Papers with computer-checked proofs

Daniel J. Bernstein[1,2]

[1] Department of Computer Science, University of Illinois at Chicago, USA
[2] Horst Görtz Institute for IT Security, Ruhr University Bochum, Germany
djb@cr.yp.to

Abstract. This report gives case studies supporting the hypothesis that it is often affordable for a paper presenting theorems to also include proofs that have been checked with today's proof-checking software.

Keywords: mathematics, proofs, formalization, computer verification, computer assistance, human time

1 Introduction

> *The theorem is thought to be true in the classical sense—that is, in the sense that it* could *be demonstrated by formal, deductive logic, although for almost all theorems no such deduction ever took place or ever will.* —1979 DeMillo–Lipton–Perlis [40, page 274]

> *The standard benchmark for the human labor to transcribe one printed page of textbook mathematics into machine-verified formal text is one week, or US$150 per page at an outsourced wage.*
> —2008 Hales [57, page 1379]

> *In the most ideal circumstances, an expert can handle approximately a half page to a page of a substantial mathematical text in a long, uninterrupted day of formalizing.*
> —2014 Avigad–Harrison [7, page 75]

> *When mathematicians say that a theorem has been "proved," they still mean, as they always have, something more like: "we've reached a social consensus that all the ideas are now in place for a strictly*

The case studies covered in this report include work funded by the Intel Crypto Frontiers Research Center; by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) as part of the Excellence Strategy of the German Federal and State Governments—EXC 2092 CASA—390781972 "Cyber Security in the Age of Large-Scale Adversaries"; by the U.S. National Science Foundation under grants 1913167 and 2037867; by the Taiwan's Executive Yuan Data Safety and Talent Cultivation Project (AS-KPQ-109-DSTCP); and by the Cisco University Research Program. "Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation" (or other funding agencies). Permanent ID of this document: ea6630d86ffe16ff50644f7b5f1ebd0b2c79aaab. Date: 2023.09.09.

> *formal proof that could be verified by a machine ... with the only task remaining being massive rote coding work that none of us has any intention of ever doing!"* —2019 Aaronson [1]

It's easy to find examples of papers *on* computer-checked proofs. Consider, e.g., [54], a 2013 paper by Gonthier, Asperti, Avigad, Bertot, Cohen, Garillot, Le Roux, Mahboubi, O'Connor, Ould Biha, Pasca, Rideau, Solovyev, Tassi, and Théry titled "A machine-checked proof of the odd order theorem". The odd-order theorem says that every finite group of odd order is solvable. The original paper proving this theorem, 1963 Feit–Thompson [44], had 255 pages; for comparison, [54] reported that its machine-checked proof had more than 150000 lines.

The general publication structure here has two stages: (1) there's a paper with a theorem and a traditional-format proof; (2) eventually there's a separate paper on a computer-checked proof.

Some mathematicians perceive a paper on computer-checked proofs as rote work—maybe useful, but not a "real" paper. Often the second paper, perhaps trying to add evidence against this perspective, says that the second proof fixes gaps and inefficiencies from the first, increases generality, and, more broadly, displays increased understanding. It is time-consuming for readers to figure out how to divide credit between the two proofs.

Now consider, as an alternative, papers *with* computer-checked proofs, by which I mean papers that merge the two stages listed above: the first paper, the one presenting the theorem, also includes a computer-checked proof. This won't retroactively happen for a paper that appeared in 1963, but the time-travel obstacle doesn't apply to new papers.

There are reasons to think that this single-stage process has fundamentally better efficiency and better incentives than the two-stage process:

- The author writing the theorem and the traditional-format proof already knows how the proof works. Most people writing a computer-checked version of the proof would have to take time to read and understand the proof; the author is in a better position.
- In cases where the traditional-format proof is wrong, writing a computer-checked proof will catch the mistake, and doing this as early as possible will minimize the time wasted by the mistake.
- The author can choose a different writing order to save more time: for example, first writing the computer-checked proof, and then extracting various steps to write the traditional-format proof. Computer assistance can save time for all parts of this process: see, e.g., [48].
- Referees and other readers provided with a computer-checked proof spend much less time worrying about the possibility that the theorem is wrong: in the words of 1991 de Bruijn [28], referees "need not bother about correctness and can concentrate on whether the paper is interesting and new". The current situation is that referees checking proofs—or delaying so as to give the impression that they might be checking proofs—create random, often annoyingly large, slowdowns in paper acceptance, so publish-or-perish

authors have a direct incentive to lighten the referee's load. As 2011 Bundy [31, Section 6.4] puts it: "Their reward for this additional effort would be a quicker refereeing process and the confidence that no embarrassing error would be found in their proofs."

- Readers don't have to figure out how to divide credit between a traditional-format proof and a computer-checked proof. The author receives credit for the whole package.

Readily available "proof assistants" include not just proof-checking software but also (1) software to help authors construct checkable proofs and (2) libraries of previously checked proofs. Six proof assistants mentioned later in this report are Coq, HOL4, HOL Light, Isabelle/HOL, Lean, and Mizar; see [21], [100], [60], [88], [85], [82], [9], [8], and [37] for more about Coq, HOL4, HOL Light, Isabelle/HOL, the core Lean 4 language, the Lean math library, the core Mizar language, the Mizar math library, and a recent open-source reimplementation of portions of Mizar. (This report, following commonly used terminology, says "Lean" etc. for the full proof assistants, including the math libraries.)

However, as far as I can tell, papers *with* computer-checked proofs—see below for some examples—are much less common than papers *on* computer-checked proofs. The elephant in the room, the obvious reason that most mathematicians today wouldn't even *try* to write papers with computer-checked proofs, is that the literature gives mathematicians the impression that producing a computer-checked proof is a painful, specialized process. For example:

- Regarding pain, 150000 lines from 15 authors sounds like a terrifying amount of work—that's 3000 pages at 50 lines per page, an order of magnitude larger than the 255 pages of the original paper. This sounds like it easily justifies, e.g., [1] describing the process as "massive rote coding work".
- Regarding specialization, having a separate side of the literature where proof-checking experts write papers on computer-checked proofs makes it easy to assume that non-experts on proof-checking would be vastly slower until they have gone through a long training period. This assumption is reinforced by, e.g., the above quote from [7] saying specifically what an expert can accomplish.

Does this impression match the actual situation in 2023? Could it be that half of the proofs that will be published next year are proofs that the authors would find affordable to computer-check with today's proof assistants as part of the paper-writing process, if they were simply made aware that they should try?

I have three preprints online with computer-checked proofs. I've also posted another computer-checked proof where I hope a joint preprint will be online soon. Section 3 summarizes the time that I spent writing computer-checked proofs for each of these four papers: in short, a few weeks per paper.

Line counts—my computer-checked proofs are 2150 lines for [13], for example, and 9951 lines for [17]—are terribly misleading. Readers hearing such line counts don't imagine that they're just a few weeks of work. Section 2 pinpoints reasons for the gap. (Similar gaps are visible, but not explained, in [67] and [62].)

Meanwhile the parts of these four papers *other than the computer-checked proofs* easily account for a year of work across the four papers. The papers have non-proof components such as software, but, in each case, coming up with the theorems in the first place was a major part of the research time. Adding a few weeks per paper for computer-checked proofs is certainly a noticeable extra cost, but also an affordable extra cost.

For one of these papers, I proved all theorems using the Lean proof assistant, and then proved all theorems again using the HOL Light proof assistant. The experiences were broadly similar, but I observed some differences that saved time for writing the Lean proofs, and some differences that saved time for writing the HOL Light proofs. See Section 4. Presumably it would be possible to modify a proof assistant to save time in both ways.

I also have two recent preprints with theorems *without* computer-checked proofs. See Section 5. My current estimate is that extending a proof-assistant library to cover all the necessary pieces of background would need a few months of work in each case.

1.1. Related work. My papers *with* computer-checked proofs are certainly not the first such papers. Here are examples from several areas of mathematics and computer science:

- 2006 Harrison [**61**] proved that consistency of the basic logic used in HOL Light follows from a large-cardinal assumption, and reported that the proof was checked by HOL Light. See also [**2**] for the latest extensions of this result.
- 2018 Buchholtz–van Doorn–Rijke [**29**] said "We present a development of the theory of higher groups, including infinity groups and connective spectra, in homotopy type theory. ... Most of the results have been formalized in the Lean proof assistant." (Converting a proof into a computer-checked version of the proof is typically referred to as "formalizing" the proof, although this can be confusing for readers who expect "formal" vs. "informal" to refer to a full traditional-format proof vs. a mere summary of proof highlights.)
- 2019 Gouëzel–Shchur [**56**] said "There is a gap in the proof of the main theorem in the article [5] on optimal bounds for the Morse lemma in Gromov hyperbolic spaces. We correct this gap, showing that the main theorem of [5] is true. We also describe a computer certification of this result." This certification uses the Isabelle/HOL proof assistant. The paper explains that the gap was found as part of an attempt to computer-check an earlier proof—but to me [**56**] still has a very different flavor from papers *on* computer-checked proofs. Other papers presenting corrected proofs of previously claimed theorems look like this one; this one simply has the extra feature of bolstering confidence by presenting a computer-checked proof.
- 2019 Strickland–Bellumat [**102**] said "We study a certain monoid of endofunctors of the stable homotopy category that includes localizations with respect to finite unions of Morava K-theories. ... The combinatorial parts of this work have been formalised in the Lean proof assistant".
- 2020 Gouëzel–Karlsson [**55**] said "A result for subadditive ergodic cocycles is proved that provides more delicate information than Kingman's subadditive

ergodic theorem ... As a test case for the usability of proof assistants for current mathematical research, Theorem 1.1 and its proof given below have been completely formalized and checked in the proof assistant Isabelle/HOL".

- 2021 Bloom [24] proved "that any set $A \subset \mathbb{N}$ of positive upper density contains a finite $S \subset A$ such that $\sum_{n \in S} \frac{1}{n} = 1$, answering a question of Erdős and Graham". Reportedly the paper is being updated to include an appendix describing a Lean-checked proof by Mehta and Bloom. I should note that [35] says the verification of [24] was "before Bloom had received a referee's report for the paper"—which is not the same as saying it was fast enough to save time for the referee, one of the basic reasons to write papers with computer-checked proofs.
- 2022 Wang–Zhang–Shao–Koenig [106] proved that a particular compiler correctly translates programs from a particular variant of the C programming language to machine language using a "nominal memory model". This paper uses the Coq proof assistant (as do many other POPL papers).
- 2023 Topaz [104] said "We present a variant of the fundamental theorem of alternating pairs which works for arbitrary fields of positive characteristic p and arbitrary coefficient fields of characteristic not dividing $2 \cdot p$. ... the proof of Theorem A has been formally verified". This verification uses Lean.

Some of these are examples I bumped into, and some are examples that people have helpfully pointed me to.

Note that, as [106] illustrates, papers with computer-checked proofs don't necessarily have traditional-format proofs. This might make them look like papers *on* computer-checked proofs. I don't claim that there's always a sharp line, but it's useful to ask whether the main goal of the paper is to computer-check a theorem from a previous paper; whether there was the initial stage of referees looking for errors in the original traditional-format proof; and whether there's the followup credit battle between computer-checked proof C saying "T didn't *really* prove that" and traditional-format proof T saying "yes, I did, C is just typing in minor details and nitpicks".

There are also various examples of theorems that first appeared as computer-checked theorems and that are clearly worthy of papers, but where papers haven't been written, often because the theorems are applied math and the authors are busy with the applications. If the papers end up being written then they will be papers with computer-checked proofs. An impressive new example is Harrison's `CURVE25519_X25519_BYTE_SUBROUTINE_CORRECT` from [64], a HOL Light proof that some fast machine-language software correctly computes the encryption function for an elliptic-curve cryptosystem; the speed per se is cutting-edge research, the theorem statement relies on having definitions of the machine language and of the mathematics, and the proof relies on a wide range of tools to connect these objects.

Even though there are some examples of papers with computer-checked proofs, it's clear to me that most mathematicians today aren't familiar with the process of writing computer-checked proofs, and have a very wrong impression of the

difficulty of the process. So I think there's value in reporting my own case studies of the process, and value specifically in challenging the misleading usage of line counts. I hope to soon see many more authors reporting not just the line counts but the *amount of time used* for successfully writing computer-checked proofs for their own work.

2 A beginner writing computer-checked proofs

I heard about proof-checking software last century, but my first project using such software was [13], the first of the papers covered in Section 3. This section explains the process I followed to start that project. The process doesn't match any of the proof-assistant tutorials I've looked at. This section also explains why readers should usually disregard line counts for computer-checked proofs.

2.1. Stating theorems. I picked the HOL Light proof assistant, for reasons discussed in Section 4. As one part of the selection process, I looked at some random examples of theorems from the HOL Light library.

The basic syntax for theorems in HOL Light is not normal mathematical writing. Imagine some logician suddenly being in charge of mathematics and saying that we all have to use special symbols $\wedge$ and $\vee$ for "and" and "or" since, well, that's what logicians do, and, see, the symbols make perfect sense, just like $\cap$ and $\cup$ for sets. And that's just the start. Oops, rewrite: $\wedge$ that's just the start.

But, okay, a bit of superficial renaming isn't enough to scare me away. I am not a wimp. This isn't *that* hard to read:

```
!m n. m MOD n = m <=> n = 0 \/ m < n
```

Translation: for each $m, n \in \{0, 1, 2, \ldots\}$, we have $m \bmod n = m$ if and only if $n = 0$ or $m < n$. (As usual, $m \bmod n$ means the remainder when m is divided by n.)

The only slightly tricky part of this translation is that definitions in HOL Light are typically set up to create hypotheses on the input ranges, such as `MOD` automatically creating a hypothesis that the inputs are in $\{0, 1, 2, \ldots\}$. The `<=>` is the logician's $\Leftrightarrow$, which for me is part of blackboard style anyway, although in a paper I would write "if and only if". The `\/` ("or") is the logician's aforementioned $\vee$ symbol, and the `!` ("for all") is an extremely rough ASCII approximation to the logician's $\forall$.

Actually, logicians who are serious about Sensible Notation write "for all" as a big prefix $\bigwedge$. See, the semantics of "for all" are really a big "and" operation: saying that all $x \in \{0, 1, 2\}$ have $f(x) = g(x)$ is the same as saying $f(0) = g(0)$ and $f(1) = g(1)$ and $f(2) = g(2)$. In other words:

$$\bigwedge_{x \in \{0,1,2\}} (f(x) = g(x)) \Leftrightarrow (f(0) = g(0)) \wedge (f(1) = g(1)) \wedge (f(2) = g(2)).$$

This is attractively parallel to

$$\bigcap_{x \in \{0,1,2\}} S_x = S_0 \cap S_1 \cap S_2.$$

The starting point of algebraic logic is saying that $\wedge$ isn't just parallel to $\cap$ but, when interpreted correctly, the same thing.

See how I've veered off topic here? Proof-assistant tutorials are full of this. You're trying to find out how to use a proof assistant to check your proofs, and you're faced with tutorials written by people who can't stop talking about logic.

Anyway, my experience was that it was easy to write definitions and theorems in the HOL Light syntax—except that at the very beginning I didn't understand the easy way to express a permutation of $\{0, 1, \ldots, n-1\}$ in HOL Light. The following paragraphs explain the issue and the solution.

As a student, I learned that a function from set X to set Y was formally defined as its graph: a subset $f \subseteq X \times Y$ such that, for each $x \in X$, there is a unique $y \in Y$ with $(x, y) \in f$. Of course, this unique y is denoted $f(x)$.

HOL Light supports sets, but its default concept of function, including the concise notation $f(x)$, is defined differently, and applies only to the special case that X is what's called a "type". This matters here because $\mathbb{N} = \{0, 1, 2, \ldots\}$ is a "type" in HOL Light but $\{0, 1, \ldots, n-1\}$ isn't. It was clear how I could define a permutation of $\{0, 1, \ldots, n-1\}$ as a graph and prove theorems about that, but then I wouldn't have the concise function notation. At best I would have something like `eval f(x)`, adding load to readers checking my theorem statements—what's `eval`? I'd have to show readers how I was defining `eval` and the underlying graphs, and convince readers that I hadn't accidentally made vacuous definitions or otherwise misdefined things.

I sent email to the relevant mailing list asking for advice. The HOL Light author explained that I could model a permutation of $\{0, 1, \ldots, n-1\}$ as a permutation of $\mathbb{N}$ that fixes $\{n, n+1, \ldots\}$. Aha!

One can also work with the larger class of functions from $\mathbb{N}$ to $\mathbb{N}$ that permute $\{0, 1, \ldots, n-1\}$. In general, one can define concise-notation functions on any subset of a type by simply defining them on the full type—with whatever behavior outside the subset—and then considering how they behave on the subset. For the case of permutations, it's particularly clean to specify the behavior outside the subset as the identity map, since the theory then factors into a theory of fixing subsets and a subset-independent theory of permutations. I'd still be happier just talking about permutations of $\{0, 1, \ldots, n-1\}$, but tossing in an identity map on the rest of $\mathbb{N}$ isn't a big complication.

2.2. Writing proofs. The proofs I looked at in the HOL Light library had many weird names of many different proof-construction tools, but there were also many short proofs that structurally looked like simply saying "this follows from these previously proven theorems". For example:

```
let MOD_EQ_SELF = prove
 (`!m n. m MOD n = m <=> n = 0 \/ m < n`,
  MESON_TAC[MOD_ZERO; MOD_LT; DIVISION; LE_1]);;
```

The first line is giving a name to this theorem. The second line is the theorem statement. The third line is naming some previous theorems in the library that together imply this theorem:

- `MOD_ZERO` says $n \bmod 0 = n$;
- `DIVISION` says that if n is nonzero then $m = \lfloor m/n \rfloor n + (m \bmod n)$ and that $m \bmod n < n$;
- `LE_1` says that $n \neq 0$, $0 < n$, and $1 \leq n$ are equivalent;
- `MOD_LT` says that if $m < n$ then $m \bmod n = m$.

`MESON_TAC` was used in many of these proofs, and was clearly some sort of general-purpose tool that automatically figures out how to put the previous theorems together with basic logic. Here's what it figures out in this example: either $n = 0$, done, or $n \neq 0$, in which case $m \bmod n < n$ from `DIVISION`, so if $m \bmod n = m$ then $m < n$ as claimed; for the converse, substitute m for n in `MOD_ZERO` to see that $m \bmod 0 = m$, i.e., $m \bmod n = m$ as claimed if $n = 0$; and use `MOD_LT` directly to see that $m \bmod n = m$ as claimed if $m < n$.

When HOL Light checks this proof, it also prints out some numbers indicating how deeply `MESON_TAC` is thinking about this:

```
0..0..2..10..39..151..solved at 158
0..0..solved at 2
0..0..1..solved at 4
```

The $158 + 2 + 4$ computations aren't expensive; this example runs in a tiny fraction of a second.

There are limits—`MESON_TAC` won't magically write a long proof for you—but you can simply split theorems into baby steps and ask `MESON_TAC` to handle each step. You have a theorem with hypotheses A, B, C where the proof draws intermediate conclusion D from A, B, and then intermediate conclusion E from C, D, and so on; so, okay, write a first theorem saying that A, B imply D, and then a second theorem saying that C, D imply E, and so on, and for each theorem give `MESON_TAC` the list of relevant definitions and lemmas.

That's exactly what I did in my first proofs. There was a lot of copying and pasting of formulas and hypotheses and conclusions, very quickly producing a very large number of very short proofs. For example, one of my first computer-checked proofs was as follows:

```
let xor1xor1 = prove(
  `!n. xor1(xor1 n) = n`,
  MESON_TAC[xor1xor1_ifodd;xor1xor1_ifeven;EVEN_OR_ODD]);;
```

I had defined `xor1` as follows:

```
let xor1 = new_definition
  `xor1 (n:num) = if EVEN n then n+1 else n-1`;;
```

HOL Light defines `num` as $\{0, 1, 2, \ldots\}$; defines `EVEN` and `ODD`; and proves a theorem `EVEN_OR_ODD` saying that each $n \in \{0, 1, 2, \ldots\}$ is even or odd. The `xor1xor1_ifeven` and `xor1xor1_ifodd` used in the above proof are lemmas that I had proven, using `MESON_TAC` and various further lemmas.

I learned, and started using, a little bit more than `MESON_TAC`. For example, `ARITH_RULE` automatically proves various linear equations and inequalities. If

you type `ARITH_RULE `n <= m /\ m < n + 1 ==> m = n`` then `ARITH_RULE` proves this for you and immediately gives you back a lemma that you can use inside `MESON_TAC`. If you want `0 < n ==> 1 <= n`, you don't have to remember or look up that this is already available as part of `LE_1`; you can type `ARITH_RULE `0 < n ==> 1 <= n``.

I also started getting some idea of how much I could feed to `MESON_TAC` at once. For unfolding definitions, it's better to use `REWRITE_TAC`, a tool that replaces L with R if it's given a definition (or lemma) saying $L = R$; `MESON_TAC`'s explorations, given the same equation, can at any moment try replacing L with R or replacing R with L, which is more flexible but also takes more time. Also, `MESON_TAC` has trouble figuring out what to substitute into a theorem's variables if the right substitution isn't staring it in the face; there's something called `ISPEC` that lets you say the substitution explicitly.

I wrote a few longer proofs. Longer proofs have the advantage of avoiding the copy-and-paste time of splitting out separate theorems—and also the naming time; `MESON_TAC` has a variant `ASM_MESON_TAC` that automatically uses all the intermediate conclusions obtained in the proof so far, with no need to give them separate names.

On the other hand, longer proofs have a disadvantage for beginners: you need to know more `TAC` names for specific types of proof steps. For example, in the `xorxor1` case, `MESON_TAC` was given lemmas saying

```
!n. EVEN n ==> xor1(xor1 n) = n
```

and

```
!n. ODD n ==> xor1(xor1 n) = n
```

and

```
!n. EVEN n \/ ODD n
```

and had no trouble figuring out how to put them together. Now imagine that, instead of splitting out separate lemmas for the even and odd cases, I had decided to write a combined proof of

```
!n. xor1(xor1 n) = n
```

saying, first, consider any particular n, and then note that that n is even or odd, and here's a proof of the even case, and here's a proof of the odd case. I would then need to know that "consider any particular n" is `GEN_TAC` and that "split into cases" is `DISJ_CASES_TAC`. For beginners, it's easier to just copy and paste the desired theorem twice to make two lemmas, insert `EVEN n ==>` and `ODD n ==>` into the two lemmas, prove those lemmas separately, and let `MESON_TAC` figure out how the main theorem follows from the lemmas.

2.3. Speed variations. For most forms of writing, I'll check everything I write two or three times, trying (not always successfully—see Section 3.4) to eliminate all mistakes:

- First, I'll check each piece soon after I've written it—for example, checking a proof, or a segment of a long proof in progress—because sometimes a mistake starts me down a very wrong path, and I'll waste less time if I catch this sooner. (Of course, most mistakes are eliminated during the pre-writing process of theorem discovery and proof discovery, but not all. It's important to write and check full proofs.)
- Second, I'll read through everything again at the end, in case something slipped through the earlier checks.
- Third, if I make a change—for example, weakening a hypothesis—then I'll recheck everything flowing from that. Sometimes this can go through more rounds. I'll also do a broader third check if the second check was catching more than an occasional mistake.

All of this hand-checking time disappears for computer-checked proofs. The computer has checked the proof; I don't need to check it again. This isn't like testing software and worrying that there are issues with untested cases. The proof is done.

Sometimes I want a variant of a previous computer-checked theorem. Okay: copy and paste, glance at the proof, tweak something, computer checks the tweaked theorem, done. Rechecking a proof after modifying a hypothesis is *much faster than doing the same thing by hand*. There are also some specific types of proofs that are generated by automated tools much more quickly than a human can type a traditional-format proof—for example, various manipulations of long formulas.

But I also encountered all sorts of random slowdowns. Sometimes I feed a proof step to `MESON_TAC`, and it sits there for seconds searching and not finding a proof. Hmmm. I divide the step into *really* baby steps, and it gets through the first step but sits there on the second. Hmmm. I check the list of lemmas that I provided to `MESON_TAC`. Oops, it turns out that I named lemma X where what I really wanted was lemma Y. Problem solved, but I've lost a minute on this, which is much more time than I was spending on a typical proof step.

This start-and-stop feeling—an exciting minute successfully producing many computer-checked lines at high speed, the next minute staring puzzled at the computer output and making vastly slower progress—is not what I had expected to encounter. What I had heard about computer-checked proofs, before I started writing them, gave me the impression of consistent drudgery.

Consider, e.g. de Bruijn [27] reporting the "*constancy of the loss factor*" (italics in original). This "loss factor", now known as the de Bruijn factor, is the length of a computer-checked proof divided by the length of the original human proof. "Constancy" says that the de Bruijn factor "does not increase if we go further in the book". This refers to experiments with computer checking of books presenting "very meticulous 'ordinary' mathematics"; the length of computer-checked proofs was observed to scale approximately linearly with the length of the traditional-format theorems.

My experience is that the length ratio between computer-checked proofs and my trying-to-be-meticulous hand-written proofs depends very much on how

much copying and pasting I happen to do. This length has very little importance; what matters is the *time to write computer-checked proofs*. The writing time varies tremendously from one step to another.

Many of the slowdowns I encountered were clearly artifacts of the tools I was using. For example, if you take a lemma in HOL Light about arbitrary functions f and then say `ISPEC` `` `g` `` to specialize the lemma to the particular function g appearing in a proof, then HOL Light will complain. The problem here is that HOL Light is parsing `` `g` `` independently of the current proof context, and has no idea that `g` is meant as a function; you instead have to type something like `` `g:X->Y` `` for the appropriate `X` and `Y`. Diagnosing and fixing a syntax issue like this would often cost me much more than a minute, and there are many of these syntax issues.

Mathematicians proving that functions f and g are equal by proving that $f(x) = g(x)$ for each x, or proving that sets S and T are equal by proving that $x \in S$ if and only if $x \in T$, or proving that $\sum_{x \in S} f(x) = \sum_{x \in S} g(x)$ by proving that $f(x) = g(x)$ for each $x \in S$, don't comment on the deduction principles being used. `MESON_TAC` doesn't handle these things automatically: you have to tell it to use `EQ_EXT` or `EXTENSION` or `SUM_EQ`.

Even in cases where I had simply forgotten to list a relevant lemma, I found myself wondering about ways that a proof assistant could have found the missing lemma automatically, or at least given me more mathematically meaningful error messages (as requested in [**6**, page 67]) so that I could immediately diagnose the problem. HOL Light and other proof assistants support interactive proof construction where, for any particular theorem you're working on proving, you can try applying one lemma after another and seeing at each step what the proof assistant says still has to be proven—but, when a lemma application fails, I often have to look very closely at the lemma to figure out why.

I don't mean to exaggerate the difficulties here. Each of my projects to write computer-checked proofs has succeeded in an affordable amount of time; see Section 3. People building proof-checking tools have done amazing work to bring the tools to their current levels of usability. Papers are continuing to appear on further improvements, and I would guess that by the end of this decade automatic tools will able to do most, if not all, of what I've done by hand. But maybe not, and in any case I don't see this as a reason to delay doing something that's already affordable and useful today.

3 Time to write computer-checked proofs

As noted in Section 1, I now have computer-checked proofs online for four papers. This section summarizes what was proven, the lengths of traditional-format proofs, and the amount of time spent writing computer-checked proofs.

The time metric is latency, reported in more detail as start dates and end dates. Latency puts an upper bound, not a lower bound, on the minutes spent; I don't have records of the minutes spent, and I was also doing other things during

the same periods. But I do think the latency is highly correlated with the actual time that I spent.

3.1. Control bits. Given "control bits" $c_0, c_1, c_2, c_3, c_4, c_5 \in \{0, 1\}$, consider the permutation of $\{0, 1, 2, 3\}$ built as follows:

- step 0: exchange 0 and 1 if $c_0 = 1$, and exchange 2 and 3 if $c_1 = 1$;
- step 1: exchange 0 and 2 if $c_2 = 1$, and exchange 1 and 3 if $c_3 = 1$;
- step 2: exchange 0 and 1 if $c_4 = 1$, and exchange 2 and 3 if $c_5 = 1$.

Any permutation of $\{0, 1, 2, 3\}$ can be obtained from some sequence of control bits.

More generally, any permutation of $\{0, 1, \ldots, 2^m - 1\}$ is the result of applying $(2m - 1)2^{m-1}$ control bits, where the first 2^{m-1} control bits perform swaps at distance 1, the next 2^{m-1} control bits perform swaps at distance 2, the next 2^{m-1} control bits perform swaps at distance 4, etc., the distances moving up and then down through powers of 2. Fast parallel algorithms to convert a permutation into a sequence of control bits were introduced by 1981 Lev–Pippenger–Valiant [**79**] and independently 1982 Nassimi–Sahni [**86**], after a fast serial algorithm that 1968 Waksman [**105**] credited to Stone.

It takes effort to see why the Lev–Pippenger–Valiant and Nassimi–Sahni algorithms work—for example, to understand why the Nassimi–Sahni algorithm always produces the same control bits as the Stone algorithm while the Lev–Pippenger–Valiant algorithm often doesn't. I wrote a paper [**13**] to give *formulas* for Stone's control bits in terms of a permutation π, and to prove that these control bits produce π. This reduces the verification of an alleged control-bit computation via, e.g., the Nassimi–Sahni algorithm to a verification that the computation matches these formulas.

Aside from computer checking, the paper is 18 pages: 3 pages of introduction, 4 pages of definition-theorem-proof in brutalist style (this is the part I turned into computer-checked proofs), 6 pages tracing the history and surveying optimizations, 3.5 pages regarding software, and 1.5 pages of references. Computer checking is covered in an appendix to the paper:

- a page discussing translation into HOL Light,
- a table indexing the computer-checked definitions and theorems, and
- 47 pages displaying the 2150 lines that the computer checked.

Let me emphasize that the numbers 47 and 2150 here are terribly misleading: these 47 pages of computer-checked material were *much faster to write* than a normal 47-page paper would have been. I started writing the computer-checked proofs on 9 September 2020, and posted the paper with computer-checked proofs on 23 September 2020.

I made sure to report the speed in the appendix: "the human time to write the proofs below was an unrecorded fraction of a two-week period". For the next version of the preprint, I'm planning to provide the computer-checked proofs as an attachment to the PDF rather than as pages in an appendix. To be clear, I do think computer checking of the proofs is an important part of the paper: "verified" is the first word of the title.

3.2. Asymptotics of a heuristic model of lattice security. I'm using "lattice" in this report to mean a discrete subgroup of $\mathbb{R}^n$, not a meet-join algebra. Say you're given a lattice $L \subseteq \mathbb{R}^n$ of rank n, and you want to find an element $v \in L$ given a nearby vector $v + \epsilon$.

There are various cryptosystem proposals over the past several years saying that solving large sizes of this problem requires time $(3/2)^{\beta/2}$, where β and κ are chosen with minimal $\beta \geq 60$ subject to $((n+\kappa)s^2+1)^{1/2}/(d/\beta)^{1/2}\delta^{2\beta-d-1}q^{\kappa/d}$ being below 1. Here $d = n+\kappa+1$; $\delta = (\beta(\pi\beta)^{1/\beta}/(2\pi\exp 1))^{1/2(\beta-1)}$; and s, q describe the sizes of ϵ, L. See [17] for full definitions.

There's no hope of proving this. It's a model based in part on hoping that a known algorithm is close to optimal, in part on heuristics for the time taken by the main algorithm steps, and in part on heuristics from [4] that lead to the formulas displayed above saying which algorithm parameters (β, κ) will succeed.

A closer look shows another risk for users of the cryptosystems. Someone attacking these cryptosystems is actually faced with multiple instances of the same problem for the same L, and might be able to exploit this:

- Perhaps there's an algorithm that finds multiple targets $v_0, v_1, \ldots \in L$, given $v_0 + \epsilon_0, v_1 + \epsilon_1, \ldots$, more efficiently than repeated applications of a single-target algorithm.
- Perhaps there's an algorithm that finds one of many targets more efficiently than picking the first target and applying a single-target algorithm.

Interesting multi-target speedups have been found for various other algorithmic tasks. However, for this lattice problem, the literature expresses a belief that an algorithm finding one of many targets has to take as much time as a single-target algorithm. A 2021 paper [43, page 3] claims to be able to prove that this belief follows from a preexisting conjecture.

An area packed so densely with conjectures and heuristics and beliefs and hopes is a culture shock for mathematicians prioritizing proofs, but shouldn't be surprising. Cryptographers trying to evaluate the cost of attack algorithms don't have the luxury of insisting on provability—in particular, of ignoring unproven speedups that are experimentally observed to work. There's a long history of unproven speedups for, e.g., integer factorization; see [18, Appendix B] for a survey.

Anyway, the main point of my paper [17] is that, if you apply the same heuristics to a particular algorithm for finding one of many targets, then you end up with asymptotically better performance than the best single-target algorithms known. Either the existing heuristics are failing, or the aforementioned belief is wrong, or both.

The asymptotic algorithm analysis isn't easy. Why shouldn't readers assume that, no, the previous literature is fine, and that there's simply some mistake buried inside my asymptotic calculations? The presence of asymptotics in my main statement also puts limits on how convincing an experimental double-check can be. Proving the accuracy of my analysis is critical for my paper, even in an area that obviously doesn't insist on proofs.

I posted the first public version of [17] in November 2022, including 4 pages with a sketch of my main asymptotic result. Part of this, 2.5 pages, was a proper theorem and proof, but that theorem covered only asymptotic analysis of the formulas above *assuming* β and κ have a particular asymptotic shape. The other 1.5 pages were summarizing how to asymptotically optimize β and κ, including summarizing how the optima can be proven to fit the asymptotic shape that I had analyzed. Of course, I wasn't using the word "theorem" for the second part.

After posting that version, I typed up a full proof of a theorem on the bottom-line asymptotics for optimal parameters, filling in all the details that were missing from the original sketch. The final theorem statement by itself is over half a page, and the proof in traditional format is a total of 13 pages.

I then wrote a computer-checked version of the proof, again using HOL Light. I started on 21 February 2023 and finished on 15 March 2023. Over the next two days I added an appendix to my paper regarding the computer-checked proofs:

- 4.5 pages going line by line through comparing the paper's main theorem statement to the computer-checked version of the statement (during these days I tweaked the computer-checked version a bit to bring the statements closer together, and, needless to say, asked the computer to check again);
- a page discussing the proof-writing process, in particular saying that I wrote this proof during a 3.5-week period; and
- a page explaining exactly how to re-run the computer checking.

I posted the computer-checked proofs and a revised preprint on 17 March 2023. I didn't post the 13 pages of traditional-format proofs, but I'm planning to post them as part of the next revision.

The computer-checked proofs occupy 9951 lines. This number is misleading not just for the reasons explained in Section 2, but also because many of the lines were actually produced by a small Python script that I wrote. The script takes a formula as input and does some easy computer algebra to automatically print out a proof, in HOL Light format, of first-order and second-order asymptotics for the formula—using various background lemmas that I proved by hand, such as anything in $f(A_0 + (A_1 + o(1))/X)$ being in $f(A_0) + (A_1 f'(A_0) + o(1))/X$ if f is continuously differentiable at A_0, where the $o(1)$ is as $X \to \infty$. There were also various other parts that I wrote by hand for the optimal-shape reasoning, but everything together was affordable—again, just a few weeks.

3.3. Speed of a gcd algorithm. Given integers f, g with f odd, recursively define δ_n, f_n, g_n as follows:

$$\begin{aligned}(\delta_0, f_0, g_0) &= (1/2, f, g);\\ (\delta_{n+1}, f_{n+1}, g_{n+1}) &= (1 - \delta_n, g_n, (g_n - f_n)/2) \qquad \text{if } \delta_n > 0 \text{ and } g_n \text{ is odd};\\ (\delta_{n+1}, f_{n+1}, g_{n+1}) &= (1 + \delta_n, f_n, (g_n + (g_n \bmod 2) f_n)/2) \qquad \text{otherwise.}\end{aligned}$$

It's easy to see that $\gcd\{f, g\} = |f_n|$ if $g_n = 0$. What the computer-checked proof from [16] says is that g_n reaches 0 within a particular number of iterations

determined by the initial sizes of f, g: specifically, if $0 \le g \le f \le 2^b$ and $9437b + 1 \le 4096m$, then $g_n = 0$ for some $n \le m$.

Yang and I had a paper [20] in 2019 introducing this type of gcd algorithm and, with $\delta_0 = 1$, proving a bound 25.1% worse than the above bound. The improved bound in [16] comes in part from Maxwell and Wuille finding a different way to prove the behavior of any particular m, in part from them finding that $\delta_0 = 1/2$ is better than $\delta_0 = 1$, and in part from our finding a way to replace the sequence of proofs for each m with a single proof handling all m at once. Hopefully we'll have a paper online soon explaining all the improvements over [20]; in the meantime, readers can already find various online descriptions and software and the computer-checked proof.

All of these proofs, starting with [20], involve large computer calculations. A second computer-checked proof from [16] involves computations on particular convex hulls with more than 100 points over a particular number field of degree 108, and improves the ratio 9437/4096 to

$$-\log_2 \left(\left(\frac{1591853137 + 3\sqrt{273548757304312537}}{2^{55}} \right)^{1/54} \right),$$

which appears to capture the actual behavior of the algorithm on worst-case inputs. The 9437/4096 result is only marginally weaker and involves considerably less computation, but still far more computation than one would want to carry out by hand.

This is not moving from a hand-checked traditional-format proof to a computer-checked proof. It is moving from a computer-assisted proof—this means a hand-checked reduction to a large calculation, plus hand-checked software to carry out that calculation—to a computer-checked proof, as in [53] and [59]. Consequently, I'm not sure what to report as the length of the original proof for comparison.

Here's the time I spent writing computer-checked proofs for this, all in HOL Light—not just the all-m proofs in [16], but also computer-checked versions of the earlier proof strategy handling any particular m:

- In 2021, I spent a week writing a script to generate a computer-checked proof for any particular m. I started 25 January 2021, and posted examples of computer-checked proofs for various m on 31 January 2021. Two weeks later, O'Connor and Poelstra [89] posted proofs for larger m verified in Coq.
- I returned to the any-particular-m proof on 18 March 2023, to see how far I could push m by optimizing the calculations inside the proof. On 23 March, I switched to writing all-m proofs. I finished on 6 April.

These stretches of time add up to 3.7 weeks. I then added a few minor tweaks. I posted the all-m proofs—actually, a 3711-line Sage script that prints out the proofs—on 16 April 2023.

3.4. Decoding binary Goppa codes. Fix nonnegative integers n, t; a finite field k containing $\mathbb{F}_2$; distinct elements $\alpha_1, \alpha_2, \ldots, \alpha_n$ of k; and a squarefree polynomial $g \in k[x]$ of degree t with $g(\alpha_1)g(\alpha_2)\ldots g(\alpha_n) \neq 0$.

A "Goppa codeword" is a vector $c \in \mathbb{F}_2^n$ satisfying $\sum_i c_i \alpha_i^j / g(\alpha_i) = 0$ for each $j \in \{0, 1, \ldots, t-1\}$. There are various algorithms in the literature that recover a Goppa codeword $c \in \mathbb{F}_2^n$ from $c + e$ for any $e \in \mathbb{F}_2^n$ with $\#\{i : e_i \neq 0\} \leq t$.

I wrote a paper [15] as a general-audience minicourse presenting the simplest known polynomial-time algorithm for this task. The core work here was optimizing the structure of theorems and proofs to get to this algorithm as efficiently as possible (for example, [15] doesn't need the Berlekamp–Massey algorithm) starting from almost nothing (the first theorem is what people call "Lagrange interpolation", although that's a misnomer—this was published by Waring [107] more than a decade before Lagrange).

I also presented various optional material, such as Sage test scripts, summaries of speedups in the literature, and a section about a cryptographic application. For that application, it's important not just to recover c from $c + e$ with c, e as above, but also to recognize whether the input in $\mathbb{F}_2^n$ has the form $c + e$ in the first place. After applying a Goppa-decoding algorithm to recover maybe-c and maybe-e from maybe-$(c+e)$, one can simply check whether c is a Goppa codeword and $\#\{i : e_i \neq 0\} \leq t$. It's easy to see other approaches from the literature; what sounds fastest involves checking a derivative formula due to Forney.

My computer experiments seemed to be saying that, actually, these checks don't do anything in the context of this Goppa-decoding algorithm. I didn't know this before, and I couldn't find it in the literature, and the coding theorists I asked hadn't heard about it. I tried hard to find counterexamples, and didn't find any. And then, aha, I figured out how to prove that obtaining a successful output from the algorithm already forced the right output of the Forney formula.

I added another section stating and proving this. I explicitly recommended against actually *using* this—"There are several reasons to recommend the second approach ... What happens if there's a mistake in the extra logic leading to Theorem 7.4, or in the handling of invalid inputs in the software implementing a decoding algorithm?"—but I was still pleased at the discovery, both for the unobvious mathematical content and for the thought that it could eventually be useful once software was verified.

I posted the paper in March 2022. In August 2022, Hovav Shacham contacted me and asked how I was obtaining a particular proof step. The answer, in short, is that there *was* a mistake in the logic. The proof was wrong. An entire lemma that I had stated—and that I had failed to test—was easily disproven. I still had ample reason to conjecture the conclusion of the main "theorem", but I hadn't proven it.

I promptly posted a revised paper with an erratum and a different, longer, trickier, I claim carefully hand-checked, proof of the main theorem—actually a more general theorem, skipping a hypothesis that had led to the shortcuts taken by the failed lemma.

Of course, you're wondering at this point whether another mistake could have slipped through. I'll skip the suspense: the theorem now has a computer-checked proof. I mentioned, in Section 1 of this report, verifying the theorems from one

paper using Lean and then also verifying them using HOL Light; that paper is [15]. Dates:

- I started writing Lean proofs on 12 July 2023 and posted them on 26 July 2023.
- I started writing HOL Light proofs on 31 July 2023 and posted them on 11 August 2023.

Over the subsequent week, I added an appendix to the paper with a more comprehensive discussion of proof-checker risks than I had put into previous papers—for example, looking in detail at how HOL Light and Lean define fields; Lean's answer is surprisingly complicated—and then 25 pages reviewing the paper's computer-checked definitions and theorem statements. I allocated a page for each theorem, putting the original theorem statement at the top of the page followed by the two computer-checked theorem statements annotated with translations into normal mathematical terminology. If this didn't fit then I used a separate page for each proof assistant. The point is that a reviewer can compare the theorem statements to annotations without flipping pages. I tweaked the theorem statements during the week to bring them closer together, as I had done for [17]. I posted the revised paper on 18 August 2023.

Deleting all text from [15] beyond the traditional-format proofs and theorem statements leaves 5.5 pages, about a third of that being the maybe-new definitely-theorem. The Lean proofs are 3632 lines—which, as usual, is terribly misleading; these lines were written during a 15-day period (aside from a few minor tweaks later). The HOL Light proofs are 9317 lines—which is even more misleading; these lines were written during a 12-day period.

I didn't write any scripts to print out proofs for [15]. I did a lot of copying and pasting. Running the final proofs through the standard `gzip` compression program (as suggested in [109]) reduces the Lean proofs from 194560 bytes to 34067 bytes and the HOL Light proofs from 411296 bytes to 57221 bytes. Those numbers don't tell the reader what ultimately matters: namely, compared to the time I spent coming up with theorem statements and traditional-format proofs for the paper, the time I spent writing computer-checked proofs—and then doing it again with another proof assistant!—was affordable.

4 Two proof assistants

> *People argue about the relative merits of the different systems much in the same way that people argue about the relative merits of operating systems, political loyalties, or programming languages.*
>
> —2008 Hales [57, page 1373]

> *HOL has no dependent types. Does this mean that there are entire areas of mathematics which are off limits to his system? I conjecture yes. Prove me wrong. . . . But do you people want to attract "working mathematicians"? Then where are the schemes?* Can your system

> even do schemes? *I don't know. Does anyone know? If it cannot then this would be very valuable to know because it will help mathematician early adopters to make an informed decision about which system to use.* —2020 Buzzard [**33**]

> *In this article we present a formalization of sheaves of rings and schemes in Isabelle/HOL reaching the benchmark result set by [6]: an affine scheme is a scheme. . . . It seems that our experience in Isabelle to formalize schemes was less tumultuous than the corresponding one in Lean . . . If, again, we gave a slightly different formulation for the definition of a scheme, following Hartshorne [16] instead of the Stacks project [23] as they did, it seems that this difficulty cannot possibly have a direct counterpart in Isabelle, since it arose from the difference between the so-called* equality types*, also known as* identity types*, and the* definitional equality*, a difference which is peculiar to dependent type theories.* —2022 Bordg–Paulson–Li [**25**]

As reported in Section 3, I wrote proofs checked by Lean for the theorems from [**15**] over a period of 15 days, and then proofs checked by HOL Light for the same theorems over a period of 12 days. I had three HOL Light projects before that. I also spent some time adding related comments to papers.

This section explains why I started with HOL Light in the first place, and then describes various aspects of how I noticed my time being used in Lean and of how I noticed my time being used in HOL Light. Among other things, this section gives concrete examples of how advantages and disadvantages of "dependent types" affected the speed of the proof-writing process for [**15**].

It seems necessary to emphasize, as a preliminary matter, that both tools were successful for [**15**]. This section is looking at small differences, not showstoppers.

4.1. Initial considerations. Here's why I picked HOL Light for my first project writing computer-checked proofs, out of several proof assistants that I noticed people frequently mentioning:

- HOL Light had already been used to successfully computer-check many previous proofs, including proofs relying on nontrivial mathematics, such as an analytic proof of the prime-number theorem. (See [**63**].) This made it seem very unlikely that there would be any big problems in feeding my own proofs to HOL Light.
- HOL Light's proof-checking kernel looked simpler and smaller than anything else I found. This, in conjunction with having nontrivial mathematics, made it seem very likely that the non-kernel part of the system had helpful tools for constructing serious towers of proofs: if you're starting from scratch then you *need* this construction process to be easy. This also reduced my concerns about the risk of kernel bugs allowing, perhaps even encouraging, incorrect proofs; see below.
- As Section 2 illustrates, explaining full details of a theorem statement written in HOL Light looked like a tolerable amount of effort: some foreign notation,

certainly, but nothing particularly complicated. I knew from the outset that I would want my readers checking the computerized versions of my definitions and theorem statements. See Section 4.2 below.

I wasn't at all sure that HOL Light was the best choice; it simply seemed the least risky overall.

Now that I have a bit of experience with proof assistants, I'm more worried about bugs. There's a comment in [7, page 74] saying "the proof checker keeps the formalizer honest, requiring every step to be spelled out in complete detail", and there are similar comments in, e.g., [57, page 1376] and [91]—but this doesn't match my experience.

Some types of proofs have been automated. I've sometimes used automation to successfully leap through a series of proof steps, and in those cases it's simply not true that my work in writing computer-checked proofs is a superset of the work I'm doing in writing traditional-format proofs. As more and more types of proofs are automated, it will become more and more common to rely on the proof assistant for steps that the human has never actually thought through. If there are bugs in proof-checkers, then presumably the automated tools will sometimes accidentally exploit those bugs, as in the case reported in [81, page 289]. The resulting pseudo-proofs have a competitive advantage, since the human wants to believe that the result is proven.

To be clear, when I'm using a proof assistant, I'm never thinking anything like "I'm working with Lean, bigger kernel, higher risk of bugs, let's slow down and check each step more carefully". I'm instead focusing on how to get the proofs done. The rest of this section presumes that proof-assistant kernels have been checked more carefully than other software and are bug-free.

4.2. Theorem readability. More and more papers are going to appear with computer-checked proofs. We'll start to see examples where what the paper says is wrong, and where the computer missed this because the computer was never asked to check the paper's theorem statement—it instead checked a different theorem statement, and nobody noticed the difference.

The main defense available today is a skeptical reader, hopefully starting with the paper's author, checking whether the theorems and underlying definitions match. Such a reader is slowed down by many discrepancies of syntax and semantics. Here are some examples visible in [15]:

- The reader doesn't expect `a/b` to be defined as $\lfloor a/b \rfloor$, and also doesn't expect it to allow $b = 0$. This is a dangerous choice of notation in Lean. Similar comments apply to some extent to HOL Light, although this is less visible in the theorem statements in [15] for reasons discussed below. There is a claim in [32] that "dividing by 0 is not allowed in mathematics, and hence this cannot be relevant to their work", but in fact a mathematician is permitted to deduce $b \neq 0$ from $c = a/b$, since a/b is undefined for $b = 0$. Redefining the notation to allow $b = 0$ breaks this. Unless it occurs to the author to state a conclusion $c = a/b$ *and* a conclusion $b \neq 0$, the proof assistant won't check that $b \neq 0$, whereas the reader will think that this *has* been checked.

- Many readers won't recognize HOL Light's `\/` or Lean's $\vee$, in part because of how unpopular the notation is in mainstream mathematics compared to "or" and in part because—depending on the font—Lean's $\vee$ looks almost identical to a lowercase v. (I spent a while searching for a font that I could use to display Lean theorem statements in [15] without any missing characters. DejaVu Sans Mono worked, but its $\vee$ and v are very similar. Perhaps I should try non-monospaced fonts.)
- Readers are confronted with many further uses of punctuation that aren't standard in mainstream mathematics, no matter how sensible the punctuation might seem to the designers of proof assistants. For example, Lean uses a combination of brackets and braces and parentheses for hypotheses (brackets for "type classes", braces for "implicit" hypotheses, parentheses for "explicit" hypotheses), uses a colon to separate hypotheses from conclusions in a theorem statement, and uses $\to$ (not $\Rightarrow$) for further implications. HOL Light uses `!` for "for all", uses `?` for "there exists", and uses `\x. x+1` as an ASCII approximation to $\lambda x.x+1$ where mathematicians normally write $x \mapsto x+1$.
- A ring formula in HOL Light is much more verbose than a ring formula in Lean. The expression `G = ring_mul(x_ring k) g g` in HOL Light takes much more time for the eye to scan than `G = g^2` in Lean or $G = g^2$ in normal writing. If this is a hypothesis and G isn't used very often then one normally eliminates the variable in favor of g^2, but this relies on the notation being concise. See Section 4.4 for the role of "types" here.
- Lean is more verbose than HOL Light in some distracting ways. For example, when Lean hypotheses are placed before the colon in a theorem statement (as the Lean documentation recommends) rather than used as implications after the colon, they're given names that aren't used elsewhere in the theorem statement. As another example, Lean keeps using the words `DecidableEq` and `noncomputable`; I do not think those mean what you think they mean.

For [13], I inserted comments into the computer-checked proofs aimed at introducing a general audience to the notation, but I think this way of organizing the material puts too much load on a reader trying to check the translation. For [17], I instead separately quoted the definitions and theorem statements in line-by-line comparisons, as noted above. For [15], I used a visual organization that I think works well for comparing theorem statements. These comparisons take time to write, and take time for every skeptical reader to read.

One of the virtues of having computer-readable definitions and theorems and proofs *should be* that you can have the computer automatically convert other people's notation into your favorite notation, and you can experiment with different converters to see what you like best, and in any case convert everything into Standard Do-Not-Annoy-The-Journal-Referees Notation for publication (why should the human have to write something in two languages?), so that there isn't a tension between optimizing notation and communicating effectively.

As a starting point, I would like a tool that converts a proof assistant's theorem statement to a traditional-format theorem statement and a comparison chart.

Tools such as [48] already demonstrate that much more than this is possible. See also [58] for discussion of the readability of theorems and proofs.

Some people are hoping to replace the usual language of mathematics with the language of a proof assistant (*this* one, not *that* one), eliminating the occasional ambiguities and avoiding any need for translation. (As an analogy, I think Sage is now familiar to so many readers that it's a better choice than English for algorithm statements.) However, in the short term, readers looking for theorem statements are generally looking for traditional-format theorem statements, so I can't omit those, and it's important to set up procedures ensuring that those are correct.

4.3. Filling in background. Each of my projects to write computer-checked proofs relied on many background lemmas that had already been proven, but also involved proving further background lemmas.

There's a cost in figuring out what has been proven already. Skimming libraries helps. Lean has put some work into context-sensitive search tools that can help: for example, you can try typing "`exact?`" as a proof step, and often Lean will immediately respond with the name in the library for exactly the lemma that finishes the proof.

Often it's faster to just go ahead and prove something than to figure out what's already there. As some trivial examples, one lemma I wrote for HOL Light for [15] says that if $m, n \in \mathbb{N}$ and $mn < n$ then $m = 0$, and one lemma I wrote for Lean for the same paper says that if $n \in \mathbb{N}$ is odd then $1 \leq n$. Writing a proof for each lemma is very fast. The only issue is the aggregate work—which, again, was affordable for each case in Section 3, but I think it's useful to say more about this. See also Section 5 for two examples of papers relying on more pieces of background.

Section 3.4 mentioned that feeding the proofs from [15] through `gzip` produces 34067 bytes for Lean and 57221 bytes for HOL Light. If I remove the parts that I think of as background then I obtain 21595 bytes for Lean and 26197 bytes for HOL Light.

Certainly a noticeable fraction of my Lean time for [15], and a larger fraction of my HOL Light time for [15], was spent writing background proofs. For example, I easily found where HOL Light was developing some of the basic theory of multivariate power series and polynomials (extension of morphisms, for example, and polynomial rings over domains being domains), and more theory of polynomials over the reals, but I ended up defining degrees for univariate polynomials over arbitrary commutative rings and proving, e.g., that $\deg fg = \deg f + \deg g$ when the base ring is a domain.

Logically, the degree definition is also a prerequisite for the theorem statements in [15], so [15] presents it as a high-risk definition that needs review—for HOL Light. Lean already had a definition of polynomial degrees, presumably reducing the risk of mismatched definitions.

As another example, I proved in HOL Light that $k[x]$ was Euclidean, to plug into HOL Light's existing proofs that Euclidean domains are principal-ideal domains and hence Bézout rings, giving the coprimality facts that I needed. A

small compensation for not finding previous HOL Light development of division with remainder for $k[x]$ is that I also had no incentive to use the dangerous notation "/" for that operation.

As yet another example, I proved in HOL Light that n homogeneous linear equations in more than n variables over a domain have a nonzero solution, using the ancient row-reduction proof (Fangcheng, very long before Gauss). Lean already had the special case that [15] needed (the case of fields), so I simply used that—although actually this wasn't so simple, for weird reasons; see Section 4.7 below.

The numbers above also don't reflect the fact that Lean already had some development of interpolation, in part because of previous attention to [15]. See [26, Section 4]. If I had decided to reuse that then the Lean numbers would have been smaller.

The background I wrote in Lean for [15] wasn't exactly a subset of the background I wrote in HOL Light—for example, Lean's degree function produces results in $\{-\infty\} \cup \mathbb{N}$ as mathematicians expect, and I ended up proving various background facts in Lean about operations on $\{-\infty\} \cup \mathbb{N}$; for HOL Light, I used the number theorist's trick of defining $2^{\deg f}$ rather than defining $\deg f$, so I didn't need to handle $-\infty$ in the first place—but was definitely much less material overall.

4.4. The conciseness of dependent types. Let's look at what the fuss is about regarding "simple types" and "dependent types".

It would be a big waste of time to write anything like the aforementioned `G = ring_mul(x_ring k) g g` on a blackboard. In normal mathematical writing, if g is an element of a ring R (such as, in this example, the polynomial ring $k[x]$ over a field k), then we freely use the concise notation g^2 to mean the square of g using R's multiplication operation.

But, wait, is the previous sentence true? Apply the same claim to the one-element ring $\{g\}$, which certainly contains g, to conclude that $g^2 = g$, which is certainly not always the same as the square of g in R, contradiction.

The actual situation is that if we've *declared* that R is a ring containing g, or if this follows from previous declarations by various rules, then we freely use the concise notation g^2. One of the rules says that if g, h are declared to be elements of R then $g + h$ is automatically declared to be an element of R, so you're free to write $(g + h)^2$. Generations of students learn these rules by osmosis.

There *isn't* a rule saying that g is declared to be an element of $\{g\}$. You can, of course, make declarations about multiple rings, but if the declared ring elements overlap then you have to write things like $+_R$ and $+_S$ to disambiguate—except, of course, in the common situation that the ring structures have been declared to be compatible. Et cetera.

Proof assistants vary in how much support they have for the complications of common notation. HOL Light defines $g + h$ if you've declared that g and h are integers, for example, but not if you've declared that they're elements of a more general commutative ring R. There are various mechanisms to modify the HOL

Light syntax, but the limited amounts I know about those mechanisms aren't useful here.

When you tell HOL Light or Lean that g has "type" $\mathbb{Z}$, written "g : int" in HOL Light or "$g : \mathbb{Z}$" in Lean, then the system knows that g is an element of $\mathbb{Z}$, and that you're allowed to write $g + g$ to refer to the sum in $\mathbb{Z}$ of g and g. This improves conciseness not just for formulas but also for proofs: the system automatically knows that $g + g$ also has type $\mathbb{Z}$.

To work with more general commutative rings in HOL Light, I was using less concise notation involving `ring_add` etc., and I had various proof steps referring to, e.g., HOL Light's trivial `RING_ADD` lemma, which says that $g + h \in R$ if $g, h \in R$. I found this verbosity less time-consuming than one might expect, for two reasons:

- Writing on a computer is not the same as writing on a blackboard. Copy and paste generates boilerplate very quickly.
- HOL Light has automated tools for proving identities: as a small example,

```
RING_RULE `ring_mul r a b =
ring_add r (ring_mul r a c) (ring_mul r a (ring_sub r b c))`
```

  automatically proves that if $a, b, c \in R$ then $ab = ac + a(b - c)$.

But there was certainly *some* slowdown from the formulas being longer. For example, one time something wasn't working and it eventually turned out that I had a `ring_add` where I should have had a `ring_sub`, which I would certainly have spotted more quickly if the formulas had been more concise.

Why does Lean support R as a type while HOL Light doesn't? This is an example of Lean supporting "dependent types" while HOL Light doesn't.

Lean has, for example, a name `Fin n` for the type $\{0, 1, \ldots, n-1\}$. To be more precise, the type is a copy of $\{0, 1, \ldots, n-1\}$. To be even more precise, an element of the type is a pair (m, h), where m is an element of $\mathbb{N}$ and h is the fact that $m < n$. To be more precise in a different direction, `Fin n` is a name for a parameterized family of types, where the parameter, n, ranges over a type, namely $\mathbb{N}$.

HOL Light also has some names for parameterized families of types. For example, the type `X->Y` is the set of functions from X to Y (and the type supports the syntax `f(x)` for applying such a function to an element x of X). To be more precise, `X->Y` is a name for a parameterized family of types, where the two parameters, X and Y, range *over the set of types*.

HOL Light doesn't have any way to define a type family with a parameter that ranges *over a type*. You can build a type in HOL Light that's (a copy of) $\{0, 1, 2\}$, but if n is a variable in a theorem then you have no way to make a $\{0, 1, \ldots, n-1\}$ type, and if a subset R of X is a variable in a theorem then you have no way to make an R type. If you want to work with functions on R in HOL Light, it's easiest to work with functions on X instead. This is why, in Section 2.1, I ended up working with permutations of $\mathbb{N}$ instead of permutations of $\{0, 1, \ldots, n-1\}$.

4.5. Problem between chair and keyboard. The obvious question at this point is why the theorems from [15] took me 15 days to write in Lean and only 12 days to write in HOL Light, despite HOL Light needing more background development and being less concise.

One generic answer is that second formalizations are faster than first formalizations. As noted in Section 1, most people explaining a proof to a computer would first have to take time to understand the proof—which is often a time-consuming task (see, e.g., [63, Sections 5 and 8]), especially when the "proof" isn't really a proof but rather a sketch of a proof strategy. Sometimes the "theorem" isn't even clearly stated, and one has to figure out what the "proof" is supposed to be proving. Someone who goes through this preliminary work as part of writing a computer-checked proof can then explain the proof to another proof assistant without repeating the preliminary work.

But I had already written detailed proofs for the previous version of [15] and knew exactly how they worked, along with all of the background. Furthermore, around half of the material that I ended up proving for HOL Light (see Section 4.3) was background material that I *hadn't* proved for Lean, since for Lean I was reusing existing lemmas. So I don't think the second-formalization effect is what's going on here.

Another obvious answer comes from differences in experience levels. I'm certainly nowhere near being a HOL Light expert, but I'm even farther from being a Lean expert. All I did with Lean before [15] was spend a few days listening to introductory Lean talks and solving a bunch of Lean exercises and watching how various people were using Lean.

I think this difference was a contributing factor—for example, I did need some time to get an idea of what background lemmas were available in Lean—but far from a complete explanation.

I'm faster with HOL Light now than I was when I first started, but the only big speedups have been in situations where I've been able to take better advantage of automation; see below. My initial copy-and-paste process from Section 2 was verbose but reasonably fast.

I had worked with HOL Light in other ways by the time I started with Lean, and the process I saw people using in Lean looked very much like one of the approaches I was familiar with from HOL Light. There were superficial syntax differences: `rw` instead of `REWRITE_TAC`, for example, and `sum_congr` instead of `SUM_EQ`. (I do prefer lowercase; the HOL Light proofs in [15] define various synonyms such as `let rw = REWRITE_TAC`.) There was also a notable absence of anything that looked like `MESON_TAC`—people were constantly working with lower-level tools for explicitly manipulating the logic, which I found surprising. I knew analogous logic tools in HOL Light and was using them in much the same way that a traditional-format proof sometimes says things like "There are now two cases", but the normal "so" and "thus" and "therefore" proof steps are trivially handled by `ASM_MESON_TAC`.

Automation can have a striking impact. Any computer-mathematics system can immediately compute derivatives, but it's even better to type

```
DIFF_CONV `\x. (ln(x) + ln(pi*x)/x - ln(&2 * pi * exp(&1)))
               / (&2 * (x - &1))`
```

into HOL Light and immediately get back a *theorem* stating the derivative of that expression under suitable conditions on x. This sort of thing makes it seem defensible that they're called "proof assistants" rather than "proof nitpickers". I definitely saved time in [17] with such tools—including my own small asymptotic-analysis script, which I think can be cleaned up for broader use. New papers are continually appearing on proof automation, saving more and more time for proof-assistant users.

Automation had a smaller role in [15] but was certainly still present. I used HOL Light's `RING_RULE`, for example; I had also used `ring` in Lean. `MESON_TAC` is also a small-scale example of automation; I was later told that supporting this in Lean was an ongoing research project. Presumably I missed some relevant automation that would have saved time for me in Lean, and presumably I also missed some relevant automation that would have saved time for me in HOL Light.

4.6. Translation steps in proofs. There's something else where I *know* I was frequently losing time with HOL Light—and losing more time with Lean.

Mathematics often has multiple ways to express the same thing. Saying that $(1+x)^{m+n} = (1+x)^m(1+x)^n$ in $\mathbb{Z}[x]$, for example, is equivalent to saying that the coefficient of x^d matches for each d, which, by the binomial theorem, is equivalent to saying $\binom{m+n}{d} = \sum_{a+b=d} \binom{m}{a}\binom{n}{b}$, which you can alternatively prove by observing the bijection between (1) ways to choose d items out of $m+n$ items and (2) ways to choose a, b with $a+b=d$, choose a items from the first m, and choose b items from the remaining n.

There's some redundancy in having things stated two ways. Some proofs are easy either way. But some proofs are easier one way, and some proofs are easier the other way, with each perspective bringing its own useful extra structure.

Unless proof assistants provide completely automatic translations, lemmas end up sometimes being stated just one way, and sometimes being stated just the other way, because people haven't done the extra work to provide both statements. If I have a proof step using a lemma stated one way, and then a proof step using a lemma stated another way, then I need to insert a translation step in between. Each translation step is easy, but these translation steps aren't as mindless as "I'm using $g+h$ so prove it's in the ring" boilerplate; they're driven back and forth by how the "real" proof steps happen to be stated.

This translation cost makes it useful to merge multiple perspectives in cases where multiple perspectives don't offer much extra value. For example, in HOL Light, there's no distinction between a subset of X, a function from X to $\{\text{True}, \text{False}\}$, and an X-indexed vector with entries in $\{\text{True}, \text{False}\}$. Great—sounds like there won't be a split of lemmas across these three ways to say the same thing. But then HOL Light somewhat spoils this unification by defining `x IN S` to mean the same as `S(x)`, with various lemmas stated in terms of `IN`, requiring translation steps (e.g., "`SET_TAC`"). Section 4.7 gives an example of how translations slowed me down more for Lean.

In set theory, typically 3 is defined as $\{0,1,2\}$, and Y^X is defined as the set of functions from X to Y, so Y^3 is the set of functions from $\{0,1,2\}$ to Y. A common way to advertise proof assistants that emphasize type theory rather than set theory is to claim that the statement $2 \in 3$ is a "fake theorem". I don't know whether the following theorem in HOL Light is also supposed to be "fake": $\{0\}(163) = \text{False}$, since $\{0\}$ is actually a function, specifically the function from $\mathbb{N}$ to $\{\text{True}, \text{False}\}$ that's False except for input 0. I do know that unified descriptions of mathematical objects saved time for me in [15], since the lemmas regarding those objects plugged directly into each other without translation steps.

4.7. The curse of dependent types. Fix a field k, two Laurent series $A, B \in k((x^{-1}))$ with $\deg A > \deg B$, and a nonnegative integer t. Then there are coprime $a, b \in k[x]$ with $\deg a \leq t$, $\deg b < t$, and $\deg(aB - bA) < \deg A - t$.

This is the second theorem in [15], except that the theorem statement in [15] is only for polynomials $A, B \in k[x]$, since this requires a bit less background to state and is good enough for the paper. The theorem was originally proven by Lagrange [76] as follows: the goal is to find a rational function b/a with small numerator and denominator very close to B/A; now apply the theory of continued fractions.

The proof in [15] is a much more direct proof due to Kronecker [75, pages 118–119 of cited PDF] without continued fractions. The proof occupies ten lines in [15] but here's the essence of it: a, b together have $2t+1$ degrees of freedom, but clearly $\deg(aB - bA) < \deg A + t$, so asking for $\deg(aB - bA) < \deg A - t$ is imposing $2t$ linear constraints, so there's a nonzero solution; to finish, divide (a, b) by $\gcd\{a, b\}$. This is a classic example of theorems about function fields often being easier to prove than more general theorems about global fields.

Should be easy to convert this to a Lean-checked proof, right? As a starting point, I quickly found Lean's linear-algebra lemmas. The lemmas that sounded most obviously useful were `linearIndependent_iff_card_eq_finrank_span` and `finrank_le`, which together immediately imply that if V is a k-module of smaller rank than $\#S$ then any S-indexed sequence of elements of V is linearly dependent.

So, okay, define S as $\{0, 1, \ldots, 2t\}$, or maybe more naturally the direct sum of $\{0, 1, \ldots, t\}$ and $\{0, 1, \ldots, t-1\}$, to index the coefficients of a, b; define V as k^{2t}, or maybe more naturally $k^{\{\deg A - t, \ldots, \deg A + t - 1\}}$; and define various elements of V by extracting the right coefficients from A and B.

But, wait, it's not that simple. The linear-algebra lemmas aren't asking for a *set* S; they're asking for a *type*. Similarly, they aren't asking for this V to be the *set* of functions from an index *set* to k; they're asking for a *type*, in this case the functions from an index *type* to k.

Lean has a way to convert sets to types, as illustrated by the aforementioned `Fin`, which converts the set $\{0, 1, \ldots, n-1\}$ into a type `Fin n`. But `Fin n` isn't the same object as that set: it's a copy of that set, namely the set of pairs (m, h) where $m \in \mathbb{N}$ and h is the fact that $m < n$. The distinction is content-free

symbol-pushing: it's simply bundling elements of a set with the knowledge that they're elements of the set.

Are Lean's lemmas about $\{0, 1, \ldots, n-1\}$ stated as lemmas about this subset of $\mathbb{N}$, or lemmas about the type `Fin n`? The answers vary:

- The Lean linear-algebra lemmas that I looked at were stated in terms of finite types such as `Fin n`.
- The Lean lemmas about sums that I looked at were sometimes stated in terms of finite types such as `Fin n`, and were sometimes stated in terms of finite subsets such as $\{0, 1, \ldots, n-1\}$.
- Kronecker's proof considers the coefficient of x^i in a polynomial for $i \in \{0, 1, \ldots, n-1\}$. The Lean lemmas about polynomials that I looked at used natural numbers as exponents of x, not elements of `Fin n`.

My Lean rendition of the proof ended up translating back and forth between the type and the subset. There's a thicket of further types used in the proof; this `Fin n` issue is just one example of the slowdown.

HOL Light also has a type $\mathbb{N}$ and a set $\mathbb{N}$, but, because it doesn't support dependent types, it doesn't have a type version of $\{0, 1, \ldots, n-1\}$. This restriction is bad for conciseness if I want to use, e.g., the abbreviation "+" for addition modulo n on $\{0, 1, \ldots, n-1\}$. But this restriction tends to make lemmas easier to compose, because—for generality—lemmas tend to be stated about subsets, and then simply work together with no translation steps. I spent less time dealing with type conversions in the HOL Light version of Kronecker's proof than in the Lean version of Kronecker's proof. I was proving more background in HOL Light—including, as noted in Section 4.3, the k-linear-dependence lemma that this proof needed—but I was also spending less time in HOL Light than in Lean dealing with type conversions for the background, and for the other theorems in [15].

It's not that HOL Light magically merged everything. For example, a lemma might be stated in terms of $\{0, 1, \ldots, n-1\}$, or might be stated in terms of the hypothesis $m < n$, as in the mathematics literature. But I encountered more splits in Lean.

Lean frequently restates a hypothesis or a set as a type. Each such type definition provides another way to say the same thing. Some lemmas are stated using the hypothesis. Some lemmas are stated using the set. Some lemmas are stated using the type. Plugging one lemma into another often requires a translation step. My experience is that these translations are *easy* but not *automatic*. For example:

- It's easy to use Lean's `Finset.sum_range`, which, for a function f defined on $\mathbb{N}$, rewrites the sum of f over the subset $\{0, 1, \ldots, n-1\}$ of $\mathbb{N}$ as the sum over `Fin n` of f implicitly composed with the map from `Fin n` to $\{0, 1, \ldots, n-1\}$. The syntax for the composition is concise, but this lemma is still a translation step between different objects.
- It's also easy to use Lean's `Finset.sum_fin_eq_range`, which, for a function f defined on `Fin n`, rewrites the sum of f over `Fin n` as the sum over

$\{0, 1, \ldots, n-1\}$ of the function g defined on $\mathbb{N}$ as follows: if $m \in \mathbb{N}$ has $m < n$ then $g(m) = f((m, h))$, where h is the fact that $m < n$; otherwise $g(m) = 0$.

- If these are used automatically, then that's news not just to me but, it seems, also to the authors and reviewers of various proofs included in Lean that explicitly use these translation steps.

As another example of Lean restating hypotheses as types, Lean has a definition of a type `degreeLE` for the subset of $k[x]$ of degree at most t, and a type `degreeLT` for the subset of $k[x]$ of degree below t. Surely Lean will also add such types for the corresponding subsets of $k((x^{-1}))$, if they aren't there already, and will gradually acquire lemmas about these types.

Certainly there are interesting interactions between the algebraic structure and the degree bound, which is exactly why one sees them coming together in Kronecker's proof and many other proofs. One can reorganize Kronecker's proof into a series of proofs about these types:

- There's not just B in $k((x^{-1}))$, but B in the subset of $k((x^{-1}))$ of degree below n, where $n = \deg A$.
- The multiplication aB is then mapping the subset of $k[x]$ with degree at most t times the subset of $k((x^{-1}))$ with degree below n to the subset of $k((x^{-1}))$ with degree below $n + t$.
- Similar comments apply to bA, coming from two further types but producing the same type as output, and then of course these are all k-vector spaces, in particular with a subtraction operation on the output type giving $aB - bA$.
- Now apply the divide-by-x^{n-t}-discarding-remainder operation, which can be defined directly or induced through some calculation (and some care since I didn't require $n \geq t$) from the same operation on $k[x]$, to map the subset of $k((x^{-1}))$ with degree below $n + t$ to the subset of $k[x]$ of degree below $2t$.
- And, great, now we have the desired linear map from dimension $2t + 1$ to dimension $2t$, with all of the index and coindex manipulations factored into separate statements about various maps between 10 different series types: $k[x]$, $k[x]$ of degree at most t, $k[x]$ of degree below t, $k((x^{-1}))$, $k((x^{-1}))$ of degree below n, $k((x^{-1}))$ of degree at most n, two product types, $k((x^{-1}))$ of degree below $n + t$, and $k[x]$ of degree below $2t$.
- There are then a few further steps since the $aB - bA$ in the middle of the proof isn't the same as the object that the theorem is talking about, namely $aB - bA$ in $k((x^{-1}))$, but rather the image of that object under maps that preserve degrees. Maybe Lean already has a degree-preserving-map type.

Pieces of this lengthy decomposition are also pieces of other proofs, offering the hope of usefully compressing a sufficiently large pile of proofs. Giving names to useful bundles of hypotheses often helps mathematics move forward. But a proof assistant that provides endless ways to say the same thing—without completely automatic translations—is making lemmas less likely to directly plug into each other, and is slowing the user down.

I'm skeptical of the idea that concise syntax requires this split of semantics. Imagine a front-end converter for HOL Light that, in the context of hypotheses saying that g, h are in a commutative ring R,

- lets the user see and type $g+h$ as an abbreviation for `ring_add R g h`, and
- automatically, whenever $g+h$ appears, inserts a boilerplate proof step saying $g + h \in R$.

This wouldn't be changing what theorems actually say inside the system, and in particular wouldn't be changing how theorems compose; it would be a separate layer improving how some theorems are displayed and typed. Mizar has a "soft typing" system that sounds similar, and Isabelle has "locales".

If it's feasible to build a modular system where the syntax advantages of dependent types are provided as a separate layer that doesn't affect theorem semantics, then the slowdown that I've labeled as "the curse of dependent types" should really be called "the curse of non-modular dependent types". For comparison, what [6] calls the "curse of DTT" sounds to me like a different problem that wouldn't be solved by such modularity—basically, whatever syntax rules you pick have to be accompanied by parsing algorithms, and this can be difficult or impossible to get right if the rules are too complicated.

4.8. User interfaces. Finally, I'll comment on user interfaces, since I noticed one user interface slowing me down considerably more than the other.

I heard that Lean supports multiple cores. I installed it on a 128-core server with 512GB of RAM, and made sure the server wasn't busy with anything else.

The documentation said "there are three editors you can use with Lean", one of those being neovim. Great—I've been using the vi family of editors for a long time, and would have been slower using anything else.

There was a warning about the neovim support not being as well tested as the Visual Studio Code support, so I wasn't terribly surprised when neovim locked up after 5 minutes of typing. Hard freeze: had to kill the editor and recover the file, which is a fast operation, except that restarting the editor takes a noticeable number of seconds for Lean to restart.

For comparison, HOL Light takes minutes to start by default—but there are tools listed in the documentation that checkpoint and restart HOL Light after the startup process. I tried one and it worked fine. I rarely restart HOL Light anyway: it keeps running even if I close the editor, and it's responding instantly when I restart the editor. Once or twice a day I'll start a separate copy of HOL Light to recheck my file of completed proofs (sometimes I reshuffle theorems and accidentally move a theorem above a lemma that it needs), but I'm not waiting for that to finish—I'm typing other proofs.

I ended up using an editing pattern for Lean that triggered crashes less frequently, so this particular issue was only a slight advantage for HOL Light. Another user-interface issue described below made more of a difference.

For Lean, the user interface has a main editing window where I'm typing theorems and proofs; this is saved as a file and ends up going online for people to check. There's a separate goal window where Lean displays the current

hypotheses and desired conclusions. (Screenshots that I've seen from Lean with other editors look like this too.) Some annotations are displayed in the main window.

The way I'm currently using HOL Light looks similar to that. There's a main editor window (vim) where I'm typing theorems and proofs. There's a separate goal window where HOL Light displays the current hypotheses and desired conclusions.

One difference is that there's a third window for chatting with the HOL Light software. HOL Light is continually saying things like what `MESON_TAC` is proving, and how many computations it's using, and, when a theorem is proven, a copy of that theorem—typically I'll be using this as a lemma for the next theorem, so it's good to have it available for reference on the side. Sometimes I'll type different theorem names in that window, and HOL Light replies with those theorems.

This interface is a HOL Light add-on `step_hol_light` from Wiedijk [111], plus two small changes; see also [110] and [78]. One change (now integrated into the latest version of [111]) is that I added "`Unix.close fd0'`" just before the end of `write_goals` in `step_hol_light`, fixing a file-descriptor leak; otherwise `step_hol_light` crashes after 8192 goal-stack updates with a common OS configuration. I encountered this after a few hours of work; evidently I'm seeing roughly one new goal stack per second on average, although this average conceals small-scale variations. The other change is that for me the goal window is a shell window running `tail -f hol_light_goals`, rather than the editor window described in the `step_hol_light` documentation.

The way I was using HOL Light before `step_hol_light` had two windows: the main editor window and the HOL Light chat window. I would type some proof steps in the editor window, and then copy and paste them into the chat window, at which point HOL Light would display the updated goals in that window. I switched to `step_hol_light` partway through working on [15], replacing the copy-and-paste steps with the Ctrl-H command from `step_hol_light`.

For Lean, typing `#check` and a theorem name in the main window keeps that theorem similarly visible—if the theorem statement fits onto one line, which it often doesn't even if I use a wide-window format (main window on top, goal window on bottom). There are various features to pop up a full theorem statement. I sometimes fired up a separate window or two to keep the full statements of multiple-line theorems visible while I was typing a proof.

Now here's the big problem—well, no, not a *big* problem, just a continual small drag, something I encountered much more frequently than startup time. Very often I would type a few proof lines and Lean wouldn't respond for a few seconds. Sometimes it wouldn't respond for 10 seconds. What was it doing? Was a 128-core machine not big enough?

There's no performance feedback from Lean by default, except for the following sharp edge: sometimes Lean says that a slow proof has crashed into a "heartbeat" limit. You don't have to do a binary search on heartbeat limits to quantify the slowness of each theorem—there's a `set_option profiler true` option that tells you how long each theorem is taking—but I didn't find a way

to see separate information about the time consumed by each proof line. So I found myself trying to figure out what was triggering Lean's non-responsiveness:

- It was clear that Lean's search tools were one of the triggers—they have random performance, sometimes very fast but sometimes very slow. But I was mainly using those as a way to get an idea of what lemmas were already available—which wasn't how I was spending most of my time. Lean's non-responsiveness seemed much more pervasive.
- I noticed Lean frequently using many cores: it was spinning up 50 threads, 100 threads, even more. The number of threads seemed related to how quickly I was typing. The obvious guess was that each character was causing Lean to re-analyze what I was typing, and it often wasn't keeping up. Writing a theorem as part of a comment, and uncommenting it when I wanted to see feedback from Lean, reduced the frequency of editor crashes and reduced the CPU load. But this didn't seem to be addressing the non-responsiveness: even with much lower load, Lean would often take a long time to respond.
- Long formulas—I mean the lengths in [15], which are nowhere near the lengths frequently appearing in some areas of mathematics—seemed to be another trigger. I figured that this was from Lean trying to decide what type each expression had, so I started sprinkling manual type declarations everywhere (e.g., `h*q:k[X]` instead of just `h*q`). This seemed to help.
- Long proofs were clearly another trigger of non-responsiveness, perhaps the most important trigger. Splitting long proofs into many lemmas clearly helped. I ended up splitting 20 lemmas out of the longest proof in [15] simply to keep Lean's response time under control. The proof is heavily connected, so these lemmas have long lists of hypotheses.

I had become accustomed to sometimes writing longer proofs in HOL Light to save some copy-and-paste time and some naming time, especially for heavily connected proofs like this. I noticed HOL Light slowing down noticeably when it was faced with long lists of intermediate conclusions. I ended up splitting 4 lemmas out of the longest proof in [15] for HOL Light. But the slowdowns for Lean felt much worse than what I've encountered for HOL Light.

For Lean, the editor window automatically includes a progress bar showing which lines Lean has processed already. This progress bar seemed to have per-theorem granularity rather than per-line granularity. It was easy to guess that, whenever I pressed a key, Lean was re-processing the entire proof from the beginning. This redundant work gave an easy explanation for how slowly Lean was handling each new proof line. I'm told that, yes, this is how Lean naturally handles proofs—switching to another editor wouldn't have helped.

I've heard about ongoing work to address various performance problems in Lean, including the problems I encountered. For example, it seems conceptually straightforward to add a cache recording how proof steps were previously processed, so that re-processing a proof spends time only on the new part. I'm not aware of any structural reason that Lean shouldn't be able to provide HOL Light's level of user-interface responsiveness.

5 Libraries

> *Provers will not be used routinely until there is a solid basis of well-developed mathematical theories from which novel research can grow. But this basis must be provided by experienced mathematicians adopting provers and formalising existing mathematics. Some initial formalisation has been started by a few strongly motivated mathematicians and those computer scientists with an interest in both automated proof and mathematics, but this is insufficient as a basis for novel mathematics. So, we have deadlock.*
>
> —2011 Bundy [31, Section 6]

There's a critical ambiguity in the above quote. How comprehensive does a proof-assistant library have to be to qualify as being "sufficient as a basis for novel mathematics"? If it makes 20% or 50% or 80% of new proofs affordable to formalize today, does one say that, no, this isn't sufficient, because obviously the objective is to reach 100%?

It is easy to see how the above argument that there's "deadlock"—neither side making progress—relies on this ambiguity:

- Interpreting "sufficient as a basis for novel mathematics" as "sufficient as a basis for *some* new mathematics" breaks the second step of the argument, the claim that "some initial formalisation" is "insufficient". One doesn't need to see counterexamples (such as the initial work being sufficient for the new mathematics in [24]) to see the gap in the argument.
- Interpreting "sufficient as a basis for novel mathematics" as "sufficient as a basis for *all* new mathematics" breaks the first step of the argument, the claim that not having such a library prevents provers from being used "routinely".

I don't mean to downplay the importance of proof-assistant libraries. On the contrary, each of my computer-checked-proof projects summarized in Section 3 included writing computer-checked versions of some background theory, and the amount I was writing certainly included whatever background people *hadn't* already explained to the computer, plus redoing things that I didn't find or that I decided not to use for whatever reason. Avigad and Harrison [7, page 75] comment that "a formalizer often finds elementary gaps in the supporting libraries that require extra time and effort to fill"; see also [63, Sections 3 and 5].

As more difficult case studies, this section explains what's going on in my two recent preprints with theorems *without* computer-checked proofs. I mentioned in Section 1 that I would expect a few months of work to be required in each case to cover all the necessary background in a proof assistant.

I don't see this as an indication that there's a deadlock for these background topics. I think these two papers simply happen to be unlucky, where the background they're pulling together includes too many pieces that I haven't found in the libraries. Meanwhile other people are writing papers pulling together

other selections of background, and will sometimes find enough in the libraries that filling in the rest is affordable.

If, for example, a library doesn't have the basic theory of lattices, it's easy to imagine the theory being added *en passant* by someone writing computer-checked proofs for a paper proving something about the latest algorithm to reduce lattice bases. If the library doesn't have Minkowski's lattice-based bound for class groups of number fields, it's easy to imagine the theory being added *en passant* by someone writing computer-checked proofs for a paper using a class-group calculation to characterize solutions to a Diophantine equation. Et cetera.

Or maybe these things are instead done by some student who's taking a number-theory course and decides to post some computer-checked proofs. Mathematics students reportedly enjoy proof-assistant courses such as [34].

Another suggestion from [31, Section 6.4] is to have papers include computer-checked proofs that take previous non-computer-checked theorems as axioms ("authors would be free to encode other mathematicians' theorems as axioms in their formalisation"). I worry that allowing this would exacerbate the problem from Section 4.2, leading to more and more examples of theorems being labeled as computer-checked but being discovered to be incorrect; the same concern was already expressed in [74, page 17]. The suggestion to allow such proofs was explicitly motivated in [31] as an attempt to overcome deadlock, but [31] never justified the claim that there was deadlock in the first place.

5.1. Algorithms. The case study in Section 5.2 below is an example of a paper with theorems referring explicitly to algorithms. There are many more such papers in the literature. Two famous examples from the Annals of Mathematics are 1987 Lenstra "Factoring integers with elliptic curves" [77] and 2004 Agrawal–Kayal–Saxena "PRIMES is in P" [3].

For the first example, one finds "algorithm" mentioned explicitly in, e.g., the statement of [77, Proposition 2.7]. For the second example, seeing that "PRIMES is in P" is a statement about algorithms requires unwrapping some definitions:

- "P" is the set of languages recognizable in polynomial time.
- A "language" means a subset of $\{0,1\}^*$. In particular, "PRIMES" means the set of prime numbers written in binary: 10, 11, 101, etc.
- An algorithm A "recognizes" a language L if $A(s)$ is 1 for all $s \in L$ and 0 for all $s \notin L$. Here $A(s)$ means the output of A given input string s.
- The algorithm A takes "polynomial time" if there's some polynomial p such that, for every ℓ, for every string s of length ℓ, the time taken by A on input s is at most $p(\ell)$.

Actually, [3] stated stronger theorems that specify a particular prime-recognition algorithm and say something more precise about the asymptotic time taken by the algorithm: [3, Theorem 4.1] says "The algorithm above returns PRIME if and only if n is prime", and [3, Theorem 5.1] says "The asymptotic time complexity of the algorithm is $O^{\sim}(\log^{21/2} n)$". The "$O^{\sim}$" notation is like O but allows lower-order logarithmic factors: for example, an algorithm taking time $O(b(\log b)^2)$ to multiply b-bit integers takes time $O^{\sim}(b)$.

What are the underlying definitions of "algorithm" and "time"? 1994 Papadimitriou [90], a textbook on the theory of computational complexity, said that Turing machines "will be our formal model for algorithms in this book", gave definitions of single-tape and multi-tape Turing machines, and defined "time" for a multi-tape Turing machine as the number of steps taken by the machine. In principle it's clear how to give a reasonably short computer-checked proof of "PRIMES is in P", and more specifically of the 21/2 time exponent, starting from these definitions:

- At the bottom level, 2020 Forster–Kunze–Wuttke [45] gave a definition of multi-tape Turing machines in the Coq proof assistant, along with a mechanism to build and analyze Turing machines starting from programs written in a nicer language.
- 1994 Schönhage–Grotefeld–Vetter [96] presented explicit multi-tape Turing machines for fast arithmetic on integers and polynomials. Computer-checked proofs for the subroutines needed for [3] shouldn't be too hard to write.
- At the top level, the original 2002 version of [3] relied on some heavy-duty analytic number theory, but Lenstra pointed out how to avoid this, achieving heavy-duty exponent 15/2 and elementary exponent 21/2; the journal version of [3] presents both approaches. As a separate issue, proving the speed of the algorithm stated in [3] seems to require transcendental number theory (there's a gap at this point in [3]; my paper [10] pinpointed the issue and supplied a proof using Baker's theorem), but it's easy to tweak the algorithm to avoid this.

2021 Chan–Norrish [38] used the HOL4 proof assistant to cover the top level, and claimed that it "establishes formally that the AKS algorithm indeed shows 'PRIMES is in P' "—but the paper never covered the bottom two levels; it didn't formally define P, for example. [38, Section 2.3, "Machine model"] postulated the time taken by various arithmetic operations and list operations; it did not define "algorithm", let alone claim that this definition of "algorithm" allows the same set of polynomial-time computations as Turing machines.

Having a definition of "algorithm" might not seem important. If a specific algorithm A is defined as "Input x; compute $y \leftarrow g(x)$; compute $z \leftarrow f(y)$; output z" then a theorem saying that $A(x) = h(x)$ for all x can be, and in the literature often is, rephrased as a theorem saying $f(g(x)) = h(x)$ for all x. Loops can be rephrased as recursive definitions. Time analyses of any specific algorithm can be rephrased as analyses of the output of a modified algorithm that incorporates a time-tracking step into each step of the original algorithm.

But the literature often has good reasons for stating theorems that explicitly involve the concept of algorithms. It is often informative to consider properties of broad classes of algorithms, such as the set of polynomial-time algorithms. Algorithm designers often design algorithms by applying known transformations of classes of algorithms, and often save time in proofs by appealing to general correctness theorems for the transformations (random example: [98])—but stating and proving the general theorems requires suitable definitions of the relevant set of algorithms. Furthermore, the literature often studies the *limits* of

what broad classes of algorithms can do; this is of interest both for its own sake and as a guide to algorithm designers.

A complication here is that there isn't just one concept of "algorithm" in the literature. For example, multi-tape Turing machines aren't good enough for the theorems in [77] about an algorithm using elliptic curves to try to factor n: [77, Section 2.5] states a "probabilistic algorithm" that starts by drawing elements of $\mathbb{Z}/n$ "at random". Supplementing a Turing machine with the ability to flip coins gives a different machine model, a randomized Turing machine. A randomized Turing machine has an output distribution and a time distribution, which are defined via the infinite probability space $\{0,1\}^\infty$, although for [77] one can get away with finite probability spaces. Anyway, enough computer-checked measure theory is already available to handle proofs about probabilities, and there are computer-checked proofs of the behavior of some *specific* probabilistic algorithms (see, e.g., [67])—which isn't the same as having a definition of "probabilistic algorithm" and theorems about that concept.

5.2. Looseness of FO derandomization. Cryptography is one context where it's important to understand the limits of broad classes of algorithms. The point is that we'd like cryptosystems to protect against all feasible attacks.

I'll focus here on one of the common design tools in cryptography, namely upgrading a simpler security property into a more complicated security property. The usual pattern is that there's a transformation that, given any cryptosystem X achieving security property S, produces a cryptosystem Y achieving security property T, backed by a proof showing how any algorithm to attack Y can be converted into an algorithm to attack X. This simplifies the analysis of the T-security of Y: it suffices to analyze the S-security of X.

Cryptographers have an unfortunate habit of using the phrase "provably secure" to refer to any cryptosystem Y produced by a transformation of this type. Often Y isn't actually secure; see 2019 Koblitz–Menezes [72] for a survey. The most obvious way this can happen, beyond proof errors, is that X isn't secure. Another common failure mode is as follows:

- The proof puts a bound on the gap between the quantitative security levels of X and Y.
- People design X to reach, e.g., 128 bits of security, meaning that every high-probability attack takes time at least 2^{128}, and leap to the conclusion that Y also has 128 bits of security.
- The bound on the gap turns out to be very large, for example not ruling out the possibility of Y having just 64 bits of security. (In cryptographic jargon, the bound is "loose".)
- Y is then shown to be in the "nightmare scenario" where the security loss is about as bad as the proven bound allows, for example having just 64 bits of security even though X still seems to have 128 bits of security. (At this point mathematicians would call the bound reasonably tight, meaning that no big improvements in the bound are possible, but in cryptography the bound is still called "loose".)

Beyond the examples of "provably secure" cryptosystems whose security claims are known to have failed because of "looseness", there are further examples of cryptosystems that have gone through the first three steps of this failure mode: the proofs are known to be "loose", but the designers aren't targeting higher security levels for X to compensate. Typically the designers are worrying about performance, and are hoping that someone will come up with a "tighter" proof, and, more to the point, are hoping that Y will be as secure as X whether or not there's a proof.

One simple example of a very widely used cryptographic transformation is "Fujisaki–Okamoto derandomization". To explain this, I need to start with the general setting of "public-key encryption":

- There's a randomized "key-generation" algorithm that produces a user's "secret key" k and "public key" K.
- There's an "encryption" algorithm E that, given a "plaintext" x and the public key K, produces a "ciphertext" y. This algorithm is allowed to be randomized, so $y = E(x, K, r)$ for some random r.
- There's a "decryption" algorithm D that, given the ciphertext $E(x, K, r)$ and the secret key k, recovers x.

The simplest security goal in this setting is "one-wayness". Let's assume that the plaintext x is chosen uniformly at random from a set of 2^{256} plaintexts. One-wayness means that every feasible algorithm has negligible probability of recovering x given $E(x, K, r)$ and the public key K; the definition is parameterized by the notions of "feasible" and "negligible".

One-wayness doesn't guarantee security if a user is actually sending a 1-bit message by choosing x to be either 0 or 1. It also does nothing to rule out the following type of disaster: if D receives an input that *doesn't* have the form $E(x, K, r)$, then D outputs the secret key k, or something else from which an attacker can easily recover k. So people design tools to upgrade one-wayness to a stronger security property.

At this point I can say what FO derandomization does: it replaces $y = E(x, K, r)$ with $y = E(x, K, H(x))$, where H is a public "hash function" that everyone can compute. This isn't actually designed to upgrade security from one-wayness to something better: it's instead designed to preserve one-wayness while making encryption deterministic, as a preparatory step for transformations that upgrade to better security but that rely on encryption being deterministic. I won't describe the latter transformations; readers who are interested can consult 1999 Fujisaki–Okamoto [46] and other papers cited in [14].

There's a proof saying that FO derandomization preserves one-wayness *except* for the following two issues:

- One issue is that the proof uses what's called the "random-oracle model", where the hash function H is modeled as a uniform random function and success probabilities of attacks are defined as averages over H. Any specific easy-to-compute function that we plug in for H could be breakable without contradicting the proof.

- The other issue is that the proof is loose. Concretely, for an attacker carrying out 2^{100} hash queries, the proof allows the attacker's success chance against $E(x, k, H(x))$ to be 2^{100} times larger than the attacker's success chance against $E(x, k, r)$.

The underlying definitions of algorithms are definitions of probabilistic oracle algorithms: Turing machines are augmented not just with a mechanism to read the next bit from an auxiliary input chosen uniformly at random from $\{0,1\}^\infty$, but also with a mechanism to convert x into $H(x)$.

Typically E is designed with the goal of ensuring that all feasible attacks have success probability considerably below 1. This isn't good enough for the proof to guarantee security of the derandomized version of E. One should instead design E with the goal of ensuring that all feasible attacks have success probability below, say, 2^{-128}.

Starting in 2018, I asked various people what they thought about this gap. Some of them said that coming up with a better proof shouldn't be hard, and started sketching better proofs, but ran into trouble.

In 2021, I realized how to turn the trouble into examples of cryptosystems where FO derandomization appears to lose security—including an example, in the random-oracle model, of a cryptosystem where FO derandomization can be *proven* to lose security. These cryptosystems are in the nightmare scenario: the attacker's success chance is multiplied by almost the number of hash queries, for example 2^{100}.

This doesn't say that previous proposals of cryptosystems using FO derandomization are broken, but it means that the gap in the security analysis of FO derandomization for those cryptosystems isn't going to be closed by a proof at the level of generality of the existing proof of derandomization. Maybe there's a proof using some specific features of those proposals—or maybe there's an attack.

My preprint [14] states theorems about the provable example: in particular, about the performance of a particular algorithm attacking the derandomized system, and about the performance of any algorithm attacking the original system. The theorem statements don't involve algorithm time, but they do involve the number of oracle calls, and they involve success probabilities. The underlying definition of algorithms needs to allow oracles and randomization.

5.3. Non-randomness of *S*-unit lattices. Readers who don't see algorithms as interesting mathematical objects will, I suspect, be happier with the following case study. As a starting point, consider the ratio between

- the number of elements of $\mathbb{Z}^2$ inside a circle of radius r centered at the origin and
- πr^2, the area inside the circle.

Gauss [49, pages 277–278] gave explicit bounds showing that this ratio converges to 1 as $r \to \infty$. More generally, for any determinant-1 lattice $L \subseteq \mathbb{R}^n$ of rank n, the ratio between

- the number of elements of L inside rB and
- the volume of rB

converges to 1 as $r \to \infty$, where $rB = \{x \in \mathbb{R}^n : |x| \leq r\}$.

Choose α so that the volume of αB is 1. Calculate ball volumes to see that $\alpha \in (1 + o(1))\sqrt{n/2\pi e}$ as $n \to \infty$. Define $\lambda_1(L)$ as the minimum length of nonzero elements of L. The following heuristic argument concludes that $\lambda_1(L)$ is between α and $3^{1/n}\alpha$, hence also in $(1 + o(1))\sqrt{n/2\pi e}$:

- Heuristically, there's 1 element of L inside αB, evidently just the origin.
- Heuristically, since $3^{1/n}\alpha B$ has volume 3, there are 3 elements of L inside $3^{1/n}\alpha B$, evidently the origin and $\pm v$ where $\pm v$ are two elements of L of length $\lambda_1(L)$.
- Consequently $\alpha < \lambda_1(L) \leq 3^{1/n}\alpha$.

The heuristic steps ignore the difference between the aforementioned ratio and 1. This obviously can't be *exactly* right (consider a ball of volume 3 when L has four vectors of length $\lambda_1(L)$, or a ball of volume 2, or a ball of volume 1.7), so let's call the conclusion a "heuristic approximation".

One can, with more work, prove something along these lines. 1945 Siegel [**99**] gave an explicit definition of a probability measure on the set of determinant-1 rank-n lattices $L \subseteq \mathbb{R}^n$ that's invariant under the action of $\mathrm{SL}_n(\mathbb{R})$, and 1956 Rogers [**95**] analyzed the distribution of $\lambda_1(L)$ for L chosen at random from this measure. See 2019 Strömbergsson–Södergren [**103**, Remark 1.8] for more precise results. A random lattice L has $\lambda_1(L) \in (1 + O((\log n)/n))\sqrt{n/2\pi e}$ with probability $1 - o(1)$ as $n \to \infty$, at least for some interpretations of the quantifiers.

1998 Hoffstein–Pipher–Silverman [**65**, page 273] used the name "gaussian heuristic" for the statement that $\sqrt{n/2\pi e} \leq \lambda_1(L) \leq \sqrt{n/\pi e}$. That paper introduced an influential line of lattice-based cryptosystems, and used heuristics about vector lengths to analyze the scalability of some attack algorithms. Many followup papers adopted the "Gaussian heuristic" name for a cloud of related heuristics: for example, the statement that $\lambda_1(L) = \sqrt{n/2\pi e}$, or the statement that $\lambda_1(L) = \alpha$, or the statement that there are $1.3^{n+o(n)}$ elements of $L \cap 1.3\lambda_1(L)B$, or the statement that the latter elements are "uniformly distributed over the ball", in the words of [**42**, Heuristic 2, "consequence of GH"].

These heuristics have been used to analyze the scalability of a variety of attack algorithms, for example producing the complicated formulas discussed in Section 3.2. As a simpler example, consider the following greedy algorithm to find an element $v \in L$ given $v + \epsilon$: try subtracting u from the input for each u in a database of short elements of L, and repeat as long as this keeps reducing the input. The heuristics imply that, to have a noticeable chance of reducing the input down to ϵ, this algorithm requires a database size exponential in n.

I doubt that Gauss would have approved of having his name used for these heuristics. Consider, for example, the important lattice $L = \mathbb{Z}^n$:

- The minimum nonzero length $\lambda_1(L)$ is 1, so $\sqrt{n/2\pi e}$ is a worse and worse approximation to $\lambda_1(L)$ as n grows.

- The only elements of $L \cap 1.3\lambda_1(L)B$ are the standard unit vectors, their negatives, and 0. Consequently, $\#(L \cap 1.3\lambda_1(L)B)$ isn't exponential in n: it's $2n + 1$.
- Greedily subtracting off those vectors from a given $v + \epsilon$ reliably finds ϵ. There's no need for an exponential-size database.

This particular counterexample to the heuristics might not seem relevant to attack papers talking about "random" lattices—but one has to check whether the lattices showing up in the attacks really are "random"; otherwise the attacks might work much better than the heuristics predict. Let's look at how some special lattices arise in one line of attacks.

2009 Gentry [51] introduced a "fully homomorphic" cryptosystem in which the secret key is an element g of (for example) the ring $R = \mathbb{Z}[x]/(x^n + 1)$ where n is a power of 2. The public key includes the ideal gR. One can view gR as a lattice, and view the secret generator g as a short element of that lattice, but finding short nonzero elements of lattices is supposed to be hard when n is big.

I wrote a blog post in 2014 [11] about "an ideal-lattice attack strategy that, unlike traditional lattice attacks, exploits the multiplicative structure of ideals":

- There were already well-known algorithms that find *some* generator of gR in, conjecturally, subexponential time. The only issue for the attacker is that this generator will have extremely large coefficients: it will be gu for some extremely large unit $u \in \mathbb{Z}[x]/(x^n + 1)$.
- Dirichlet's log map converts the set of units into a lattice, and converts the problem of finding u close to gu into the problem of finding a lattice vector close to the log of gu. This lattice has rank only $n/2 - 1$. Rank isn't the only parameter influencing the performance of lattice algorithms, but it's an important parameter.

I described this method of finding generators as "reasonably well known among computational algebraic number theorists", and explained how to use subrings of $\mathbb{Z}[x]/(x^n + 1)$ to reduce the dimension further.

2014 Campbell–Groves–Shepherd [36] then observed that one can quickly solve this particular lattice problem by subtracting off logs of particular units, namely textbook generators of the group of "cyclotomic units". There's an algebraic obstruction here—cyclotomic units aren't necessarily all units of $\mathbb{Z}[x]/(x^n+1)$—but the index (number theorists call this "h_n^+") is typically small, and in particular is reasonably conjectured to be 1 whenever n is a power of 2.

Meanwhile 2010 Lyubashevsky–Peikert–Regev [80] had shown that breaking some other cryptosystems implied being able to find reasonably short nonzero elements of arbitrary ideals, and had described this as a "very strong hardness guarantee" for those cryptosystems. The attack against Gentry's cryptosystem was finding short elements of *some* ideals; could it be extended to handle arbitrary ideals?

In 2016 [12], I suggested "solving a close-vector problem in the S-unit lattice, to recover a short g from any S-unit multiple ug" as a generalization of unit attacks. As background, "S-units" are a standard generalization of units in

number theory: there's a set S of primes, and anything whose factorization is supported on those primes is called an S-unit (or "S-smooth"). For example, $10/3$ is a $\{2,3,5\}$-unit. The lattice of logs of units extends straightforwardly to a lattice of logs of S-units. The S-unit lattice is also the main object used in the conjecturally-subexponential-time algorithms to find generators of principal ideals.

2019 Pellet-Mary–Hanrot–Stehlé [92] analyzed this approach, specifically using the greedy algorithm to solve the close-vector problem, and concluded that this was better than previous attacks for some sizes of ϵ but took exponential time for the sizes of ϵ used in [80]. Internally, this analysis applied the aforementioned heuristics to S-unit lattices.

Lange and I have a preprint [19] saying that these heuristics "underestimate the power of S-unit attacks: S-unit lattices, like $\mathbb{Z}^d$, have much shorter vectors and reduce much more effectively" than the heuristics predict. In particular, for each n, an S-unit attack with database size $\#S^{1+o(1)}$ has success probability converging to 1 as S increases; the heuristic analysis of [92] instead says that the probability converges exponentially to 0.

This still leaves open the possibility that the heuristics are reasonably accurate for smaller S—so [19] also collects evidence against the heuristics in further cases: first, all sizes of S for the case $n = 1$; second, for each n, the smallest S-unit lattice, namely the unit lattice.

Not everything in [19] is stated as a theorem: for example, the random-lattice evidence for the heuristics is merely reviewed, not reproven. But there are various theorems in [19], and the proofs rely on a range of algebraic and analytic background facts, including various theorems of Kummer about cyclotomic units, Dirichlet's unit theorem, the Brauer–Siegel theorem (in the form of [108, page 44, bottom paragraph]) about the asymptotics of products of class numbers and regulators, Robbins's version [94] of Stirling's formula, etc.

5.4. Interoperability among proof assistants. Research mathematics is a cooperative, worldwide endeavor that transcends superficial language differences. You can use a theorem that was published in French or German or Russian—and, on the flip side, you don't get any research credit for being the first person to write the same thing in English, even if the exposition is a valuable service for monolingual students.

Now imagine an alternate universe where each language has its own walled-off mathematics literature. If you want to do mathematics, first you have to listen to people claiming that you can't write serious analysis in French and people claiming that you can't write serious algebra in English, and then you have to pick a language for your first paper. You can't use lemmas from any of the other languages in your own theorems until they've been written in *this* language.

That's the current universe of proof assistants. When people using proof assistant P want theorems that are already in proof assistant Q, they generally end up translating proofs by hand, with only partial automation. Even something that sounds as minor as moving proofs from version 3 of Lean to version 4 of Lean was reportedly a large task overall.

Say a paper relies on background A, B, C, D, E, F, and proof assistant P provides background A, B, C, and proof assistant Q provides background A, D, E. Writing computer-checked proofs then means adding background D, E, F if one picks proof assistant P, or adding background B, C, F if one picks proof assistant Q, rather than just adding background F. The redundant work doesn't stop progress, but it slows it down.

I don't expect this aspect of the proof-assistant situation to persist. For comparison, programming languages already tend to be interoperable. The most powerful feature of each programming language is its ability to use a mountain of software written in other programming languages.

There's already progress in automating proof translations: see, e.g., [**73**]. Surely someone will manage to write a proof assistant that makes it easy to use lemmas from other proof assistants. Users will love this. Other proof assistants will follow. Proof assistants will continue to compete for ease of use, but I hope and expect that they won't continue competing for time spent building separate libraries. Instead they'll be cooperating as tools for expanding a unified, shared, computer-comprehensible version of the mathematics literature.

References

[1] Scott Aaronson, *Death of proof greatly exaggerated* (2019). URL: https://scottaaronson.blog/?p=4133. Citations in this document: §1, §1.

[2] Oskar Abrahamsson, Magnus O. Myreen, Ramana Kumar, Thomas Sewell, *Candle: A verified implementation of HOL Light*, in ITP 2022 [**5**] (2022), 3:1–3:17. URL: https://cakeml.org/itp22-candle.pdf. Citations in this document: §1.1.

[3] Manindra Agrawal, Neeraj Kayal, Nitin Saxena, *PRIMES is in P*, Annals of Mathematics. Second Series **160** (2004), 781–793. URL: https://annals.math.princeton.edu/wp-content/uploads/annals-v160-n2-p12.pdf. Citations in this document: §5.1, §5.1, §5.1, §5.1, §5.1, §5.1, §5.1, §5.1, §5.1.

[4] Erdem Alkim, Léo Ducas, Thomas Pöppelmann, Peter Schwabe, *Post-quantum key exchange—a new hope*, in USENIX Security 2016 [**66**] (2016), 327–343. URL: https://eprint.iacr.org/2015/1092. Citations in this document: §3.2.

[5] June Andronick, Leonardo de Moura (editors), *13th international conference on interactive theorem proving, ITP 2022, August 7–10, 2022, Haifa, Israel*, 237, Schloss Dagstuhl—Leibniz-Zentrum für Informatik, 2022. ISBN 978-3-95977-252-5. See [2].

[6] Jeremy Avigad, *Automated reasoning for the working mathematician* (2019). URL: https://www.andrew.cmu.edu/user/avigad/Talks/london.pdf. Citations in this document: §2.3, §4.7.

[7] Jeremy Avigad, John Harrison, *Formally verified mathematics*, Communications of the ACM **57** (2014), 66–75. URL: https://www.cl.cam.ac.uk/~jrh13/papers/cacm.html. Citations in this document: §1, §1, §4.1, §5.

[8] Grzegorz Bancerek, Czeslaw Bylinski, Adam Grabowski, Artur Kornilowicz, Roman Matuszewski, Adam Naumowicz, Karol Pak, *The role of the Mizar mathematical library for interactive proof development in Mizar*,

Journal of Automated Reasoning **61** (2018), 9–32. URL: https://mizar.uwb.edu.pl/people/romat/doi-10.1007-s10817-017-9440-6.pdf. Citations in this document: §1.

[9] Grzegorz Bancerek, Czeslaw Bylinski, Adam Grabowski, Artur Kornilowicz, Roman Matuszewski, Adam Naumowicz, Karol Pak, Josef Urban, *Mizar: state-of-the-art and beyond*, in CICM 2015 [**69**] (2015), 261–279. Citations in this document: §1.

[10] Daniel J. Bernstein, *Computing logarithm floors in essentially linear time* (2004). URL: https://cr.yp.to/papers.html#logfloor. Citations in this document: §5.1.

[11] Daniel J. Bernstein, *A subfield-logarithm attack against ideal lattices* (2014). URL: https://blog.cr.yp.to/20140213-ideal.html. Citations in this document: §5.3.

[12] Daniel J. Bernstein, *S-unit attacks* (2016). URL: https://groups.google.com/g/cryptanalytic-algorithms/c/mCMdsFemzQk/m/3cewE8Q5BwAJ. Citations in this document: §5.3.

[13] Daniel J. Bernstein, *Verified fast formulas for control bits for permutation networks* (2020). URL: https://cr.yp.to/papers.html#controlbits. Citations in this document: §1, §2, §3.1, §4.2.

[14] Daniel J. Bernstein, *On the looseness of FO derandomization* (2021). URL: https://cr.yp.to/papers.html#footloose. Citations in this document: §5.2, §5.2.

[15] Daniel J. Bernstein, *Understanding binary-Goppa decoding* (2023). URL: https://cr.yp.to/papers.html#goppadecoding. Citations in this document: §3.4, §3.4, §3.4, §3.4, §3.4, §4, §4, §4, §4.2, §4.2, §4.2, §4.2, §4.3, §4.3, §4.3, §4.3, §4.3, §4.3, §4.3, §4.3, §4.3, §4.5, §4.5, §4.5, §4.5, §4.5, §4.6, §4.7, §4.7, §4.7, §4.7, §4.7, §4.8, §4.8, §4.8, §4.8.

[16] Daniel J. Bernstein, *Hull Light* (2023). URL: https://cr.yp.to/2023/hull-light-20230416.sage. Citations in this document: §3.3, §3.3, §3.3, §3.3.

[17] Daniel J. Bernstein, *Multi-ciphertext security degradation for lattices* (2023). URL: https://cr.yp.to/papers.html#lprrr. Citations in this document: §1, §3.2, §3.2, §3.2, §3.4, §4.2, §4.5.

[18] Daniel J. Bernstein, Tung Chou, *CryptAttackTester: formalizing attack analyses* (2023). URL: https://cr.yp.to/papers.html#cat. Citations in this document: §3.2.

[19] Daniel J. Bernstein, Tanja Lange, *Non-randomness of S-unit lattices* (2021). URL: https://cr.yp.to/papers.html#spherical. Citations in this document: §5.3, §5.3, §5.3, §5.3.

[20] Daniel J. Bernstein, Bo-Yin Yang, *Fast constant-time gcd computation and modular inversion*, IACR Transactions on Cryptographic Hardware and Embedded Systems **2019.3** (2019), 340–398. URL: https://gcd.cr.yp.to/papers.html. Citations in this document: §3.3, §3.3, §3.3.

[21] Yves Bertot, Pierre Castéran, *Interactive theorem proving and program development: Coq'art: the calculus of inductive constructions*, Springer, 2013. ISBN 978-3-642-05880-6. Citations in this document: §1.

[22] Jasmin Blanchette, Catalin Hritcu (editors), *Proceedings of the 9th ACM SIGPLAN international conference on certified programs and proofs, CPP 2020, New Orleans, LA, USA, January 20–21, 2020*, ACM, 2020. ISBN 978-1-4503-7097-4. See [45], [82].

[23] Sandrine Blazy, Christine Paulin-Mohring, David Pichardie (editors), *Interactive theorem proving—4th international conference, ITP 2013, Rennes, France, July 22–26, 2013, proceedings*, 7998, Springer, 2013. ISBN 978-3-642-39633-5. See [54].

[24] Thomas F. Bloom, *On a density conjecture about unit fractions* (2021). URL: https://arxiv.org/abs/2112.03726. Citations in this document: §1.1, §1.1, §5.

[25] Anthony Bordg, Lawrence C. Paulson, Wenda Li, *Simple type theory is not too simple: Grothendieck's schemes without dependent types*, Experimental Mathematics **31** (2022), 364–382. URL: https://arxiv.org/pdf/2104.09366.pdf. Citations in this document: §4.

[26] Martin Brain, Carlos Cid, Rachel Player, Wrenna Robson, *Verifying Classic McEliece: examining the role of formal methods in post-quantum cryptography standardisation*, in CBCrypto 2022 [**41**] (2022), 21–36. URL: https://eprint.iacr.org/2023/010. Citations in this document: §4.3.

[27] Nicolaas Govert de Bruijn, *A survey of the project AUTOMATH*, in [**97**] (1980), 579–606. URL: https://research.tue.nl/en/publications/a-survey-of-the-project-automath-2. Citations in this document: §2.3.

[28] Nicolaas Govert de Bruijn, *Checking mathematics with computer assistance*, Notices of the American Mathematical Society **38** (1991), 8–15. URL: https://research.tue.nl/en/publications/checking-mathematics-with-computer-assistance. Citations in this document: §1.

[29] Ulrik Buchholtz, Floris van Doorn, Egbert Rijke, *Higher groups in homotopy type theory*, in LICS 2018 [**39**] (2018), 205–214. URL: https://florisvandoorn.com/papers/higher-groups.pdf. Citations in this document: §1.1.

[30] Joe Buhler (editor), *Algorithmic number theory, third international symposium, ANTS-III, Portland, Oregon, USA, June 21–25, 1998, proceedings*, 1423, Springer, 1998. ISBN 3-540-64657-4. See [65].

[31] Alan Bundy, *Automated theorem provers: a practical tool for the working mathematician?*, Annals of Mathematics and Artificial Intelligence **61** (2011), 3–14. URL: https://www.pure.ed.ac.uk/ws/files/416060/Automated_theorem_provers_a_practical_tool_for_the_working_mathematician.pdf. Citations in this document: §1, §5, §5, §5, §5.

[32] Kevin Buzzard, *Division by zero in type theory: a FAQ* (2020). URL: https://xenaproject.wordpress.com/2020/07/05/division-by-zero-in-type-theory-a-faq/. Citations in this document: §4.2.

[33] Kevin Buzzard, *Where is the fashionable mathematics?* (2020). URL: https://xenaproject.wordpress.com/2020/02/09/where-is-the-fashionable-mathematics/. Citations in this document: §4.

[34] Kevin Buzzard, *Formalising mathematics* (2023). URL: https://www.ma.imperial.ac.uk/~buzzard/xena/formalising-mathematics-2023/. Citations in this document: §5.

[35] Kevin Buzzard, *Lean 2022 round-up* (2023). URL: https://xenaproject.wordpress.com/2023/01/08/lean-2022-round-up/. Citations in this document: §1.1.

[36] Peter Campbell, Michael Groves, Dan Shepherd, *Soliloquy: a cautionary tale* (2014). URL: https://docbox.etsi.org/Workshop/2014/201410_CRYPTO/S07_Systems_and_Attacks/S07_Groves_Annex.pdf. Citations in this document: §5.3.

[37] Mario Carneiro, *Reimplementing Mizar in Rust*, in ITP 2023 [**87**] (2023), 10:1–10:18. URL: https://arxiv.org/abs/2304.08391. Citations in this document: §1.

[38] Hing-Lun Chan, Michael Norrish, *Mechanisation of the AKS algorithm*, Journal of Automated Reasoning **65** (2021), 205–256. URL: https://link.springer.com/content/pdf/10.1007/s10817-020-09563-y.pdf. Citations in this document: §5.1, §5.1.

[39] Anuj Dawar, Erich Grädel (editors), *Proceedings of the 33rd annual ACM/IEEE symposium on logic in computer science, LICS 2018, Oxford, UK, July 09–12, 2018*, ACM, 2018. See [29].

[40] Richard A. DeMillo, Richard J. Lipton, Alan J. Perlis, *Social processes and proofs of theorems and programs*, Communications of the ACM **22** (1979), 271–280. URL: https://dl.acm.org/doi/pdf/10.1145/359104.359106. Citations in this document: §1.

[41] Jean-Christophe Deneuville (editor), *Code-based cryptography—10th international workshop, CBCrypto 2022, Trondheim, Norway, May 29–30, 2022, revised selected papers*, 13839, Springer, 2023. ISBN 978-3-031-29688-8. See [26].

[42] Léo Ducas, Thijs Laarhoven, Wessel P. J. van Woerden, *The randomized slicer for CVPP: sharper, faster, smaller, batchier*, in PKC 2020 [**70**] (2020), 3–36. URL: https://eprint.iacr.org/2020/120. Citations in this document: §5.3.

[43] Julien Duman, Kathrin Hövelmanns, Eike Kiltz, Vadim Lyubashevsky, Gregor Seiler, *Faster lattice-based KEMs via a generic Fujisaki-Okamoto transform using prefix hashing*, in CCS 2021 [**71**] (2021), 2722–2737. URL: https://eprint.iacr.org/2021/1351. Citations in this document: §3.2.

[44] Walter Feit, John G. Thompson, *Solvability of groups of odd order*, Pacific Journal of Mathematics **13** (1963), 775–1029. Citations in this document: §1.

[45] Yannick Forster, Fabian Kunze, Maxi Wuttke, *Verified programming of Turing machines in Coq*, in CPP 2020 [**22**] (2020), 114–128. URL: https://www.ps.uni-saarland.de/Publications/documents/ForsterEtAl_2019_VerifiedTMs.pdf. Citations in this document: §5.1.

[46] Eiichiro Fujisaki, Tatsuaki Okamoto, *Secure integration of asymmetric and symmetric encryption schemes*, in Crypto 1999 [**112**] (1999), 537–554. URL: https://link.springer.com/content/pdf/10.1007/3-540-48405-1_34.pdf. Citations in this document: §5.2.

[47] Ulrich Furbach, Natarajan Shankar (editors), *Proceedings of the third International Joint Conference, IJCAR 2006*, Lecture Notes in Computer Science, 4130, Springer, 2006. See [61].

[48] M. Ganesalingam, W. Timothy Gowers, *A fully automatic theorem prover with human-style output*, Journal of Automated Reasoning **58** (2017), 253–291. URL: https://arxiv.org/abs/1309.4501. Citations in this document: §1, §4.2.

[49] Carl Friedrich Gauss, *De nexu inter multitudinem classium, in quas formae binariae secundi gradus distribuuntur, earumque determinantem*, in [**50**] (1876), 269–291. Paper says "Commentatio prior societati regiae exhibita 1834". URL: https://ia902808.us.archive.org/21/items/117771763_002/117771763_002.pdf. Citations in this document: §5.3.

[50] Carl Friedrich Gauss, *Werke, Band II* (1876). URL: https://ia902808.us.archive.org/21/items/117771763_002/117771763_002.pdf. See [49].

[51] Craig Gentry, *Fully homomorphic encryption using ideal lattices*, in [**83**] (2009), 169–178. URL: https://dl.acm.org/doi/abs/10.1145/1536414.1536440. Citations in this document: §5.3.

[52] Henri Gilbert (editor), *Advances in Cryptology—EUROCRYPT 2010, 29th annual international conference on the theory and applications of cryptographic*

techniques, Monaco / French Riviera, May 30–June 3, 2010, proceedings, 6110, Springer, 2010. ISBN 978-3-642-13189-9. See [80].

[53] Georges Gonthier, *Formal proof—the four color theorem*, Notices of the American Mathematical Society **55** (2008), 1382–1393. URL: https://www.ams.org/notices/200811/tx081101382p.pdf. Citations in this document: §3.3.

[54] Georges Gonthier, Andrea Asperti, Jeremy Avigad, Yves Bertot, Cyril Cohen, François Garillot, Stéphane Le Roux, Assia Mahboubi, Russell O'Connor, Sidi Ould Biha, Ioana Pasca, Laurence Rideau, Alexey Solovyev, Enrico Tassi, Laurent Théry, *A machine-checked proof of the odd order theorem*, in ITP 2013 [**23**] (2013), 163–179. URL: https://inria.hal.science/hal-00816699. Citations in this document: §1, §1.

[55] Sébastien Gouëzel, Anders Karlsson, *Subadditive and multiplicative ergodic theorems*, Journal of the European Mathematical Society **22** (2020), 1893–1915. URL: https://perso.univ-rennes1.fr/sebastien.gouezel/articles/SubMult.pdf. Citations in this document: §1.1.

[56] Sébastien Gouëzel, Vladimir Shchur, *A corrected quantitative version of the Morse lemma*, Journal of Functional Analysis **277** (2019), 1258–1268. URL: https://perso.univ-rennes1.fr/sebastien.gouezel/articles/morse_lemma.pdf. Citations in this document: §1.1, §1.1.

[57] Thomas Hales, *Formal proof*, Notices of the American Mathematical Society **55** (2008), 1370–1380. URL: https://community.ams.org/journals/notices/200811/tx081101370p.pdf. Citations in this document: §1, §4, §4.1.

[58] Thomas Hales, *An argument for controlled natural languages in mathematics* (2019). URL: https://jiggerwit.files.wordpress.com/2019/06/header.pdf. Citations in this document: §4.2.

[59] Thomas Hales, Mark Adams, Gertrud Bauer, Tat Dat Dang, John Harrison, Hoang Le Truong, Cezary Kaliszyk, Victor Magron, Sean McLaughlin, Tat Thang Nguyen, Quang Truong Nguyen, Tobias Nipkow, Steven Obua, Joseph Pleso, Jason Rute, Alexey Solovyev, Thi Hoai An Ta, Nam Trung Tran, Thi Diep Trieu, Josef Urban, Ky Vu, Roland Zumkeller, *A formal proof of the Kepler conjecture*, Forum of Mathematics, Pi **5** (2017), 29. URL: https://arxiv.org/abs/1501.02155. Citations in this document: §3.3.

[60] John Harrison, *HOL Light: A tutorial introduction*, in FMCAD 1996 [**101**] (1996), 265–269. URL: https://www.cl.cam.ac.uk/~jrh13/papers/demo.html. Citations in this document: §1.

[61] John Harrison, *Towards self-verification of HOL Light*, in IJCAR 2006 [**47**] (2006), 177–191. URL: https://www.cl.cam.ac.uk/~jrh13/papers/holhol.html. Citations in this document: §1.1.

[62] John Harrison, *A formalized proof of Dirichlet's theorem on primes in arithmetic progression*, Journal of Formalized Reasoning **2** (2009), 63–83. URL: https://www.cl.cam.ac.uk/~jrh13/papers/dirichlet.html. Citations in this document: §1.

[63] John Harrison, *Formalizing an analytic proof of the Prime Number Theorem (dedicated to Mike Gordon on the occasion of his 60th birthday)*, Journal of Automated Reasoning **43** (2009), 243–261. URL: https://www.cl.cam.ac.uk/~jrh13/papers/mikefest.html. Citations in this document: §4.1, §4.5, §5.

[64] John Harrison, *The x25519 function for curve25519* (2023). URL: https://github.com/awslabs/s2n-bignum/blob/main/x86/proofs/curve25519_x25519.ml. Citations in this document: §1.1.

[65] Jeffrey Hoffstein, Jill Pipher, Joseph H. Silverman, *NTRU: A ring-based public key cryptosystem*, in ANTS 1998 [**30**] (1998), 267–288. URL: https://ntru.org/f/hps98.pdf. Citations in this document: §5.3.

[66] Thorsten Holz, Stefan Savage (editors), *25th USENIX security symposium, USENIX Security 16, Austin, TX, USA, August 10–12, 2016*, USENIX Association, 2016. URL: https://www.usenix.org/conference/usenixsecurity16. See [4].

[67] Joe Hurd, *Verification of the Miller-Rabin probabilistic primality test*, Journal of Logic and Algebraic Programming **56** (2003), 3–21. URL: https://joe.leslie-hurd.com/papers/miller.pdf. Citations in this document: §1, §5.1.

[68] Yuval Ishai, Vincent Rijmen (editors), *Advances in Cryptology—EUROCRYPT 2019—38th annual international conference on the theory and applications of cryptographic techniques, Darmstadt, Germany, may 19–23, 2019, proceedings, part II*, 11477, Springer, 2019. ISBN 978-3-030-17655-6. See [92].

[69] Manfred Kerber, Jacques Carette, Cezary Kaliszyk, Florian Rabe, Volker Sorge (editors), *Intelligent computer mathematics—international conference, CICM 2015, Washington, DC, USA, July 13–17, 2015, proceedings*, 9150, Springer, 2015. ISBN 978-3-319-20614-1. See [9].

[70] Aggelos Kiayias, Markulf Kohlweiss, Petros Wallden, Vassilis Zikas (editors), *Public-Key Cryptography—PKC 2020—23rd IACR international conference on practice and theory of public-key cryptography, Edinburgh, UK, May 4–7, 2020, proceedings, part II*, 12111, Springer, 2020. ISBN 978-3-030-45387-9. See [42].

[71] Yongdae Kim, Jong Kim, Giovanni Vigna, Elaine Shi (editors), *CCS '21: 2021 ACM SIGSAC conference on computer and communications security, virtual event, Republic of Korea, November 15–19, 2021*, ACM, 2021. ISBN 978-1-4503-8454-4. See [43].

[72] Neal Koblitz, Alfred Menezes, *Critical perspectives on provable security: fifteen years of "another look" papers*, Advances in Mathematics of Communications **13** (2019), 517–558. URL: https://eprint.iacr.org/2019/1336. Citations in this document: §5.2.

[73] Michael Kohlhase, Florian Rabe, *Experiences from exporting major proof assistant libraries*, Journal of Automated Reasoning **65** (2021), 1265–1298. URL: https://link.springer.com/article/10.1007/s10817-021-09604-0. Citations in this document: §5.4.

[74] Angeliki Koutsoukou-Argyraki, *Formalising mathematics – in praxis; a mathematician's first experiences with Isabelle/HOL and the why and how of getting started*, Jahresbericht der Deutschen Mathematiker-Vereinigung (DMV) **123** (2021), 3–26. URL: https://api.repository.cam.ac.uk/server/api/core/bitstreams/85e930f3-81e2-4902-9196-59f88f2da94b/content. Citations in this document: §5.

[75] Leopold Kronecker, *Zur Theorie der Elimination einer Variablen aus zwei algebraischen Gleichungen*, Monatsberichte der Königlich Preussischen Akademie der Wissenschaften zu Berlin (1881). URL: https://archive.org/details/werkehrsgaufvera02kronuoft/page/114/mode/2up. Citations in this document: §4.7.

[76] Joseph-Louis Lagrange, *Sur l'usage des fractions continues dans le calcul intégral*, Nouveaux Mémoires de l'Académie royale des Sciences et Belles-Lettres de Berlin (1776). URL: https://gallica.bnf.fr/ark:/12148/bpt6k229223s/f303. Citations in this document: §4.7.

[77] Hendrik W. Lenstra, Jr., *Factoring integers with elliptic curves*, Annals of Mathematics **126** (1987), 649–673. ISSN 0003-486X. MR 89g:11125. URL: https://www.math.leidenuniv.nl/~hwl/PUBLICATIONS/1987c/art.pdf. Citations in this document: §5.1, §5.1, §5.1, §5.1, §5.1.

[78] Joe Leslie-Hurd, *Emacs macros for HOL Light proof* (2015). URL: https://github.com/gilith/hol-light-emacs/blob/master/README.md. Citations in this document: §4.8.

[79] Gavriela Freund Lev, Nicholas Pippenger, Leslie G. Valiant, *A fast parallel algorithm for routing in permutation networks*, IEEE Transactions on Computers **C-30** (1981), 93–100. Citations in this document: §3.1.

[80] Vadim Lyubashevsky, Chris Peikert, Oded Regev, *On ideal lattices and learning with errors over rings*, in Eurocrypt 2010 [**52**] (2010), 1–23. URL: https://eprint.iacr.org/2012/230. Citations in this document: §5.3, §5.3.

[81] Donald MacKenzie, *Mechanizing proof: computing, risk, and trust*, MIT Press, 2004. Citations in this document: §4.1.

[82] The mathlib community, *The Lean mathematical library*, in CPP 2020 [**22**] (2020), 367–381. URL: https://arxiv.org/abs/1910.09336. Citations in this document: §1.

[83] Michael Mitzenmacher (editor), *Proceedings of the 41st annual ACM symposium on theory of computing, STOC 2009, Bethesda, MD, USA, May 31–June 2, 2009*, ACM, 2009. ISBN 978-1-60558-506-2. See [51].

[84] Otmane Aït Mohamed, César A. Muñoz, Sofiène Tahar (editors), *Theorem proving in higher order logics, 21st international conference, TPHOLs 2008, Montreal, Canada, August 18–21, 2008, proceedings*, 5170, Springer, 2008. ISBN 978-3-540-71065-3. See [100].

[85] Leonardo de Moura, Sebastian Ullrich, *The Lean 4 theorem prover and programming language (system description)*, in CADE 28 [**93**] (2021), 625–635. URL: https://leanprover.github.io/papers/lean4.pdf. Citations in this document: §1.

[86] David Nassimi, Sartaj Sahni, *Parallel algorithms to set up the Benes permutation network*, IEEE Transactions on Computers **C-31** (1982), 148–154. Citations in this document: §3.1.

[87] Adam Naumowicz, René Thiemann (editors), *14th international conference on interactive theorem proving, ITP 2023, July 31 to August 4, 2023, Białystok, Poland*, 268, Schloss Dagstuhl—Leibniz-Zentrum für Informatik, 2023. ISBN 978-3-95977-284-6. See [37].

[88] Tobias Nipkow, Lawrence C. Paulson, Markus Wenzel, *Isabelle/HOL: a proof assistant for higher-order logic*, Lecture Notes in Computer Science, 2283, Springer, 2002. ISBN 978-3-540-43376-7. Citations in this document: §1.

[89] Russell O'Connor, Andrew Poelstra, *A formal proof of safegcd bounds* (2021). URL: https://medium.com/blockstream/a-formal-proof-of-safegcd-bounds-695e1735a348. Citations in this document: §3.3.

[90] Christos M. Papadimitriou, *Computational complexity*, Addison-Wesley, 1994. ISBN 0201530821. MR 95f:68082. Citations in this document: §5.1.

[91] Lawrence C. Paulson, *The de Bruijn criterion vs the LCF architecture* (2022). URL: https://lawrencecpaulson.github.io/2022/01/05/LCF.html. Citations in this document: §4.1.

[92] Alice Pellet-Mary, Guillaume Hanrot, Damien Stehlé, *Approx-SVP in ideal lattices with pre-processing*, in Eurocrypt 2019 [**68**] (2019), 685–716. URL: https://eprint.iacr.org/2019/215. Citations in this document: §5.3, §5.3.

[93] André Platzer, Geoff Sutcliffe (editors), *Automated deduction—CADE 28—28th international conference on automated deduction, virtual event, July 12–15, 2021, proceedings*, 12699, Springer, 2021. ISBN 978-3-030-79875-8. See [85].

[94] Herbert Robbins, *A remark on Stirling's formula*, The American Mathematical Monthly **62** (1955), 26–29. Citations in this document: §5.3.

[95] C. A. Rogers, *The number of lattice points in a set*, Proceedings of the London Mathematical Society. Third Series **6** (1956), 305–320. Citations in this document: §5.3.

[96] Arnold Schönhage, Andreas F. W. Grotefeld, Ekkehart Vetter, *Fast algorithms: a multitape Turing machine implementation*, Bibliographisches Institut, 1994. ISBN 3-411-16891-9. MR 96c:68043. Citations in this document: §5.1.

[97] Jonathan P. Seldin, J. Roger Hindley (editors), *To H.B. Curry: Essays on combinatory logic, lambda calculus and formalism*, Academic Press, 1980. ISBN 0-12-349050-2. See [27].

[98] Victor Shoup, *Fast construction of irreducible polynomials over finite fields*, Journal of Symbolic Computation **17** (1994), 371–391. URL: https://www.shoup.net/papers/fastirred.pdf. Citations in this document: §5.1.

[99] Carl Ludwig Siegel, *A mean value theorem in geometry of numbers*, Annals of Mathematics. Second Series **46** (1945), 340–347. Citations in this document: §5.3.

[100] Konrad Slind, Michael Norrish, *A brief overview of HOL4*, in TPHOLs 2008 [84] (2008), 28–32. URL: https://trustworthy.systems/publications/. Citations in this document: §1.

[101] Mandayam K. Srivas, Albert John Camilleri (editors), *Formal methods in computer-aided design, first international conference, FMCAD '96, Palo Alto, California, USA, November 6–8, 1996, proceedings*, Lecture Notes in Computer Science, 1166, Springer, 1996. ISBN 3-540-61937-2. See [60].

[102] Neil Strickland, Nicola Bellumat, *Iterated chromatic localisation* (2019). URL: https://arxiv.org/abs/1907.07801. Citations in this document: §1.1.

[103] Andreas Strömbergsson, Anders Södergren, *On the generalized circle problem for a random lattice in large dimension*, Advances in Mathematics **345** (2019), 1042–1074. URL: https://arxiv.org/abs/1611.06332. Citations in this document: §5.3.

[104] Adam Topaz, *Alternating pairs with coefficients* (2023). URL: https://raw.githubusercontent.com/adamtopaz/CoeffAltPairs/main/main.pdf. Citations in this document: §1.1.

[105] Abraham Waksman, *A permutation network*, Journal of the ACM **15** (1968), 159–163. Citations in this document: §3.1.

[106] Yuting Wang, Ling Zhang, Zhong Shao, Jérémie Koenig, *Verified compilation of C programs with a nominal memory model*, Proceedings of the ACM on Programming Languages **6** (2022), 1–31. URL: https://flint.cs.yale.edu/flint/publications/nominalmm.pdf. Citations in this document: §1.1, §1.1.

[107] Edward Waring, *Problems concerning interpolations*, Philosophical Transactions of the Royal Society **69** (1779), 59–67. URL: https://royalsocietypublishing.org/doi/pdf/10.1098/rstl.1779.0008. Citations in this document: §3.4.

[108] Lawrence C. Washington, *Introduction to cyclotomic fields, second edition*, Springer, 1997. ISBN 0-387-94762-0. Citations in this document: §5.3.

[109] Freek Wiedijk, *The de Bruijn factor* (2000). URL: https://www.cs.ru.nl/~freek/factor/factor.pdf. Citations in this document: §3.4.

[110] Freek Wiedijk, *A synthesis of the procedural and declarative styles of interactive theorem proving*, Logical Methods in Computer Science **8** (2012). URL: https://www.cs.ru.nl/F.Wiedijk/miz3/miz3.pdf. Citations in this document: §4.8.

[111] Freek Wiedijk, *Step, a vi mode for HOL Light* (2023). URL: https://www.cs.ru.nl/~freek/step/step.tar.gz. Citations in this document: §4.8, §4.8.

[112] Michael J. Wiener (editor), *Advances in cryptology—CRYPTO '99, 19th annual international cryptology conference, Santa Barbara, California, USA, August 15–19, 1999, proceedings*, Lecture Notes in Computer Science, 1666, Springer, 1999. ISBN 3-540-66347-9. See [46].