Proof Auditing Formalised Mathematics

Mark Adams
Proof Technologies Ltd, UK, and Radboud University, Nijmegen, The Netherlands

The first three formalisations of major mathematical proofs, all in the last decade, have heralded a new era in formalised mathematics, establishing that informal proofs at the limits of what can be understood by humans can be checked by machine. However, formalisation itself can be subject to error, and yet there is currently no accepted process of checking, or even much concern that such checks have not been performed. In this paper, we explain why formalisation proofs should be checked, and propose rigorous and independent *proof auditing*. We discuss the issues involved in performing an audit, and propose an effective auditing process. We use the Flyspeck project, one of the major formalisations, to illustrate our point, and subject it to a partial audit.

1. INTRODUCTION

In 2005, Georges Gonthier announced that he had successfully completed his project to formally prove the Four-Colour Theorem using a theorem prover [4]. This was a ground-breaking moment in the history of mathematics formalisation, representing the first time a major mathematical proof had been mechanically checked. Within a decade, two more major formalisation projects would be complete, first the Feit-Thompson Lemma,[1] again by Gonthier [5], and then the Kepler Conjecture, by Tom Hales [11].

The original, informal proofs of these theorems are of giant proportion, each running to dozens of pages of mathematical text and, in the case of the Four-Colour Theorem and the Kepler Conjecture, at least a few thousand lines of computer source code. Their formalisations were also major projects, each involving several man years of expert effort and resulting in several tens of thousands of lines of proof script (see Table I). These projects have significantly increased confidence in the original mathematical results.

The informal proofs are each considered important mathematical results in their own right. The Four-Colour Theorem was conjectured in the 19th Century, had a false proof that escaped notice for 11 years, and its eventual first proof in 1976 sparked much philosophical debate [20] for its use of computers, although the controversy died down a little when a shorter version, based on the original and still using computers, was found in 1996. The Feit-Thompson Lemma is a crucial lemma in the classification of finite simple groups, which underpins much of group theory, and its proof from 1962 relies on a large amount of supporting theory. And a solution to the Kepler Conjecture had eluded mathematicians for almost 400 years until its proof in 1998, which, due to its overwhelming complexity, was in the unusual situation of its referees finding no error but holding back from giving their full endorsement. By adding to the confidence in these proofs, the formalisations have themselves become important mathematical artefacts.

[1]Also known as the Odd Order Theorem.

Theorem	Informal Proof			Formal Proof		
	Year	Text (pages)	Program (lines)	Year	Script (lines)	Theorem Prover
Four-Colour	1996	43	2,500	2005	50,000	Coq
Feit-Thompson	1962	250	-	2012	130,000	Coq
Kepler	1998	300	40,000	2014	500,000	HOL Light, Isabelle/HOL

Table I. The three formalisations and the informal proofs they are based upon. Program and script lines refer to the number of non-comment, non-blank lines.

However, despite the importance of these formalisations, and despite their complexity and thus scope for error, they have so far undergone little, if any, independent scrutiny. Certainly their proof scripts are freely available for download (at [31], [32] and [33]) for inquisitive users who wish to replay them through a theorem prover, but watching hundreds of screenfuls of output whizz past a computer screen hardly constitutes a thorough check that a theorem has been formally proved. There are a multitude of things that can go wrong in a formalisation, ranging from simply formalising the wrong theorem to unknowingly exploiting bugs in the theorem prover, and it is a valid question to ask whether a given formalisation project has indeed established its claimed result, rather than simply trust that this is the case.

In this paper, we propose rigorous independent assessment of formalisation proofs by an expert, that is as objective as possible and suitably sceptical, not assuming anything about the reputation of the team carrying out the formalisation, and taking into account the weaknesses of the theorem prover(s) being used. We call this activity *proof auditing*. We illustrate the issues involved by looking in some detail at the Flyspeck project, that formalises the Kepler Conjecture proof.

In Section 2, we give some background about theorem provers and the Flyspeck project. In Section 3, we classify the reasons why a sceptic might doubt the claimed results of a formalisation project, and cover some concerns with specific theorem provers. In Section 4, we discuss the issues involved in proof auditing, suggest an efficient process for performing an audit and look at potential tool support. In Section 5, we look at how Flyspeck could be audited, and carry out a partial audit using our proposed process. In Section 6, we discuss related work. In Section 7, we present our conclusions.

2. BACKGROUND

2.1 Theorem Provers used in Formalisation

A *theorem prover*[2] is a tool for performing mechanised formal proof, rigidly adhering to the deductive system of a given formal logic. There are dozens of contemporary theorem provers,[3] but only four are prominent in the formalisation of mathematics. HOL Light [12] and Isabelle/HOL [22] are two of various members of the HOL family of theorem provers that implement the HOL logic, a classical higher-order logic with a simple polymorphic type system. Coq [3] implements the Calculus of

[2]Also known as a *proof assistant*.
[3]For two excellent comparisons of the most widely used, see [27] and [28].

Constructions, an intuitionistic higher-order logic with a dependent type system. Mizar [25] implements Jaskowski-style natural deduction, a classical first-order logic with set theory. Unlike the other systems, rather than having its deductive system "hardwired" as source code, Isabelle/HOL is an instantiation of the Isabelle generic theorem prover [26], which provides a logical framework within which the primitive inference rules for HOL are encoded as meta-level axioms. HOL Light, Isabelle and Coq are all implemented in dialects of the ML programming language, whereas Mizar is implemented in Pascal.

Implementing a sound theorem prover is notoriously susceptible to error, and virtually all theorem provers, including the above four, have suffered from logical unsoundness in a previous version. There are various approaches to avoiding unsoundness, but the LCF approach [6] has become predominant over the years, and is used in HOL Light, Isabelle and Coq. This involves using a software architecture based around an *inference kernel* of primitive inference rules and theory extension commands. The programming language's type system is used to ensure that theorems can only be constructed via this kernel, through use of strongly enforced abstract datatypes. Thus all routines implemented outside the kernel have a reassuring soundness "safety net" that they cannot introduce unsoundness to the system. Another aspect of the LCF approach is that the user can extend the system with their own source code defining new proof commands, using the system's implementation language, and that these extensions also enjoy the soundness safety net, although this user extendibility can be restricted by the system's user interface.

The *logical core* of a theorem prover is the implementation of its formal logic, which in LCF-style systems consists of the inference kernel plus code defining the logic's initial theory. Regardless of the software architecture used, an error in the logical core can potentially give rise to unsoundness. We use the term *core system* to mean the logical core plus other basic functionality such as a parser and a pretty printer for expressions in the logic's formal language. In LCF-style systems, the core system encompasses the trusted source code.

All four of the above theorem provers are essentially interactive, rather than automated, systems, where a user guides a proof by issuing a series of proof commands in a *proof session* (although these commands are often submitted in batch mode). In such systems, there are typically some proof commands that correspond to basic inferences of the logic,[4] and some that are high-level instructions that get unravelled by the theorem prover into a series of basic inferences, often hundreds or even thousands of basic inferences long. A *proof script* is the series of proof commands used to perform a formal proof. Note that for user-extendible LCF-style systems, proof scripts may include their own user extensions, and we count all of this as proof script of a project, even project-specific supporting script files that only define proof commands rather than actually formally prove anything.

A *proof object* captures the basic inferences actually used in a formal proof, so that they could feasibly be imported and replayed in a different system. A *proof checker* is a tool for checking that the inference steps performed in a proof object correctly adhere to a formal logic's deductive system. A theorem prover could act

[4]By *basic inference* we mean either a primitive inference or the composition of a few primitive inferences.

$$\vdash A/B \leq \pi/\sqrt{18}$$

where A is the volume occupied by a packing of same-sized spheres within a containing sphere of volume B, as the radius of B tends to infinity.

Fig. 1. The informal statement of the Kepler Conjecture.

as a proof checker, but the term is usually used to mean a simple tool dedicated purely to checking proof objects rather than also being able to interactively or automatically create formal proofs.

2.2 The Flyspeck Project

2.2.1 *History.* The Kepler Conjecture (see Figure 1) states that there is no packing of same-sized spheres in infinite three-dimensional space denser than the face-centred cubic packing, i.e. the pyramid-shaped arrangement commonly used by grocers to stack oranges, which has density $\pi/\sqrt{18}$ (approximately 74.0%). This was first posited by Johannes Kepler in 1611, and was one of the longest unsolved problems in mathematics until 1998, when Tom Hales announced his completed proof and submitted it for publication.

The text of Hales' original proof [7] was around 300 pages long and used the results of three bespoke computer programs written in C++, Java and Mathematica totalling a few tens of thousands of lines of computer source code. The first program generated and classified an exhaustive list of planar graphs satisfying a condition that Hales called *tameness*, the second solved linear inequalities, and the third solved non-linear inequalities.

However, the proof became controversial after its referees, exhausted after five years of reviewing its mass of detail, found no error but could only say they were "99% certain" of its correctness, although they agreed to publish. This was most unusual because, of course, referees of mathematical journals only normally publish proofs that they are 100% certain about.

In response to this state of limbo, Hales instigated the Flyspeck project [8] in 2003, to formalise his proof. His motivation was not only to remove any doubt about his own proof. He had also become an enthusiastic proponent of the QED agenda [1], seeing formal proof as being "fundamental to the long-term growth of mathematics", given the increasing difficulty in refereeing ever longer and more complicated proofs, and saw Flyspeck as demonstrating its feasibility.

Hales subsequently considerably revised his informal proof [9] since its first publication. Better techniques were used that simplify the overall proof, explanation and transparency were improved in the text, and the computer programs were adjusted and/or reimplemented. These changes improved the proof itself, but the main motivation was to make it more amenable to formalisation. Included in the changes were the use of hypermaps to replace planar graphs, and the use of a different geometric partition of space. The changes also included corrections to various small errors that existed in the original proof (that had no overall effect on the proof's correctness) that were found during the process of formalisation. It is almost inevitable that formalisation of a large and complex proof roots out various errors of this kind. The resulting updated main text of the proof is now available as [10].

Hales used an international team of mathematicians and computer scientists to work on Flyspeck. Considerable work on formalising the linear and non-linear inequalities results was done in Isabelle/HOL and Coq respectively, but both ended up being reworked and completed in HOL Light for the final proof. The classification of tame planar graphs was undertaken in Isabelle/HOL. However, the bulk of the Flyspeck effort was concerned with formalising the main mathematical text, which was done in HOL Light. It was broken down into around 700 lemmas, each with a cash bounty attached that was awarded on completion of its formal proof. Hales was willing to consider contributions from anyone interested in taking part, and assigned lemmas upon request. Contributors submitted their completed formal proofs as HOL Light proof script files, which were rerun by Hales before being incorporated into the project repository. Around 10 contributors succeeded in having their work incorporated. By using this system of bounty payments, Hales was effectively outsourcing the main text formalisation and reducing the risk of overspending.

The Flyspeck project was completed in 2014. It had consumed around 20 man years of effort, which was remarkably the same as Hales had predicted in 2003.

2.2.2 *Overview of the Formal Proof.* The Flyspeck formal proof breaks down into four parts, corresponding to the four parts of the informal proof: one for the main text, and one for each of the three computer programs. We provide a brief overview here. A more extensive overview can be found in [11]. The entire project, including the informal and formal versions of the proof, is downloadable as an SVN repository [33]. The project files we refer to in this paper are all from the `text_formalization` directory of the repository root directory.

The formalisation of the main text builds on HOL Light's standard theory and Multivariate libraries. Its 700 lemmas and final theorem are formally proved in around 280 ML proof scripts (including supporting scripts), with a total of around 450,000 non-comment/blank lines of ML, resulting in about 1.3 billion HOL Light primitive inference steps.[5] Together with the formalisation of the linear inequalities, it executes in around 5 hours of processing.[6] Within the proof scripts, anything was acceptable for it to be incorporated into the project, so long as no new axioms[7] were added and a pre-arranged ML identifier was assigned the theorem stating the pre-arranged lemma result. See Figure 2 for a typical fragment of proof script. Various supporting scripts defined project-specific ML utility functions or bespoke tactics, such as in Figure 3, and many proof scripts freely interspersed such definitions amongst actual tactic proofs.

The program for generating tame graphs is formalised using Isabelle/HOL's program extraction capability. This involves expressing the specification of the program (i.e. that it produces an exhaustive list of tame graphs) as a statement in HOL, transforming this over a series of steps into an executable specification, justifying

[5]This total includes around 8 million steps for the standard theory library, and around 175 million steps for the Mulitvariate library.
[6]This time is from Hales [11], using a 2.0 GHz CPU, and includes the time to process the standard theory and Multivariate libraries.
[7]In this paper, by *axiom* we mean theory extension by a general extension command that does not enforce conservative extension.

```
let MUL_POW2 = REAL_ARITH‘ (a*b) pow 2 = a pow 2 * b pow 2 ‘;;

let x = Some(0, ‘x pow 4 = x pow 2‘) ;;

let COMPUTE_SIN_DIVH_POW2 = prove(‘! (v0: real^N) va vb vc.
let betaa = dihV v0 vc va vb in
let a = arcV v0 vc vb in
let b = arcV v0 vc va in
let c = arcV v0 va vb in
let p =
&1 - cos a pow 2 - cos b pow 2 - cos c pow 2 +
&2 * cos a * cos b * cos c in
~collinear {v0, vc, va} /\ ~collinear {v0, vc, vb} ==>
( sin betaa ) pow 2 = p / ((sin a * sin b) pow 4) ‘,

REPEAT STRIP_TAC THEN MP_TAC (SPEC_ALL RLXWSTK ) THEN
REPEAT LET_TAC THEN SIMP_TAC[SIN_POW2_EQ_1_SUB_COS_POW2 ] THEN
REPEAT STRIP_TAC THEN REPLICATE_TAC 2 (FIRST_X_ASSUM MP_TAC) THEN
NHANH (NOT_COLLINEAR_IMP_NOT_SIN0) THEN
EXPAND_TAC "a" THEN EXPAND_TAC "b" THEN PHA THEN
SIMP_TAC[REAL_FIELD‘ ~( a = &0 ) /\ ~ ( b = &0 ) ==>
&1 - ( x / ( a * b )) pow 2 = (( a * b ) pow 2 - x pow 2 ) / (( a * b ) pow 2 )‘;
eval "x"] THEN
ASM_SIMP_TAC[] THEN STRIP_TAC THEN
MATCH_MP_TAC (MESON[]‘ a = b ==> a / x = b / x ‘) THEN
EXPAND_TAC "p" THEN SIMP_TAC[MUL_POW2; SIN_POW2_EQ_1_SUB_COS_POW2] THEN
REAL_ARITH_TAC);;
```

Fig. 2. An extract from a typical Flyspeck proof script, `trigonometry/trig2.hl`.

```
let GMATCH_SIMP_TAC thm gl =
  let w = goal_concl gl in
  let lift_eq_thm =
          MESON[] ‘! a b c. (a ==> ((b:B) = c)) ==> (!P. a /\ P c ==> P b)‘ in
  let lift_eq t = GEN_ALL (MATCH_MP lift_eq_thm (SPEC_ALL t)) in
  let thm’ = hd (mk_rewrites true thm []) in
  let t1 = fst (dest_eq(snd (dest_imp(concl(thm’))))) in
  let matcher u t =
    let m = term_match [] t1 t in
    let _ = subset (frees t) (frees u) or failwith "" in
      m in
  let w’ = find_term (can (matcher w)) w in
  let var1 = mk_var("v",type_of w’) in
  let vv = variant (frees w) var1 in
  let athm = REWRITE_CONV[ ASSUME (mk_eq (w’,vv))] w in
  let bthm = (ISPECL [mk_abs(vv,rhs (concl athm));w’] BETA_THM) in
  let betx = SYM(TRANS bthm (BETA_CONV (rhs (concl bthm)))) in
   (ONCE_REWRITE_TAC[betx] THEN MATCH_MP_TAC (lift_eq thm’) THEN
      BETA_TAC THEN REWRITE_TAC[]) gl;;
```

Fig. 3. A bespoke Flyspeck tactic from `general/hales_tactic.hl`.

```
|- import_tame_classification /\ the_nonlinear_inequalities
   ==> the_kepler_conjecture
```

Fig. 4. The final theorem of the main text and linear inequalities.

the transformation by proving its correctness condition using Isabelle/HOL, and then extracting the executable specification as ML source code. This ML source code is then executed to produce a text file, referred to as the *archive*, detailing an exhaustive list of around 20,000 tame graphs. The archive is used by HOL Light to generate the linear inequalities.

The program for proving the linear inequalities is not itself formalised. Instead the results it proves are formalised in HOL Light using a bespoke automatic proof procedure. This is implemented in a few thousand lines of ML. It executes in the same HOL Light session as the main text formalisation.

The program for proving the non-linear inequalities, again, is not formalised but has the results it proves formalised in HOL Light using a bespoke automatic proof procedure. This is implemented in 25,000 lines of ML. There are around 23,000 non-linear inequalities, which it proves in around 5,000 hours of processing, spread across approximately 600 parallel HOL Light sessions.

The separate parts of the project are brought together in the final build file, `general/the_kepler_conjecture.hl`. The main text final theorem and the linear inequalties, which execute in the same HOL Light session, are used to prove a final theorem for the combination of these two parts, stating the Kepler Conjecture assuming the classification of tame graphs and the non-linear inequalities (see Figure 4). These two antecedents are not actually discharged, but the non-linear inequalities are effectively assumed as a single axiom by a routine that rigorously checks that these have been proved using theory contexts that are not conflicting with each other or the main HOL Light session. Thus the Kepler Conjecture is formally proved, albeit assuming results established by around 600 separate HOL Light sessions and an ML program generated in an Isabelle/HOL session.

3. REASONS TO BE DOUBTFUL

In this section, we discuss concerns that a sceptic might reasonably hold about the correctness of a claimed formalisation proof. We first classify these concerns, and then examine concerns related to specific theorem provers.

3.1 Classification of Concerns

We consider here only issues we view as standing a realistic chance of causing problems in large formalisation projects employing contemporary theorem provers, not assuming anything about how a project is organised or the people involved. Note that we further discuss issues in Section 4.1.

3.1.1 *Unreproducible Results.* The source components of the project might fail to combine to prove the final formal theorem when processed. This may be because processing fails, perhaps due to software version control issues, or due to the final version of the project not being tested to iron out last-minute changes. Or it may be because, even though processing completes, the claimed result is nowhere to be seen. How do we know that the final theorem was actually proved?

3.1.2 *The Wrong Formal Statement.* The formal statement of the final theorem might not accurately capture the intended meaning of the statement of the informal theorem. This may be due to a subtle problem in the way the statement has been formalised, or similarly in the definition of one of the constants used in the formal statement, or in the definition of one of the constants used in one of the definitions. In large projects, there will be a dependency graph of definitions used in the formal statement, and furthermore these may rely on substantial supporting theory with its own definitions. A problem in any of these definitions will mean that the project is not proving what it is claiming to prove. How do we know the right final theorem was proved?

3.1.3 *Theory Inconsistency.* Axioms might have been added to the theorem prover's theory that render the logic inconsistent, thus invalidating the formalisation. This may be due to an axiom contradicting an inference rule, a definition or another axiom. Such inconsistency may be extremely subtle. How do we know that the theory has not become inconsistent?

3.1.4 *Unsound Inference.* The theorem prover might be susceptible to making unsound inferences. This may be because its logical core has been incorrectly implemented. For example, a subtle error in the implementation of variable substitution could mean that a primitive inference rule is unsound. It may also be because its software architecture fails to prevent unsound extensions to its logical core. For example, if the statement of a theorem can be altered after it is proved, or if a "trojan horse" is installed to replace an original trusted component of the system. How do we know that the inferences performed in the formal proof were sound?

3.1.5 *Misinterpreted Display of Formulae.* The final theorem, or its supporting definitions, might be displayed by the system's pretty printer in a way that gets misinterpreted. Some theorem provers are susceptible to ambiguous display of formulae, as in Pollack-inconsistency [29]. For example, irregular or overloaded names may be displayed in a way that naturally gets interpretted as meaning something quite different. Also, theorem provers have display settings that can alter the interpretation of what is being displayed, and unconventional settings may have been used. For example, the precedence and associativity of infix operators may be configurable, which can affect how a formula involving a mix of infix operators gets interpreted. Also, a theorem prover's software architecture may allow uncontrolled adaption to the way formulae are displayed, for example allowing an alternative pretty printer to be installed. How can we know the true syntax of the results that get displayed?

3.1.6 *Multiple Sessions.* The formalisation project may be split across more than one theorem prover session, or may even be split across more than one theorem prover (as in Flyspeck). If this is the case, then it is of crucial importance that the separate parts fit together in a logically coherent way so that the final theorem is correctly deduced. Theorems proved in one session might have been established in an incompatible theory context to those proved in another session of the same

```
# let t = fst (dest_const (concl TRUTH)) in
  let () = t.[0] <- 'F' in
  let FALSE = EQ_MP (REFL `F`) TRUTH in
  let () = t.[0] <- 'T' in
  FALSE;;
val it : thm = |- F
```

Fig. 5. Exploiting OCaml string mutability to prove false in HOL Light without leaving a trace.

theorem prover, for example if a constant is given a different definition in each session. When bringing together results from different theorem provers, the risks are greater still. Incompatible theory context is much more likely, and there is the additional risk of incorrect translation between formal languages or intermediate notations used to pass results, whether performed by hand or by machine. If the theorem provers implement different formal logics, then these risks are even greater. How do we know that a proof spread across multiple sessions is logically equivalent to the proof done in a single session?

3.2 Theorem Prover Concerns

Here we discuss trustworthiness-related issues with specific theorem provers. It is perhaps surprising to those outside the field of theorem proving that most theorem provers, including those prominent in the formalisation of mathematics, have known trustworthiness issues that are allowed to persist. This is not to say, of course, that large proofs performed in these systems are necessarily wrong, but it does raise concerns. We look at the four main formalisation theorem provers, and put the spotlight on HOL Light in particular because it is the main system used in the Flyspeck project, the focus of this article.

3.2.1 *HOL Light.* The HOL Light system[8] has an LCF-style architecture, a core system of just 2,300 lines[9] of OCaml code, including a simple and well-studied logical core of only 900 lines,[10] and implements one of the most simple and widely-understood formal logics, HOL. There has also been a formal verification of the correct implementation of its deductive system [13]. In these respects, HOL Light is very highly regarded for its trustworthiness.

However, there are various known trust-related concerns with the system. Firstly, although it captures all axioms and constant definitions, it does not capture type definitions in its state. This means that it is not possible to perform a simple query of the system state to find the definition of a type constant, and instead examination of the proof script must be used.

Secondly, HOL Light's software architecture has vulnerabilities. Most significantly, its LCF-style kernel is not completely watertight: it is possible to process proof script that will result in unsound deduction. This is due to HOL Light not addressing dangerous aspects of its implementation language, a dialect of ML called

[8]Our observations relate to recent versions of HOL Light, including SVN revisions 197 and 210.
[9]In this paper, when counting lines of source code, we count non-blank, non-comment lines, and include supporting bespoke library code but exclude all module interfaces.
[10]We consider the HOL Light logical core to consist of the build files up to `fusion.ml`, and the core system to extend to `parser.ml` but excluding `nets.ml`.

```
# let v1 = mk_var ("x",`:num`) in
  let v2 = mk_var ("!y. y",`:num`) in
  let tm = mk_exists (v1, mk_eq (v2,v1)) in
  EXISTS (tm,v2) (REFL v2);;
val it : thm = |- ?x. !y. y = x
```

Fig. 6. Exploiting irregular variable names in a misleadingly displayed theorem in HOL Light.

OCaml, which is also the language its proof scripts are written in. One vulnerability is that HOL Light does not protect against OCaml's mutable strings, and so the name of a HOL constant can be altered by the user simply by altering the string storing the name (see Figure 5). Another is that OCaml has an (undocumented) function called `Obj.magic`, which subverts the OCaml type system to return its argument typecasted according to type context, and so can be used to bypass the LCF-style inference kernel and create arbitrary theorems.

Another architecture-related vulnerability is that there is nothing to stop the HOL Light user overwriting the inference kernel with a "trojan horse" one, with a new theorem datatype that has the same name as the real theorem datatype but allowing arbitrary theorems to be constructed. Similarly there is nothing to stop the pretty printer being overwritten with a bogus one, displaying theorems unfaithfully. These trojan horse problems are essentially an unavoidable consequence of the user-extendible LCF-style architecture without explicit support from the implementation language to prevent overwriting of key components.

Thirdly, there are various flaws in the display of HOL concrete syntax that make it ambiguous and potentially misleading for the user. One such flaw is that type annotation is never used, so that, for example, a theorem may appear to be universally true for variables of any type when it has actually only been proved for variables of a specific type (e.g. a type with just one element). Another flaw is in the display of irregular or overloaded names, so that, for example, a variable with a name involving space characters may give the impression that a formula has a completely different syntactic form from its true form (see Figure 6). Wiedijk [29] provides further examples.

3.2.2 *Isabelle.*

The trustworthiness credentials of Isabelle[11] lie in its LCF-style architecture, the relative simplicity of its meta logic, its relatively wide user base and its use of the SML dialect of ML as its implementation language. SML is more precisely defined than OCaml and does not have dangerous features such as arbitrary typecasting or mutable strings.

The main concern for Isabelle arises from the richness of its features, which include: constant overloading, axiomatic type classes, multi-threading, the Isar language with embedded ML and a jEdit IDE. All these features are supported in the Isabelle core system, which has a relatively complicated implementation in perhaps around 18,000 lines of SML code.[12] There are very few people that fully understand the implementation of this core system.

[11]Our observations relate to Isabelle2014.

[12]We consider the Isabelle core system to approximately consist of the build files in its `Pure` directory up to `thm.ML`, but find it difficult to determine its or the logical core's precise extent.

Kuncar [19] discovered logical inconsistency in Isabelle2013-2's treatment of constant overloading, which itself incorporated an adaption in attempt to correct a previous inconsistency in Isabelle2005. This has now been corrected in Isabelle2014, but a nagging question for the inquisitive mind is what other, unknown unsoundnesses are lurking in the core system or will get introduced as more features are added.

In addition, there are known trustworthiness issues with Isabelle's display of formulae. These are similar to some of the problems suffered by HOL Light, and concern issues such as irregular or overloaded names. Wiedijk [29] provides some examples.

3.2.3 *Coq.* The Coq system[13] has the safety net of an LCF-style inference kernel, and has a relatively wide user base (making the discovery of any existing soundness flaws more likely). Furthermore, it is able to output proof objects that can be replayed by a separate program called Coqchk (see Section 4.5.6) to check that the inferences performed are correct.

Coq's logical core implements a logic that is considerably more complex than HOL, and as part of this performs various automated tasks, including reflective proof, normalisation of formulae, establishing termination for recursive function definitions and checking correct universe chain ordering. Its core system is implemented in around 45,000 lines of OCaml and 1,600 lines of C code, including a logical core of around 11,500 lines of OCaml and 1,600 lines of C.[14] The chief concern for Coq is that its relatively complicated logical core, which is fully understood by very few people, may have unknown soundness flaws lurking.

Logical inconsistencies have been uncovered in Coq 8.4 in recent years, including in its treatment of types with more than 255 constructors [34], termination analysis [35], De Bruijn indices used in the representation of terms [36] and coinductive fixed points [37]. These soundness flaws don't always make it to public releases, and are usually addressed within a few weeks, but the frequency of their discovery is unnerving for such a mature system.

Coq is implemented in OCaml, and there is nothing to prevent user-programmed LCF-style extensions from exploiting mutable strings and arbitrary typecasting (see Section 3.2.1), although in normal usage proof scripts are written in Coq's dedicated language Ltac, rather than ML, and cannot exploit these vulnerabilities other than by calling user-programmed commands that do. Coq's display of formulae also suffers from ambiguity in various scenarios, and again Wiedijk [29] provides examples.

3.2.4 *Mizar.* Unlike the other three main formalisation systems, the source code of Mizar is relatively closed, only being available to members of the Association of Mizar Users. This opaqueness makes it more difficult for outsiders to understand the implementation of Mizar and discover new problems, making it more likely that undiscovered problems exist. A few unsoundnesses have been uncovered over the

[13]Our observations relate to Coq 8.4.

[14]We consider the Coq logical core to consist of the files from the `kernel` build directory plus those files it uses from the `lib` directory, and the core system to approximately consist of the files from the directories `lib`, `kernel`, `library`, `interp`, `parsing` and `pretyping`.

years, including one in Mizar 6.3 in 2003 [38]. Also, the display of formulae suffers from similar problems to the other systems, as shown by Wiedijk [29].

3.3 Puzzle

Now it is time for us to come clean. We have doctored the example code fragment in Figure 2 to exploit one of the HOL Light trustworthiness issues mentioned in Section 3.2.1. The statement of theorem `COMPUTE_SIN_DIVH_POW2` has been changed to something that is not true, but has been proved in HOL Light using the proof script we show. This is potentially catastrophic for the project, since this false theorem could be used to obtain an invalid formal proof of the final theorem. We challenge readers familiar with HOL Light to solve the puzzle of how we managed to prove the false statement. Our hint is that it is sufficient to consider only the code fragment shown and the effect of processing the (undoctored) project build initialisation file, `strictbuild.hl`. We supply the solution in Section 5.3.

4. PROOF AUDITING

We propose the activity of *proof auditing* to independently assess a formalisation and convincingly address the sceptic's concerns elaborated in Section 3. An audit should aim to establish a *correctness case* for the proof, in much the same way as a safety case is established for the certification of safety-critical equipment in engineering. This would be a robust argument that a complete and correct formal proof of the original informal theorem had been performed, backed up with strong evidence showing that the risks had been sufficiently mitigated. The correctness case could be suggested by the formalisation team, or created by the auditor.

In the rest of this section, we discuss the various issues involved in proof auditing, suggest an effective process for this activity and consider potential tool support.

4.1 Philosophical Issues

We first discuss some philosophical points about proof auditing.

4.1.1 *Small Errors.* Note that the ultimate question is whether a formal proof has been performed, not whether the formalisation team's formal proof is correct in every last detail. It is sufficient if the auditor can establish their own formal proof by filling in the gaps and correcting small mistakes in the original. In the same way that the original formal proof may involve various small corrections to the original informal proof, the auditor should be aiming at establishing a robust correctness case, if this is within easy reach, rather than rejecting the formalisation for a small problem that can be easily fixed.

4.1.2 *Simplicity.* The auditor should aim to make the correctness case as simple as possible, relying on as few assumptions as possible. Straightforward arguments are more convincing than complicated ones. If it is possible, it is better to avoid reliance on a flawed or untrusted software tool altogether, for example, than to produce complex arguments that attempt to justify why it has been used in a safe way.

4.1.3 *Innocent Error vs Malicious Intent.* The auditor should not assume the competence or good intentions of the project team. Teams are composed of human beings who make mistakes, and who are not always motivated purely by the desire to act in a conscientious and truthful manner. If an unscrupulous team member knew about a back door to creating theorems, perhaps they would be tempted to subtly exploit this to boost their recognition or pay. Or perhaps the team manager might be tempted to exploit a flaw in order to get the project completed on time or on budget. The risk is greater still if parts of the project are outsourced, as in Flyspeck, which is a trend that is likely to grow in the future. In their review, the auditor should pessimistically assume malicious intent, rather than use arguments about the improbability of innocent error.

4.1.4 *The Dangers of LCF-Style Extensions.* It should be noted that use of a user-extendible LCF-style architecture, although generally greatly aiding the trustworthiness of formal proofs, carries with it its own risks when used with systems that suffer from trustworthiness flaws, however minor or obscure those flaws might seem. Given that users are allowed to program their own extensions, a poorly-programmed automatic proof routine might unwittingly make an obscure flaw much more likely to occur than in interactive proof. For example, the routine might accidentally generate variable names that have irregular lexical syntax or that are overloaded with constant names, and such variable names could make it through to the statement of the final theorem or of one of its dependent definitions and result in confusingly displayed formulae. Or, for example, the routine may use clumsy string manipulation and unknowingly exploit the string mutability problem in HOL Light (see Section 3.2.1). Arguments based on the improbability of innocent error exploiting known flaws are weakened further in the context of thousands of lines of LCF-style user extensions of unknown quality.

4.1.5 *Inclusive vs Exclusive Auditing.* The auditor should not be placing unnecessary restrictions on how the formalisation project is done in order to make their job easier. A theorem prover that happens to suffer from some trustworthiness flaws may be the best system to use as the project theorem prover, and just because it has flaws does not mean these will necessarily get exploited. Likewise, techniques such as LCF-style user extension can be extremely effective (it is no coincidence that all three major formalisation projects made heavy use of the technique), and the associated dangers might not materialise in practice. Formalisation of large proofs is an extremely challenging process, and the project team should be allowed to use the tools and techniques that they feel appropriate, so long as auditing is feasible.

4.1.6 *Lesser Concerns.* Of lesser concern than the issues discussed in Section 3 are various small risks that we view as extremely unlikely to cause a problem in practice, even on large projects. However, if these concerns can be addressed by an auditing process for little cost, then all the better.

Such lesser concerns include the risk of error in the supporting software and hardware stack[15] affecting the correctness of the formal proof. For example, an error in the programming language's type system might compromise the architecture of an

[15]The supporting software/hardware stack is the software and hardware relied upon by the project theorem prover, such as the theorem prover's programming language, the computer's operating system and the computer's memory chips and processor.

LCF-style theorem prover, or the operating system may overwrite the area of RAM used by the theorem prover in a way that doesn't crash the system and happens to allow an unsound deduction to take place. Component failure in hardware may also happen in a way that looks like the hardware is still functioning normally but that appears to allow unsound deduction, for example in a pixel on the display screen, or in the RAM or CPU chips. Malicious external influence may conceivably disrupt computation and appear to allow unsoundness, for example due to a computer virus infecting the software stack, or the display signal being intercepted and adjusted.

4.1.7 *Mitigation through Diversity.* It is valid for the auditor to use arguments about diversity to help mitigate concerns. For example, the lesser concerns in Section 4.1.6 could be effectively mitigated by running the formal proof using different variants of the theorem prover's implementation language, different operating systems and different machines. However, the auditor must be careful not to rely too heavily on such arguments when true diversity does not exist. For example, performing the formal proof on two different theorem provers helps mitigate the risk of unknown flaws in the theorem prover, but the benefit is somewhat reduced if both were implemented by the same team or if both shared some of the same critical source code.

4.2 Practical Issues

We now discuss various practical issues that should be taken into consideration when designing a proof auditing process.

4.2.1 *Efficiency.* It is highly desirable that the proof auditing process is fairly quick and easy to carry out. A process that requires years of computing time or months of man power when applied to major projects is much less likely to ever get used on such projects, and yet these are the very projects that most need auditing.

4.2.2 *Concrete Syntax vs Primitive Syntax.* For theorem provers used in non-trivial applications, the ability of the pretty printer to display formal language at a concrete syntax level is a practical necessity. Expressions would otherwise become almost unreadable due to excessive type annotation, bracketing and lack of everyday shorthand (consider the small example in Figure 7, and how much worse it could get for expressions spanning many lines). Similarly, it is highly desirable for the auditor to be able to view formulae displayed in concrete syntax, so that they can concentrate on whether the formulae are correct rather than being distracted by interpreting their structure. The audit would be greatly slowed down and carry its own risk of error if primitive syntax or, worse still, syntax destructor functions were required to view formulae, whether this was because the system being used did not support concrete syntax or because its display of concrete syntax could not be trusted.

4.2.3 *Session Final State vs Proof Script.* As can be seen from the formalisations in Table I (see Section 1), a project's proof scripts can run into hundreds of thousands of lines. Any problems with the formalisation, if they exist in the project, will ultimately be evident by considering just these scripts together with the project theorem prover. However, establishing the absence of problems by trawling through every line of every proof script would be tortuously slow and tedious

```
`!(x:num) y. x < y ==> x <= y`

`((!):(num->bool)->bool)
    (\(x:num).
       (((!):(num->bool)->bool)
          (\(y:num). (((==>):bool->bool->bool)
                        ((<) (x:num) (y:num))
                        ((<=) (x:num) (y:num))))))`
```

Fig. 7. Concrete and primitive syntax for the same expression in HOL Light.

for large projects. This is especially the case if user-extendible LCF-style systems are being used, where free-form ML can obscure what a proof script is really doing.

It is clearly preferable if the absence of problems can be established by examining the theorem prover's final state instead. For example, to determine whether any axioms have been added, it is clearly better to print a complete list of axioms from the final state, rather than look for lines in the proof scripts that directly or indirectly result in an axiom being added. Another advantage of examining the final state is that the auditor avoids having to trust the system's parser to faithfully read in syntax from the proof scripts, and instead only relies on its pretty printer to faithfully and unambiguously display syntax.

However, obviously just querying the final state does not work for information that is not captured in the state, or, to be precise, not reliably captured in the state. Neither does it work if the theorem prover has trustworthiness flaws. For example, the code for exploiting string mutability in HOL Light (see Figure 5 from Section 3.2.1) could be embedded anywhere in the proof scripts, or worse still obfuscated and spread over several proof scripts, and querying the HOL Light final state would not shed any light on this.

4.2.4 *Proof Porting.* It is not necessary to limit the audit to using the project theorem prover, or to using it in the same way as it was used in the project. An alternative approach is to use *proof porting* to recreate the formal proof in a separate *target session* of a target system. This involves using an export-adapted version of the project theorem prover to record the formal proof steps performed as the proof scripts are processed and export these to disk as proof object files, and then using an import-adapted target system to read in these proof objects and replay the proof steps in the target session. The target system can be the same system as the project theorem prover, or can be a different theorem prover, or a proof checker. If it is a different system, it would normally be dedicated to the same formal logic as the project theorem prover, since porting to a different logic involves significant extra challenges to accurately capture the original proof without an explosion in proof object size.

There are various reasons why proof porting can be useful for proof auditing. Firstly, any flaws or shortcomings in the project theorem prover, including any unknown flaws, can be avoided by porting to a system that is trusted not to suffer from such problems. So, for example, proofs performed in a system that does not capture all required information in its state can be ported to a system that does, to enable the audit to be performed by examining final state rather than the proof scripts (see Section 4.2.3). And, for example, concerns about a system's logical soundness can be addressed by porting the proof to a system that is trusted to be sound.

Secondly, porting to a different system gives diversity that helps mitigate unknown errors in the project theorem prover, as well as the "lesser concerns" (see Section 4.1.6), even if the target system itself is not trusted.

Thirdly, the dangers of LCF-style user extensions for theorem provers with flaws (see Section 4.1.4) can be avoided if proof objects, rather than proof scripts, are used to recreate the formal proof without extending the theorem prover with user source code. This is true even if the project theorem prover is used as the target system.

Fourthly, the formal proof may not have been originally proved in one session, or even in the same theorem prover, but with proof porting the various sessions can be consolidated into a single coherent target session.

Note that it is of significant advantage if the target system can display information about what it has replayed, such as the final theorem and any extensions made to the theory. This removes the need to trust the proof exporting mechanism, which may otherwise be subject to the risk of exporting a correct proof but of the wrong theorem. It is of little use for the auditor to be informed that the exported proof object successfully replayed on the target system, if they do not know what was replayed or in what context.

4.3 Suggested Proof Auditing Process

We now outline our suggested process for proof auditing. We presuppose the existence of a proof porting capability that works for massive formal proofs, and a highly trustworthy target system that can be relied upon to perform sound deduction, to faithfully and unambiguously display concrete syntax and to capture all relevant information in state. We describe an ideal capability in more detail in Section 4.4.

The process breaks down into three main stages as follows:

(1) Replay the original project using the project theorem prover(s):
   (a) Run each of the sessions of the project's formal proof;
   (b) Identify the final theorem.
(2) Port the project to a trusted target system:
   (a) Use proof porting software to rerun each session and export proof objects;
   (b) Consolidate the proof objects into a single session of a trusted target.
(3) Examine the final state of the target system:
   (a) Examine the display settings;
   (b) Examine the list of axioms;
   (c) Review the statement of the final theorem, and its dependency graph of supporting definitions.

By porting the proof to a trusted target system, the concerns enumerated in Section 3.1 can all be addressed. Replaying the proof objects in the target system establishes that the final theorem is formally proved (see 3.1.1). The target system can be trusted to reliably hold in its state complete and queriable lists of axioms, definitions and display settings, allowing the formal statement of the final theorem

(3.1.2) and the axioms (3.1.3) to be reviewed without concern that information is missing or wrongly recorded, and the system's display of formulae in these reviews can be trusted so that it is not liable to being misinterpreted (3.1.5). The target system can also be relied upon not to perform unsound inference in deducing the final theorem (3.1.4). Finally, if the original formal proof is proved over numerous sessions, these can all be imported into a single session in the target system to check the coherence of the separate sessions (3.1.6).

As well as addressing the concerns, the process is also relatively quick and easy to perform. Porting to a trusted target system means that it is only necessary to examine the final state of the target system, rather than to examine the project proof scripts. This avoids the painful process of wading through tens (or even hundreds) of thousands of lines of proof script, trying to work out whether a subtle problem exists that invalidates the entire project. Examining proof scripts would be particularly problematic for projects split over multiple sessions, where the coherence of the sessions and their interfacing would need to be established. Given a proof that replays in the trusted target system, the auditor need not be concerned with the contents of the proof scripts, the correctness of the project theorem prover, the correctness of the proof porting capability or the contents of the proof object files. All that matters is the resulting target session state.

To gain maximum assurance from an audit, the formal proof should be ported to more than one target system, ideally using different software and hardware stacks. In this case, Stage 3 would need to be repeated for each target. Arguments about diversity could then be employed in the correctness case. This would be especially useful if a fully-trusted target system could not be used in the audit, although the targets would have to be very carefully considered to not exhibit the same risks (such as the same propensity to display ambiguous concrete syntax for a given formula). The correctness case could then argue that the diversity of using more than one target could be considered as effectively equivalent to using a trusted target. Note that it is proof porting that allows the auditor the luxury of using diverse theorem provers.

4.4 Ideal Proof Auditing Capability

Here we describe desirable qualities for tools that could support our suggested proof auditing process, to enable them to be trusted and applied in practice.

4.4.1 *Ideal Proof Porting Capability.* There are various desirable qualities for the capability for exporting and importing proof objects.

Reliability. It should port proofs reliably and faithfully, so that the formal proof recreated in the target system accurately corresponds to the formal proof performed in the original system. Otherwise the proof auditor may be wasting their time reviewing the wrong formal proof, or checking that they are reviewing the right formal proof.

Modularity. It should be capable of exporting proof objects in a modular fashion, so that exporting can be split up into separate sessions and yet produce coherent proof objects that can be imported into a single session. Projects that are originally proved in many separate sessions (such as Flyspeck), cannot otherwise be ported.

Even if a project is proved in a single session, modular exporting can be used to avoid the risk that the exporter finds the session too big to export in a single session.

Ease of Use. It should not require special preparation to export a proof. Having to adjust proof scripts to work with the exporter, or adjust the exporter to work with the proof scripts, can significantly add to the effort and expertise required to audit a proof.

Common Format. Ideally, it should export proof objects in some common format that can be imported by various systems. This enables the proof to be diversely checked on more than one system.

Scalability. It should be able to handle recording, exporting and importing massive formal proofs without excessive demands on RAM usage, execution time or storage size. Large formalisation projects can involve hundreds of millions of primitive inferences. The more reliable and efficient the capability, the less effort the auditor needs to spend making the proof porting work.

4.4.2 *Ideal Target System.* There are various desirable qualities for the target system into which the proof objects are imported to check the proof.

Trustworthiness. It should be easy to trust for its logical soundness. It should have simple and well-documented implementation of its trusted core. If it is a theorem prover, it should have an LCF-style architecture with a small inference kernel, and this kernel should be watertight, so it cannot be circumvented or subverted. Ideally the system will be formally verified to correctly implement its formal logic.

Feedback. It should capable of providing to the user all information about the checked proof that is relevant for proof auditing, i.e. the statement of the final theorem, whether the final theorem was successfully checked, the state of the theory (its axioms, declarations and definitions), and the display settings.

Display of Formulae. It should reliably, unambiguously and faithfully display formulae in easy-to-read concrete syntax at a suitably high level. Ideally its pretty printer will be formally verified to be unambiguous and faithful.

Scalability. It should be capable of replaying massive formal proofs without excessive demands on RAM usage or execution time. Execution time should ideally be significantly less than the time to build the project on the source system, so that it is feasible to replay projects that have been split up over many sessions.

Open Source. Finally, the system should be open source, widely used and widely scrutinised, to minimise the risk of there being unknown flaws.

4.5 Potential Proof Auditing Tools

Here we consider various existing tools that could potentially support our suggested proof auditing process. These are considered with respect to the ideal tools we describe in Section 4.4.

4.5.1 *OpenTheory.* The OpenTheory project [14] is for enabling portability of proofs between HOL theorem provers, via the export of proof objects. It is based around the basic theory commands and inference rules of HOL Light. Exporters and importers for the OpenTheory proof object format have been implemented for HOL Light, HOL4 and ProofPower, and an importer also exists for Isabelle/HOL.

Projects can be exported in a modular fashion as collections of proof objects. HOL Light proof scripts need annotation before they can be recorded and exported, and so far this has been done for only about a third of the HOL Light standard theory library, which records and exports with an execution time overhead of about 140%, into a 5 MB `.tgz` file.

4.5.2 *Common HOL.* The Common HOL project [40] is for enabling portability of source code and proofs between HOL theorem provers. It is based around a platform [2] of basic theory, inference rules and utilities that is more-or-less common to all HOL systems. For source code portability, it provides an API of around 450 basic components, including, amongst other things, theory commands, basic inference rules and theorems.

For proof portability, Common HOL provides a proof object file format based on the API inference rules. An exporter for this format has been written for HOL Light, and importers for HOL Light and HOL Zero. Projects can be exported in a modular fashion as collections of proof objects, called *proof modules*. A proof module for the HOL Light standard theory library is recorded and exported with an execution time overhead of about 55%, into a 2.6 MB `.tgz` file, which can be read in and replayed in an import-adapted HOL Light core system in about 20% of the time it would normally take to build the library. The system is able to handle very large proofs, such as the main text part of the Flyspeck project, which it records and exports with an execution time overhead of about 100% into about 160 MB of `.tgz` files.[16] The import-adapted versions of HOL Zero and HOL Light are open source, but the export-adapted HOL Light is proprietary.

4.5.3 *ProofCert.* The ProofCert project [21] aims to enable checking of proofs from different proof systems and formal logics, via the export of proof objects. It involves designing a suitably generic proof object format and building a proof checker for this format. This project has the ambitious target of being able to use the same proof checker to check proofs from a diverse array of proof systems, including all kinds of theorem provers, dedicated SAT solvers and model checkers. The project is currently underway, and it is not yet clear whether it will deliver tools capable of checking large mathematics formalisation projects.

4.5.4 *Kaliszyk and Krauss.* Kaliszyk and Krauss [15] implemented a capability for porting proof objects from HOL Light to Isabelle/HOL. This exports a single monolithic proof object file for an entire session. The HOL Light standard theory library can be imported into Isabelle/HOL in about 40% of the time it would normally take to build in HOL Light. The system is also able to handle very large proofs, such as the main text part of an incomplete version of the Flyspeck project, with a recording and export execution time overhead of about 275% into about 180 MB of `.gz` file,[17] although the exporter used a bespoke adaption for Flyspeck to reduce the size of the exported proof objects and the replay time in Isabelle/HOL.

[16]Statistics are for HOL Light SVN release 197 and Flyspeck SVN release 3692, November 2014.
[17]Statistics are for HOL Light and Flyspeck SVN versions from July 2012.

4.5.5 *Other Proof Porting Tools.* Wong [30] implemented proof exporters for the HOL88 and hol90 theorem provers, and a proof checking tool for the exported proof objects. The proof checker is not a theorem prover, and just reads in a proof object file and checks that each step's output conforms to its inputs, outputting an overall "yes" or "no" for each proof object. Although able to work on proofs of several thousand steps, this was about the limit of its capability, and it was not maintained. The proof object files occupied considerably more disk space than corresponding files exported from the other proof porting capabilities we describe.

Obua and Skalberg [23] implemented a capability for porting proof objects from HOL4 and HOL Light to Isabelle/HOL. This exported proof objects as separate files but only for the entire session. The HOL4 and HOL Light standard theory libraries could be exported in 13 MB and 21 MB of `.tgz` file respectively, but the system struggled with significantly larger proofs such as Hales' formal proof of the Jordan Curve Theorem in HOL Light, and was not maintained.[18] Keller and Werner [16] used the Obua-Skalberg HOL Light exporter as the basis for porting proofs to Coq. This had the extra challenge of translating between logics, and also struggled with larger proofs.

4.5.6 *Coqchk.* Coqchk is a proof checking tool for Coq proofs that comes bundled with the Coq toolset. It has a separate implementation from the Coq theorem prover itself, although is developed by the same team. Coq is able to export proof objects (as `.vo` files), and these can be read in by Coqchk, which outputs an overall "yes" or "no" for each proof object. It is not capable of taking queries from the user about the state of the system, other than printing a list of the names of the axioms that have been assumed. It displays the names of the theorems that have been checked, but is not capable of displaying the statements of these theorems.

The proof objects exported by Coq are expressed at a relatively high level, and Coqchk must fill out reasoning gaps in formula normalisation, recursive function termination and universe chain ordering checks that are performed automatically by Coq's logical core. This makes Coqchk's implementation relatively complicated for a proof checker, and despite not supporting parsing or pretty printing of concrete syntax, it stretches to 4,500 lines of OCaml code plus over 3,000 lines of bespoke library and kernel OCaml code shared with Coq. Another problem is that Coqchk is known to struggle with checking some very large proofs.

4.5.7 *HOL Zero.* The HOL Zero system[19] [41] is a basic theorem prover for the HOL logic, developed mainly for checking formal proofs originally performed on other HOL systems. It has poor support for interactive and automated theorem proving, but has been carefully designed to excel at trustworthiness. It has simple and well-commented source code, that is intended to be as easy to review as possible, and robustly implements an LCF-style architecture. It has a queriable state for all axioms, definitions and display settings. It supports display of concrete syntax, but unlike the other HOL systems this display is unambiguous. Its core system is implemented in around 3,900 lines of OCaml, including around 1,250 lines for is

[18]Statistics are for unstated versions of HOL4 and HOL Light circa 2005.
[19]Our observations relate to HOL Zero version 0.6.2.

logical core.[20]

Although HOL Zero is implemented in OCaml, it manages to address the associated vulnerabilities that HOL Light suffers from, for example it protects against mutable strings by making copies at suitable points. There is also a bounty reward of $100 for discovering trustworthiness-related flaws (flaws in logical soundness or the display of formulae), and a list of exposed flaws is published on the HOL Zero homepage (the most recent flaw was in 2011). There are currently no known trustworthiness-related flaws other than its vulnerability to having trojan horses overwriting trusted components of the core (see Section 3.2.1).

4.5.8 *CakeML.* The CakeML project [39] is for building a verified software stack for a HOL theorem prover, involving a verified version of HOL Light [17] and a verified compiler for CakeML [18], a new dialect of ML based on SML. This project is producing trustworthy components that can be relied upon in future formalisation projects. The results have not yet filtered through to tools that can be used on actual projects, but the prospect is not far off.

5. AUDITING FLYSPECK

In this section, we show how the proof auditing process described in Section 4.3 can be applied by performing a partial audit of the Flyspeck formal proof. A full audit, covering all parts of Flyspeck, would be beyond the scope of this paper, for reasons discussed in Section5.2. Rather, the aim is to give the reader a flavour of what is involved, and to discuss how a full audit might be performed.

Because the bulk of Flyspeck is done in HOL Light, we chose a tool capability capable of exporting from HOL Light. We chose to use the Common HOL proof porting tools, since these currently have the best performance for porting very large proofs, and there is already an importer into HOL Zero, which we wanted as our target system because we view it as the most trustworthy HOL system.

Note that we have had some involvement in the Flyspeck project ourselves, and so our audit cannot constitute a proper audit because it is not independent.

5.1 The Partial Audit

5.1.1 *Replaying the Original Project.* Because the four parts of Flyspeck (the main text formalisation, the classification of tame graphs, the linear inequalities and the non-linear inequalities) are formalised across two theorem provers and a multitude of sessions, totalling around 5,000 hours of processing, replaying the project is not an easy exercise.

Instead, our partial audit concentrated on just the main text formalisation, which is meant to process in a single HOL Light session. Using modest hardware,[21] the processing completed in around 3.5 hours. We used HOL Light SVN release 197, OCaml version 3.12.1 and Flyspeck SVN release 3692. The project build file for this part of Flyspeck is `text_formalization/build.hl`. However, note that this build file also incorporates the linear inequalities, so we did not need to process every line in this file, and the final file `general/the_kepler_conjecture.hl` only

[20]We consider the HOL Zero logical core to consist of the build files up to `thm.ml` excluding `reader.ml` but including `corethry.ml`, and its core system to consist of all files up to `store.ml`.
[21]We used a 2.5 GHz Intel Core i5 CPU with 8 GB RAM running 64-bit Linux.

```
|- !a. tame_classification a /\
       good_linear_programming_results a /\
       the_nonlinear_inequalities
       ==> the_kepler_conjecture
```

Fig. 8. The final theorem for the main text.

needed processing up until the theorem `kepler_conjecture_with_assumptions`, i.e. the final theorem for the main text formalisation.

The processing completed successfully with the proof of the final theorem. This theorem states that the Kepler Conjecture holds assuming some tame graph classification and linear inequalities for this classification and the non-linear inequalities (see Figure 8).

5.1.2 *Porting the Project.* To port the proof, we used Common HOL's export-adapted HOL Light to export the proof objects as proof modules each corresponding to a block of lines in the Flyspeck main text formalisation build file. Total processing time, including the time for exporting the proof modules, took around 7 hours, about twice the time for processing the scripts with unadapted HOL Light. The exported proof objects, compressed as `.tgz` files, occupied 159 MB on disk.

We then imported the proof objects into a single session of import-adapted versions of HOL Zero and HOL Light. For HOL Zero, this took around 6 hours for the entire main text formalisation. For HOL Light, import was much quicker, taking just over 30 minutes. Even though there are no known flaws in HOL Zero, we used the diversity of two target systems to give us added assurance that no unknown flaws were being exploited.

The exported proof modules, together with import-adapted versions of HOL Light and HOL Zero, are available for download from the Flyspeck homepage on the Proof Technologies website [42].

5.1.3 *Examining the Target Systems' Final State.* We first examined the HOL Zero target session.

The HOL Zero display settings had numerous extra fixities declared, which we took into account when reviewing the statement of the final theorem and its supporting definitions.

HOL Zero's list of axioms was not added to by importing the main text formalisation, confirming that the theory had not been made inconsistent as part of the formal proof.

Reassuringly, HOL Zero displayed precisely the same final theorem as the project version of HOL Light (see Figure 8). In our review of the meaning of this final theorem in HOL Zero, we only examined its consequent, i.e. the formal statement of the Kepler Conjecture, because the antecedents, once instantiated with the tame graph classification, are intermediate results that would be discharged when replaying the entire project. In any case, these instantiated antecedents expand into huge expressions, and reviewing them would be a mammoth task.

The formal statement of the Kepler Conjecture is defined as a constant called `the_kepler_conjecture` (see Figure 9). Bound variable V in this definition represents a set of points in 3-dimensional space, corresponding to the centres of unit-

```
|- the_kepler_conjecture <=>
    (!V. packing V
         ==> (?c. !r. real_of_num 1 <= r
                      ==> real_of_num (CARD (V INTER ball (vec 0, r)))
                          <= pi * r pow 3 / sqrt (real_of_num 18) +
                             c * r pow 2))
```

Fig. 9. The formal statement of the Kepler Conjecture.

```
|- !S. packing S <=>
        (!u v. S u /\ S v /\ ~ (u = v)
               ==> (real_of_num 2 <= dist (u,v)))
```

Fig. 10. The definition of `packing`.

radius spheres. The statement says that if V is a sphere packing, then there exists a constant c such that, for any radius r, the number of spheres from V with centres inside a containing sphere of radius r is no greater than $\pi r^3/\sqrt{18} + cr^2$.

Although the formal statement does not transparently correspond to the informal statement (see Figure 1), it is fairly straightforward to see why the two are equivalent, assuming appropriate definitions of the constants used in the formal statement. By dividing both sides of the inequality by r^3, we can see that right-hand side is $\pi/\sqrt{18}$ plus an error term of c/r.[22] This error term tends to 0 as radius r tends to infinity. The left-hand side of the inequality is then a quotient, and multiplying top and bottom by $4/3\ \pi$ respectively gives the total volume of the packing's spheres that lie inside the containing sphere,[23] and the volume of the containing sphere, i.e. the density A/B from the informal statement. Finally, we can generalise the result for packings of unit-sized spheres to packings of same-sized spheres and any size.

We next shifted our attention to the definition of the constant `packing` (see Figure 10), used in the above definition. Here `S` is a set of points, represented as a predicate on points. From the formal definition we can see that `S` is deemed to be a packing if any two distinct points are at least two units apart. This corresponds to our expected notion of a sphere packing, because the spheres in the formal statement are of unit radius.

It is possible to see how this audit in HOL Zero would continue. The full dependency graph of constant definitions for all constants featured in the statement of the final theorem would need examination, including `vec`, `ball`, `dist`, `CARD` and `real_of_num`.

We then examined the final state of the HOL Light target session. Reassuringly, this gave identical results to the HOL Zero session.

[22]The need for the error term is most obvious when radius r is small. If r is 1, for example, then the density of the packing can be 100%.

[23]More precisely, it gives the total volume of the packing's spheres with *centres* that lie inside the containing sphere, but as r tends to infinity, this gives the same result. The justification that this gives the same result is not addressed in this partial audit.

5.2 Auditing the Whole of Flyspeck

Using our process to audit the whole of Flyspeck, which would involve incorporating the entire formalisation into a single theorem prover session, would be more difficult than auditing just the main text formalisation. This is because it presents various technical challenges for tool support.

Ideally, the replaying of the formal proof in a target system would rely on just the core system of the target system, and not on a program extraction facility that carries its own risks of flaws. For this, the formalisation of the classification of tame graphs would need to be reworked so as to prove the archive as a HOL theorem, and to export the proof of this theorem as a Common HOL proof module. This theorem could be proved in HOL Light or Isabelle/HOL. It might be easier to perform this proof by adapting the existing work done in Isabelle/HOL, but it would require an Isabelle/HOL Common HOL proof exporter.

Porting the linear inequalities may cause problems for the HOL Light Common HOL exporter due to the large size of the terms involved.

Porting each of the 600 or so sessions for solving the non-linear inequalities may also present challenges. HOL Light Common HOL proof exporting tends to take about twice the time to prove, record and export as it does to just prove, and so it should not be a major problem to rerun each of the sessions in parallel with proof exporting turned on. However, importing these proof modules into a single consolidated HOL session might cause problems for the target system. The current version of HOL Zero would probably struggle to process everything in reasonable time. Import into a HOL Light session looks more promising, given that the text formalisation imported and replayed in HOL Light seven times faster than running the proof scripts. This could feasibly be achieved with a month of processing time, but it would be necessary to somehow mitigate HOL Light's trustworthiness concerns.

An alternative approach to the above is to use Isabelle/HOL as the target system. It may be simpler to keep the classification of tame graphs in Isabelle/HOL and port the rest of Flyspeck from HOL Light into Isabelle/HOL. This could be done using Kaliszyk and Krauss's existing porting capability or OpenTheory, although implementing a Common HOL proof importer for Isabelle/HOL looks like the most likely to succeed given the sheer size of the HOL Light proof. Mitigation of Isabelle/HOL's trustworthiness concerns, including any concerns about its program extraction facility (unless the proof were reworked to avoid this), would be necessary.

It may be technically unachievable to incorporate the whole of Flyspeck into a single session of a theorem prover, at least with current technology. If this is the case, the human aspect of the audit would have to take on a much greater role, expanding to include review of the interfaces between the proof sessions. The greatest focus here would need to be on the correctness of translation of results between systems.

5.3 Puzzle Solution

The root of the problem in Figure 2 is the innocuously named `eval` function defined in the three lines in Figure 12 from `update_database_310.ml` (loaded from

```
let MUL_POW2 = REAL_ARITH` (a*b) pow 2 = a pow 2 * b pow 2 `;;



let COMPUTE_SIN_DIVH_POW2 = prove(`! (v0: real^N) va vb vc.
let betaa = dihV v0 vc va vb in
let a = arcV v0 vc vb in
let b = arcV v0 vc va in
let c = arcV v0 va vb in
let p =
&1 - cos a pow 2 - cos b pow 2 - cos c pow 2 +
&2 * cos a * cos b * cos c in
~collinear {v0, vc, va} /\ ~collinear {v0, vc, vb} ==>
( sin betaa ) pow 2 = p / ((sin a * sin b) pow 2) `,

REPEAT STRIP_TAC THEN MP_TAC (SPEC_ALL RLXWSTK ) THEN
REPEAT LET_TAC THEN SIMP_TAC[SIN_POW2_EQ_1_SUB_COS_POW2 ] THEN
REPEAT STRIP_TAC THEN REPLICATE_TAC 2 (FIRST_X_ASSUM MP_TAC) THEN
NHANH (NOT_COLLINEAR_IMP_NOT_SIN0) THEN
EXPAND_TAC "a" THEN EXPAND_TAC "b" THEN PHA THEN
SIMP_TAC[REAL_FIELD` ~( a = &0 ) /\ ~ ( b = &0 ) ==>
&1 - ( x / ( a * b )) pow 2 = (( a * b ) pow 2 - x pow 2 ) / (( a * b ) pow 2 ) `
] THEN
ASM_SIMP_TAC[] THEN STRIP_TAC THEN
MATCH_MP_TAC (MESON[]` a = b ==> a / x = b / x `) THEN
EXPAND_TAC "p" THEN SIMP_TAC[MUL_POW2; SIN_POW2_EQ_1_SUB_COS_POW2] THEN
REAL_ARITH_TAC);;
```

Fig. 11. The undoctored extract from `trigonometry/trig2.hl`.

```
let eval n =
  exec ("let buf__ = ( " ^ n ^ " );;");
  Obj.magic (Toploop.getvalue "buf__");;
```

Fig. 12. The `eval` function from `general/update_database_310.ml`.

`strictbuild.hl`). This uses `Obj.magic` and can be used to construct arbitrary theorems (see Section 3.2.1). We inserted into the Figure 2 code fragment a definition for an ML value named `x`, that can be typecasted to a theorem (in OCaml, integer `0` has the same representation in memory as an empty list of HOL terms, and likewise `Some` has the same representation as `Sequent`, used to construct theorems). We embedded a call to `eval` in a rewrite list in the proof script, so that `x` becomes a theorem for rewriting `x pow 4` to `x pow 2`, which gets applied to `(sin a * sin b) pow 4` to prove the unprovable (the statement of the theorem should read `pow 2` instead of `pow 4` in the last line). See Figure 11 for the real original Flyspeck code fragment.

This puzzle underlines how difficult it can be to discover subtle problems by examining the project proof scripts. Would the project's manager, or an auditor for that matter, really notice such a problem buried in one of many 5,000 line proof scripts? Note that, even if the auditor were aware of `eval`, they might miss its usage here, given that that there are over 30 other occurrences of the ML binding name in the Flyspeck proof scripts, all of which have nothing to do with the `eval` defined in `update_database_310.ml`. However, using our suggested proof auditing process, of porting the proof to another session and examining the resulting system state, would uncover the problem. In this particular example, `trig2.hl` would fail to build in the export-adjusted HOL Light, due to the implicit typecast failing.

6. RELATED WORK

Wong [30] advocated checking formal proofs by porting them to a trustworthy proof checker, and, as described in Section 4.5.5, developed proof exporters for HOL88 and HOL90 and a HOL proof checker. This capability has been demonstrated to work on proofs of several thousand inference steps. However, the aim of such checking was purely to address the concern of whether the formal proof is a correct derivation, and the proof checker just outputted an overall "yes" or "no" for each proof object. This is more limited than our notion of a proof audit, which includes review of the axioms and final theorem of a formal proof project, and consideration of how a project split over several sessions fits together.

Pollack [24] wrote a paper on the philosophy of trusting formalisation proofs. He stated two main concerns: whether the formal proof constitutes a correct derivation, and whether the right theorem has been proved. To address the first, he advocated porting the proof to a trustworthy proof checker. To address the second, he advocated using the proof checker to review the axioms, definitions and statement of the final theorem. Pollack states similar concerns to our sceptic, and advocates an appraoch to proof auditing similar to ours. However, he concentrates on use of a logical framework as the basis for providing a trustworthy proof checker, and gives little detail on justifying the auditing approach or the practicalities of carrying it out. Also, if only due to the lack of available technology and sizeable formalisations at the time of writing, he does not use detailed motivating examples.

7. CONCLUSIONS

The formalisation of mathematics by use of theorem provers has reached the stage where previously questioned mathematical proofs have now been formalised. Sceptics, however, have good reason to doubt that nowhere in these large projects are lingering subtle problems that render the formalisation invalid. There are various pitfalls of formalisation, ranging from knowingly or unknowingly exploiting flaws in the theorem provers being used (and such flaws do exist) to simply proving the wrong theorem. It is certainly not satisfactory to rely on reputation to answer the sceptics, since project teams are composed of human beings who may become motivated by things other than veracity, especially when outsourcing is used.

However, despite the complexity of the major formalisations, and despite their mathematical and historical importance, they have not yet been subject to comprehensive, independent scrutiny. This sits uncomfortably with the mathematical certainty that these projects are supposed to establish. To address this situation, we propose the activity of *proof auditing*, to independently assess a formalisation.

In order to avoid the risk of proof auditing itself being a long and error prone activity, we propose a process based around proof porting technology with a trustworthy target system. This sidesteps any flaws that may exist in the project theorem prover, and means that the auditor only needs to review the target system's final state, rather than the project proof scripts and the project theorem prover.

This makes it possible to effectively audit a large project with minimal effort, and yet achieve the highest levels of assurance.

This approach will work for any project provided that sufficiently strong tool support exists. Proof importers and exporters capable of porting large proofs are being written for the various HOL systems as part of both the OpenTheory and Common HOL projects, and HOL Zero is a suitable blank canvas to act as a trustworthy target system. Coq already supports export of proof objects, but its dedicated checker Coqchk falls short of our criteria in some respects, although these issues could be addressed in a future version. For Mizar we know of no system for porting proof objects, and considerable investment may be required. Overall, the biggest technical challenge for tool support for our process is handling import into a single consolidated session of a target system for projects that involve a multitude of computationally-intensive parallel sessions, such as Flyspeck.

We demonstrate the feasibility of our approach by carrying out a partial audit of Flyspeck, using Common HOL proof porting to export a significant proportion of the formal proof, and importing and reviewing it in both a skeleton HOL Light system and HOL Zero. As far as we are aware, this is the first large-scale deployment of proof auditing on a major formalisation project. We also demonstrate the effectiveness of the process by showing how it can easily detect an obscure malicious piece of code, which exploits a weakness in HOL Light, that would otherwise typically evade detection. It is technically feasible that the whole of Flyspeck could be audited using the same simple process, although this would first require significant rework to get the Isabelle/HOL portion of the proof expressed in HOL Light, and with current tools the auditor would have to wait for perhaps a month whilst the consolidated 600 sessions import into a single target session.

Fast and effective auditing of large formalisation projects is within reach, and the mathematics formalisation community should seize this opportunity to banish all reasonable doubt about the correctness of their work.

ACKNOWLEDGMENTS

We wish to thank Josef Urban, Cezary Kaliszyk, Robbert Krebbers and the reviewers for their many helpful comments on this paper. We are also grateful to Tom Hales and John Harrison for providing assistance with aspects of the partial audit.

References

[1] Anon. *The QED Manifesto.* In Proceedings of the 12th International Conference on Automated Deduction, Volume 814 of Lecture Notes in Computer Science, pages 238-251. Springer, 1994.

[2] M. Adams. *The Common HOL Platform.* Accepted for publication in Proceedings of the Fourth International Workshop on Proof eXchange for Theorem Proving, 2015. Preprint available at: `http://www.proof-technologies.com/commonhol/platform_pxtp2015.pdf`.

[3] Y. Bertot & P. Casteran. *Interactive Theorem Proving and Program Development: Coq'Art: The Calculus of Inductive Constructions.* Springer, 2004.

[4] G. Gonthier. *Formal Proof - The Four-Color Theorem.* In Volume 55(11) of Notices of the American Mathematical Society, pages 1382-1393. AMS, 2008.

[5] G. Gonthier et al. *A Machine-Checked Proof of the Odd Order Theorem.* In Proceedings of the 4th International Conference on Interactive Theorem Proving, Volume 7998 of Lecture Notes in Computer Science, pages 163-179. Springer, 2013.

[6] M. Gordon, R. Milner & C. Wadsworth. *Edinburgh LCF: A Mechanised Logic of Computation.*, Volume 78 of Lecture Notes in Computer Science. Springer, 1979.

[7] T. Hales & S. Ferguson. *A Formulation of the Kepler Conjecture.* Volume 36(1) of Discrete and Computational Geometry, pages 21-69. Springer, 2006.

[8] T. Hales. *Introduction to the Flyspeck Project.* In Mathematics, Algorithms, Proofs, Volume 05021 of Dagstuhl Seminar Proceedings. Internationales Begegnungs- und Forschungszentrum für Informatik, 2005.

[9] T. Hales et al. *A Revision of the Proof of the Kepler Conjecture.* Volume 44(1) of Discrete and Computational Geometry. Springer, 2010.

[10] T. Hales. *Dense Sphere Packings: A Blueprint for Formal Proofs.* Volume 400 of London Mathematical Society Lecture Note Series. Cambridge University Press, 2012.

[11] T. Hales et al. *A Formal Proof of the Kepler Conjecture.* arXiv:1501.02155v1 [math.MG]. arxiv.org, 2015.

[12] J. Harrison. *HOL Light: An Overview.* In Proceedings of the 22nd International Conference on Theorem Proving in Higher Order Logics, Volume 5674 of Lecture Notes in Computer Science, pages 60-66. Springer, 2009.

[13] J. Harrison *Towards Self-Verification of HOL Light.* In Proceedings of the Third International Joint Conference on Automated Reasoning, Volume 4130 of Lecture Notes in Computer Science, pages 177-191. Springer, 2006.

[14] J. Hurd. *The OpenTheory Standard Theory Library.* In Proceedings of the Third International Symposium on NASA Formal Methods, Volume 6617 of Lecture Notes in Computer Science, pages 177-191. Springer, 2011.

[15] C. Kaliszyk & A. Krauss. *Scalable LCF-Style Proof Translation.* In Proceedings of the 4th International Conference on Interactive Theorem Proving, Volume 7998 of Lecture Notes in Computer Science, pages 51-66. Springer, 2013.

[16] C. Keller & B. Werner. *Importing HOL Light into Coq.* In Proceedings of the First International Conference on Interactive Theorem Proving, Volume 6172 of Lecture Notes in Computer Science, pages 307-322. Springer, 2010.

[17] R. Kumar et al. *CakeML: A Verified Implementation of ML.* In the 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pages 179-191. ACM, 2014.

[18] R. Kumar et al. *HOL with Definitions: Semantics, Soundness, and a Verified Implementation.* In Proceedings of the 5th International Conference on Interactive Theorem Proving, Volume 8558 of Lecture Notes in Computer Science, pages 308-324. Springer, 2014.

[19] O. Kuncar. *Correctness of Isabelle's Cyclicity Checker: Implementability of Overloading in Proof Assistants.* In Proceedings of the 2015 Conference on Certified Programs and Proofs, pages 85-94. ACM, 2015.

[20] D. MacKenzie. *Mechanizing Proof: Computing, Risk, and Trust.* MIT Press, 2001.

[21] D. Miller et al. *ProofCert: Broad Spectrum Proof Certificates.* ERC Advanced Grant 2011 Technical Description. INRIA Saclay, 2011.

[22] T. Nipkow, L. Paulson & M. Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic.* Volume 2283 of Lecture Notes in Computer Science. Springer, 2002.

[23] S. Obua & S. Skalberg. *Importing HOL into Isabelle/HOL.* In Proceedings of the Third International Joint Conference on Automated Reasoning, Volume 4130 of Lecture Notes in Computer Science, pages 298-302. Springer, 2006.

[24] R. Pollack. *How to Believe a Machine-Checked Proof.* In Twenty-Five Years of Constructive Type Theory, chapter 11. Oxford University Press, 1998.

[25] A. Trybulec & H. Blair. *Computer Assisted Reasoning with MIZAR.* In Proceedings of the 9th International Joint Conference on Artificial Intelligence, pages 26-28. Morgan Kaufmann, 1985.

[26] M. Wenzel, L. Paulson & T. Nipkow. *The Isabelle Framework.* In Proceedings of the 21st International Conference on Theorem Proving in Higher Order Logics, Volume 5170 of Lecture Notes in Computer Science, pages 33-38. Springer, 2008.

[27] F. Wiedijk. *Comparing Mathematical Provers.* In Proceedings of the Second International Conference on Mathematical Knowledge Management, Volume 2594 of Lecture Notes in Computer Science, pages 188-202. Springer, 2003.

[28] F. Wiedijk (Ed.). *The Seventeen Provers of the World.* Volume 3600 of Lecture Notes in Computer Science. Springer, 2006.

[29] F. Wiedijk. *Pollack-Inconsistency.* In Volume 285 of Electronic Notes in Theoretical Computer Science, pages 85-100. Elsevier Science, 2012.

[30] W. Wong. *Validation of HOL Proofs by Proof Checking.* Formal Methods in System Design 14. Kluwer Academic Publishers, 1999.

[31] Four Color Theorem formalisation homepage: `http://research.microsoft.com/en-us/downloads/5464e7b1-bd58-4f7c-bfe1-5d3b32d42e6d/default.aspx`.

[32] Odd Order Theorem formalisation homepage: `http://ssr.msr-inria.inria.fr/`.

[33] Flyspeck homepage: `http://code.google.com/p/flyspeck/`.

[34] Falso GitHub homepage: `https://github.com/clarus/falso`.

[35] Coq Club Mailing List archive for December 2013, message 119: `https://sympa.inria.fr/sympa/arc/coq-club/2013-12/msg00119.html`.

[36] Coq Developers Mailing List archive for January 2014, message 45: `https://sympa.inria.fr/sympa/arc/coqdev/2014-01/msg00045.html`.

[37] Coq Developers Mailing List archive for April 2014, message 52: `https://sympa.inria.fr/sympa/arc/coqdev/2014-04/msg00052.html`.

[38] Mizar Forum Mailing List archive for June 2003, message 10: `http://mizar.uwb.edu.pl/forum/archive/0306/msg00010.html`.

[39] CakeML homepage: `https://cakeml.org/`.

[40] Common HOL homepage:
`http://www.proof-technologies.com/commonhol/`.

[41] HOL Zero homepage: `http://www.proof-technologies.com/holzero/`.

[42] Proof Technologies Flyspeck homepage:
`http://www.proof-technologies.com/flyspeck/`.