translation fully formally. We note that scaling and rotation could be applied in similar ways to remove two further variables, although this has not yet been implemented in our theory.

6 Conclusion

We have, of course, benefitted from an enormous amount of work in automation and semi-automation which it would be impossible to describe exhaustively. What we have tried to do in the course of this paper is describe some of the proof techniques and automation which we have implemented, and which we have found useful, in hopes that these ideas and their implementations may prove of benefit to others. Further details on the Prover's Palette and our empirical results can be found in [10].

Our experience with theorem proving has led us to the conclusion that there is no one "magic bullet" which will make formal verification suddenly easy. Equally, however, our observations and experiences (and our results with verifying geometric algorithms, to be published later this year) leave us convinced that this will be achieved—through the gradual accumulation of theory libraries, automation techniques and proof techniques—shared among the community and subsequently improved upon. We welcome feedback on our approaches described here.

Acknowledgements. We would like to thank the reviewers for their useful comments. This work was funded by the EPSRC grant EP/E005713/1.

References

[1] Aspinall D., C. Lüth, and D. Winterstein, *A Framework for Interactive Proof*, Towards Mechanized Mathematical Assistants, Springer LNAI 4573 (2007), 161-175.

[2] Brown C. W., *QEPCAD B: a program for computing with semi-algebraic sets using CADs*, SIGSAM Bulletin, **37** (2003), 97-108.

[3] Chou S. C., 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, 17:325-3477, 1996.

[4] Chou S. C., 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.

[5] Collins G. E., and H. Hong, *Partial Cylindrical Algebraic Decomposition for Quantifier Elimination*, Journal of Symbolic Computation **12** (1991), 299-328.

[6] Harrison J., *Without Loss of Generality*, TPHOLs, LNCS, v. 5674, pp 43-59, 2009.

[7] Hoare C. A. R., *An axiomatic basis for computer programming*, Communications of the ACM, v.12 n. 10, pp 576-580, 1969.

[8] Janicic P., J. Narboux, and P. Quaresma, *The Area Method : a Recapitulation*, submitted to JAR, 2009.

[9] Meikle L. I., and J. D. Fleuriot, *Combining Isabelle and QEPCAD-B in the Prover's Palette*, AISC/MKM/Calculemus (2008), 315-330.

[10] Meikle L. I., *The Formal Verification of Geometric Algorithms*, to appear as PhD Thesis, University of Edinburgh.

[11] Nipkow T., Paulson L. C., and Wenzel M. *Isabelle HOL: A Proof Assistant for Higher-Order Logic*,
The tutorial can be found at:
`www.cl.cam.ac.uk/research/hvg/Isabelle/dist/Isabelle/doc/tutorial.pdf`