NUMBER THEORY IN A PROOF ASSISTANT

JOHN THICKSTUN

1. INTRODUCTION

This paper discusses the development of elementary number theory in the proof assistant Lean, from the perspective of someone with background in mathematics, but no background in formal verification. I used this project to learn about proof assistants, and this paper shares some of what I learned along the way. A complete listing of the formal results discussed in this paper is available online at `https://github.com/jthickstun/lean`.

I chose to formalize elementary number theory because I know it well, it has a small ontology, and the results are constructive. By picking a topic with a small ontology, I was able to dive deep and prove substantial theorems, instead of building up a broad but shallow base of types and their elementary properties. The point about constructivism is more subtle and I don't profess to fully understand the motivations for constructive reasoning; my motivation at the outset of this project was simply that type-theorists take constructivism seriously, and I wanted consider proof-assistants (and by proxy, type theory) on their native turf.

The main result of this work is the fundamental theorem of arithmetic.

Theorem 1. *Every natural number has a unique prime factorization.*

To prove a claim in a proof assistant, we need to encode it in the formal language of the proof assistant. Here is an encoding of the fundamental theorem in Lean.

Listing 1. The fundamental theorem of arithmetic, extracted from numbers.lean

```
theorem prime_uniqueness (n : ℕ) : n ≠ 0 ⟹ ∃! l : list ℕ,
  plist l = tt ∧ sorted l = tt ∧ product l = n
```

It is not obvious a priori that this formal claim encodes the theorem presented above. We will discuss this phenomenon in more detail in Section 2.

We will work our way to the fundamental theorem of arithmetic in a more-or-less traditional way. The existence of a prime factorization is shown directly by (strong) induction. For uniqueness, we use the GCD to establish Euclid's Lemma and from there another strong induction gets us unique factorization. We carefully avoid variants of this argument that go through Bezout's identity, which requires reasoning about integers. We restrict our attention throughout to the natural numbers. This is usually to our advantage; restricting the ontology means there is less grunt-work to be done proving basic properties. But it occasionally hurts us (see section 6); I speculate that the fundamental theorem of arithmetic is about as far as we can go in number theory without feeling pressure to introduce a larger ontology, in particular $\mathbb{Z}$, and the quotient rings $\mathbb{Z}/n\mathbb{Z}$.

The remainder of this paper is structured as follows. In Section 2 we will dig deeper into the encoding of number theory in Lean. In Section 3 we consider the challenges of presenting formally verified mathematics in a human-readable form. In Section 4 we take a broader view of readability and consider the interplay between abstraction and formal verification. In Section 5 we consider some foundational questions about definitions. In Section 6 we will consider some of the more technical aspects of constructing formal proofs, and comment on the power of generality. In Section 7 we will consider some of the bugs that can arise in the workflow of formal verification, which leads to some more philosophical speculation in Section 8. Section 9 contains some concluding remarks.

2. A PROOF ASSISTANT AS A FOREIGN LANGUAGE

From a mathematician's perspective, working in a proof assistant feels like traveling to a foreign country where no one speaks your language. We have assertions that we wish to express, and we need translate these assertions into the formal language of the proof assistant. For every theorem or syllogism we wish to cite, we must learn its translation in the formal language: the logical contrapositive is `mt`, transitivity of inequality is `lt_trans`, and distributivity (on the right) of multiplication over subtraction is `mul_sub_right_distrib`. Like beginner students of language, we find that stating anything of consequence is an exercise in browsing the dictionary. Except unlike a dictionary, the assistant's library has no index; browsing becomes a search problem, constrained only by the high-level structure of the library that can guide the search towards more likely files and locations.

To some extent, this painful process is unavoidable. And like any language acquisition, it becomes easier over time as the most common vocabulary is memorized and the structure of the library is learned, enabling a more precise search the for more obscure results. But there is an aspect to this process that could be eased by a more powerful assistant: many of the most simple (and most commonly cited results) are ones that, in colloquial mathematics, would require no citation. Stronger tactics libraries that could justify the most basic algebraic and logical manipulations would help a lot here.

Like any pair of languages, there is some friction in the translation between colloquial mathematics and its formal counterpart. Compare the statement of the fundamental theorem of arithmetic (Theorem 1) to its formalization in Lean (Listing 1). Parts of this translation are unsurprising; syntax is introduced for quantification; explicit variables n and l are introduced for the number and prime factorization in question. Other choices are, while not necessarily surprising, certainly less mechanical. We explicitly spell out the definition of a prime factorization of n: a list of numbers, each a prime, with product n. There was no mention of lists in the colloquial statement of this theorem. And although lists are a natural enough choice (we could imagine speaking colloquially of a "list of primes") they are not our only option. We could, for example, have chosen to represent a prime factorization with a multi-set.

This latter observation brings us to the largest source of translational friction between the colloquial and formal statements of the fundamental theorem: the formal condition `sorted l = tt`. There is no mention, even implicitly, of this condition in the colloquial claim; it is an artifact of our representation of a prime factorization with a list (contrast this with the situation for multi-sets). We want to make a uniqueness claim "up to ordering" of the prime factorization. Once we commit to the list data type, we must be careful about the substance of our uniqueness claim.

Despite the trouble that lists cause for us, it is not clear that a different type–the obvious alternative is a multi-set–is a better choice for formalizing prime factorizations. There are two reasons to be skeptical of multi-sets. First, from the formal perspective, they are an inherently more complicated type than lists. The complications that stem from choosing a more sophisticated type likely outweigh the issues we face with lists. Second, from the view of colloquial mathematics, it seems that what we have in mind when we speak of a prime factorization is much closer to the formal list type than a multi-set.

Even after we commit to working with lists, there are several options for formalizing the fundamental theorem. One alternative is to explicitly encode the idea of "uniqueness up to ordering." We could introduce the syntax ∃∼!, with behaves like ∃! except instead of demanding a proof that `l = l'` for all `l'` that satisfy the existential condition, it would demand that `l` ∼ `l'`; i.e. `l'` is a permutation of `l`.

Listing 2. Alternative statement of the fundamental theorem of arithmetic; compare to Listing 1

```
theorem prime_uniqueness′ (n : ℕ) : n ≠ 0 ⟹
  ∃∼! l : list ℕ, plist l = tt ∧ product l = n
```

This alternate formalization is similar to the one presented in Listing 1; the former avoids introducing new syntax, but if uniqueness-up-to-ordering turns out to be a common pattern, then this syntax might be justified. We avoid explicitly mentioning sorts, in exchange for explicitly mentioning permutations.

Writing a proof of the fundamental theorem in the form of either Listing 1 or Listing 2 requires considerable work to develop the theory of sorted lists and permutations. Here is a (partial) catalog of the work that needs to be done:

- Define what it means for a list to be sorted.
- Define what is means for two lists to be permutations of each other.
- Define a sort function, for example insertion sort.
- Show that insertion sort returns a sorted permutation of its input.
- Show that a list of primes remains a list of primes under permutation.
- Show that permutation preserves the product of a list of primes.

We could use the following formalization to avoid all this work.

Listing 3. Another alternative statement of the fundamental theorem of arithmetic; compare to Listing 1

```
theorem prime_uniqueness″ (n : ℕ) : n ≠ 0 ⟹
   ∃ l : list ℕ, plist l = tt ∧ product l = n ∧ (sorted l ⟹
   ∀ l′ : list ℕ, plist l′ = tt ∧ product l′ = n ∧ sorted l′ ⟹ l = l′)
```

This is a fair encoding of Theorem 1, in the sense that the deductions required to prove it encompass the deductions a mathematician would present in a colloquially rigorous proof of Theorem 1. It requires a definition of a sorted list, but it doesn't require the construction of a sorted list or any discussion of permutations (which would never be presented in a colloquial proof of the fundamental theorem). It is satisfying to the extent that the audience is willing to accept that we can convert a list of primes into a sorted list, containing the same primes, with the same product. Most audiences would accept these facts implicitly, without even realizing the oversight. But when we lay out the assumptions explicitly and consider all the work required to establish these facts, this is maybe a little surprising!

3. READABILITY OF PROOFS

Recall the two-column proof format, often used to introduce young students to formal reasoning in a high school geometry class. Consider the fact that for any two natural numbers n, m, $(\operatorname{succ} n) * (\operatorname{succ} m) = 1 + m + n + m * n$. We can demonstrate this fact using a simple (albeit messy) two column proof:

	Statement	Reason
1	$(\operatorname{succ} n) * (\operatorname{succ} m) = (1 + m) * (1 + n)$	Definition of successor
2	$(1 + m) * (1 + n) = 1 * 1 + m * 1 + 1 * n + m * n$	Distributivity
3	$1 * 1 + m * 1 + 1 * n + m * n = 1 + m + n + m * n$	One is the identity
4	$(\operatorname{succ} n) * (\operatorname{succ} m) = 1 + m + n + m * n$	1, 2, 3, and transitivity

A two-column proof is explicit to the point of pedantry about both the current state of knowledge (the "statement" column) and the transformation rules that modify this state of knowledge ("reasons").

When writing mathematics for a more mature audience, we streamline our proofs by omitting the reasons from our proofs when they can easily be inferred by the reader. The proof above would become a simple calculation that demonstrates how we can transform the left-hand side to the right-hand side:

$$(\operatorname{succ} n) * (\operatorname{succ} m) = (1 + m) * (1 + n) = 1 + m + n + m * n.$$

The focus of a mathematical proof is on the true statements we have asserted, and explicit identification of reasons is saved for non-obvious deductive leaps; usually the application of a theorem or lemma.

Proof assistants, for technical reasons, take the opposite approach; a proof in a proof assistant typically writes out reasons in excruciating detail, leaving statement of the claims these reasons imply to the reader. Here is how the proof discussed above can be written in Lean.

Listing 4. A simple, unreadable proof in Lean

```
lemma foil (m n : ℕ) : (nat.succ m)*(nat.succ n) = 1 + m + n + m*n := by
rw [←one_add m, ←one_add n, left_distrib, right_distrib, right_distrib]; simp
```

Proofs like this can be followed in a live proof environment, where the assistant displays intermediate statements based on the position of the cursor in the document. But in a static environment such as this PDF, these proofs are basically unreadable. Steps can be taken to ameliorate this, usually by changing the goal internally with "have" statements and establishing many intermediate, explicit goals. Another option for algebraic proofs like this one is Lean's "calc" environment, which offers a syntax that looks more like a two-column proof.

Listing 5. A more readable proof in Lean, extracted from nat_extra.lean

```
lemma foil (m n : ℕ) : (nat.succ m)*(nat.succ n) = 1 + m + n + m*n :=
calc (nat.succ m)*(nat.succ n)
    = (1+m)*(1+n)                  : by rw [←one_add m, ←one_add n]
... = 1 * 1 + m * 1 + (1 + m) * n  : by rw [left_distrib, right_distrib]
... = 1 + m + n + m*n              : by rw [one_mul, mul_one, right_distrib]; simp
```

There is a stylistic choice to be made here. Should we optimize proofs for reading with the aid of the proof assistant? Or should we strive to write proofs that can be read in a static form?

4. PRIMAL AND DUAL INTERFACES

I found it helpful to write algorithms in a specific style that made it easier to prove formal results. Consider the following primality test.

Listing 6. A functional test of compositeness

```
def is_composite' : ℕ → bool := λ n : ℕ,
  let
    test_divisors := list.tail (list.tail (list.range n)),
    div_test := λ k : ℕ, zero_test (n % k)
  in
  list.foldr (bor) ff (list.map (div_test) (test_divisors))
```

This is a correct (albeit naive) primality test, but it is difficult to prove claims about it. The difficulty is that to prove something about `is_composite'`, we need to make claims about the behavior of `map` and `fold`. These are powerful, general functions: the sort of useful function that you would expect to find in a standard library. But the same power and generality that makes them so useful for writing programs also makes it difficult to prove claims about them.

I settled into writing functions in a bottom-up style without library calls.

Listing 7. A more direct test of compositeness, extracted from definitions.lean

```
def is_composite_aux (n : ℕ) : ℕ → bool
| 0 := ff
| 1 := ff
| (nat.succ m) := bor (zero_test $ n % (nat.succ m)) (is_composite_aux m)

def is_composite : ℕ → bool
| 0 := ff
| (nat.succ n) := is_composite_aux (nat.succ n) n
```

For a direct recursive implementation like this, we can prove everything we need with unadorned inductive arguments. But it is frustrating that I felt forced to completely give up on standard programming abstractions.

To support the use of these standard functions, we might ask a proof assistant to come packaged with a collection of lemmas to support proofs about programs that incorporate these functions, a sort of dual "lemma interface" to the corresponding primal function interface. An obstacle here is that the generality of these functions means we will want to use them in a wide variety of applications; it is not clear whether a dual interface of reasonable size could support all the uses we could imagine for a function. If a particular use of a function isn't supported by the dual interface, then we will need to write a difficult proof from scratch.

Perhaps what is needed here is a change of perspective. First, observe that we already encounter this problem in the primal domain, when a library has code that supports a feature we desire but its interface neglects to expose it. In this case we have to break the abstraction of the interface and dig into the internals of the system to get the feature we want. This is usually difficult, and it commits us to regular maintenance duties: future versions of the library may make internal changes that break our patches (without warning) even if the authors are responsible and committed to interface stability. For these reasons, good software usually treats an interface as a black box; we refuse to peer into its internals and write our software based on the documented interface.

It seems possible that we could treat the dual interface in the same way. A software library in this world would come in two parts: a primal programming interface, and a dual lemma interface. The primal interface would enumerate the calls you make into the library, and the dual interface would enumerate the guarantees you could make about these calls. In some ways, the dual interface would behave like documentation, or perhaps better than documentation, because it is guaranteed to be up-to-date. A conscientious library author would preserve interface compatibility in the form of the guarantees given by the dual interface, just like current author often pledge to preserve documented behavior.

One objection to this view of the future is that it might not be possible to write down a powerful enough dual interface to make the primal interface useful. I don't believe this holds up to scrutiny; it seems equivalent to claiming that it is impossible to write down the spec of a sufficiently general program. If this were true, it would doom the entire project of formal verification, and empirically this doesn't seem to be the case. It is certainly possible to write down a formal spec for the `map` and fold functions considered above. Nevertheless, in this world programmers will likely complain that certain true behavior of the primal interface isn't documented by the dual interface. For example, the dual interface for `map` might not declare that "map" processes a list in order, even if the implementation behaves this way internally. But this is no different than the situation for unexposed behavior in the primal interface, and making use of such undocumented behavior is already generally considered uncouth. Indeed, reliance on such behavior prevents certain optimizations (such as parallelizing `map`) that might otherwise be possible. Current software interfaces sometimes try to deal with this by explicitly renouncing certain behavior: "we make no guarantee about the order of processing of `map`'s input." A dual interface would formalize these pronouncements in an opt-in rather than opt-out manner: an interface only guarantees the behavior guaranteed by its lemmas.

A more practical objection is that the specs for programming libraries might be so abstract and general as to be useless; it would be easier for the programmer to write less general utilities from scratch than attempt to leverage a library. This is certainly the case when no duall interface is provided (as seen above in the example of a primality test). Designing practical, dual interfaces that support programming applications and verification in a streamlined way ought to be a high priority for the formal verification community.

5. DEFINITIONS

Keeping the theme of primality tests from the previous section, consider the following definition of a composite number:

Listing 8. The definition of a composite number, extracted from definitions.lean

```
def composite (n:ℕ) : Prop := n ≠ 0 ∧ ∃ a : ℕ, ∃ b : ℕ, a ≠ 1 ∧ b ≠ 1 ∧ n = a*b
```

Observe that we now have two definitions of a composite number: an extensional definition given by the relation defined in Listing 8 and and operational definition given by the algorithm described in Listing 7. We can prove an equivalence between these two definitions (the proof is not trivial; see decidable_relations.lean):

Listing 9. The link between composite numbers and primality tests, extracted from decidable_relations.lean

```
lemma computable_composite {n : ℕ} : is_composite n = tt ⟺ composite n
```

In the sense of logically equivalence, we could take either `is_composite` or `composite` as "the definition" of a composite number. Generally when we have two logically equivalent characterization of a phenomenon, it is admissible to take either as the definition. But in some sense (that I haven't been able to fully pin down) extensional characterizations seems like better, more fundamental definitions than operational characterizations. One perspective on this intuition is that the relation `composite` can be interpreted as a spec for the algorithm `is_composite`. If we take `is_composite` to be the definition of a composite number, then it is beyond the reach of formal verification; there cannot, a priori, be a bug in its implementation.

Perhaps the choice of the relational definition is uncontroversial for primality, but we can't conclude that extensional definitions are strictly preferable. Consider the following definition of the product of a list of numbers.

Listing 10. The definition of a product of a list, extracted from definitions.lean

```
def product : list ℕ → ℕ
| []        := 1
| (h :: t) := h * (product t)
```

In this case, an operational definition seems quite natural. Indeed, it is hard to image what an extensional definition of this product would even look like. One might argue that this is a recursive definition, and try to draw a distinction between recursive definitions and general operational definitions. But I am not sure how to formalize this distinction.

For a murkier case, consider the definition of (natural number) division. In elementary education, we are introduced to division operationally: n/k is the number of times k "goes into" n. Essentially, we define n/k and the remainder $n \% k$ by the following algorithm.

Algorithm 1 Natural number division

```
begin
    $a \leftarrow 0$
    while $a * k < n$ do
        $a \leftarrow a + 1$
    return $a$, $n - a$
end
```

Another variant of this algorithm starts the loop at n and repeated subtracts off k (this is nicer from a programming perspective, because we can write it as a recursion with a well-defined base case and easily prove that it halts, but it is probably not the algorithm most children run). Later in our education, we learn the more sophisticated long division. But this is the same game: definition by algorithm.

To write down an extensional definition of division, we first need to prove a lemma (see `division` in numbers.lean for a formal version of this lemma).[1]

Lemma 2. *Let $n, k \in \mathbb{N}$ with $k > 0$. There exist unique natural numbers q and r such that*

$$n = qk + r \qquad \text{and} \qquad 0 \le r < k.$$

[1]For reasons that I can only understand as a kind of narrowly-scoped insanity, many mathematicians like to call this *lemma* the division *algorithm*.

We can now write down the following definition of division:

Definition 3. *The dividend (denoted n/k) and remainder (denoted $n\%k$) are the unique natural numbers q and r given by Lemma 2.*

This extensional definition of division is a lot harder to swallow than the relational definition of a composite number. First, we need to write down a pretty abstract claim about the existence and uniqueness of certain numbers. Then we have to prove this claim. Only then do we get to give a definition of division. And finally, we need to show that a division algorithm (e.g. long division) correctly computes the dividend and remainder. In contrast, if we define division operationally then we get existence and uniqueness for free (the halting property guarantees existence and determinism guarantees uniqueness). And there's no need for a correctness proof because the definition makes the algorithm correct by fiat. It's not surprising then that we take the easy route when we present division to young children, but this does seem to inject a degree of arbitrariness and rote-learning into mathematical education. In this presentation there is no justification for long division, it's just an algorithm that must be memorized.

Let's now consider the consequences of this discussion in the context of a proof assistant. First, observe that if we restrict ourselves to constructive reasoning then we are forced to provide decision procedures for many of the concepts that we define extensionally. The reason is that we often want to argue by cases; for example, to prove a property for all numbers it suffices to prove it for prime numbers and also prove it for composite numbers. This argument relies, prima facie, on the law of excluded middle. But if we establish a decision procedure–as we do in Listing 9 for the `composite` relation–then we can split by cases on the result of this algorithm. We recover a limited form of excluded middle for decidable propositions.

In practice, proving the correctness of algorithms for decidable propositions can be so tedious that we will be tempted to work directly with an algorithmic definition: see decidable_relations.lean for the work required to establish the connection between the composite relation and primality test. In informal mathematics, we avoid this tedium by either appealing directly to excluded middle or hand-waving that we could construct a decision procedure if we wanted to. For once, there is a corresponding sort of hand-waving that we can do in the proof assistant to avoid tedium: ignore the extensional definition and work directly with an algorithm.

For our main results I find this approach unsatisfying. The theorems we would prove under operational definitions of their constituent terms could invite misleading interpretation: its easy to interpret these theorems as claims about the equivalent extensional definitions. But the formal theorems do no make these claims; this is a deduction from outside the formal system that relies on unverified claims about the correctness of the algorithms; the situation is similar to the one presented in Listing 3. This extra-formal distinction is illustrated dramatically if there is a bug in our implementation of one of our operational definitions. As we discussed at the outset of this section, there is no internal mechanism in the proof assistant to identify such mistakes. And in this case, there will be a profound mismatch between the (verified) theorems we prove and our interpretation of them.

In light of this discussion, the terms in the fundamental theorem proved in numbers.lean are given the most natural possible definitions: properties of lists are given recursive (i.e. operational) definitions, but the crucial property of being "a list of primes" (i.e. a `plist`) is defined using the extensional definition of a prime.

Listing 11. The definition of a plist, extracted from list_extra.lean

```
def plist : list ℕ → bool
| []        := tt
| (h :: t) := nt.irreducible h && plist t
```

Listing 12. The definition of a prime, extracted from definitions.lean

```
def irreducible (p : ℕ) :
  Prop := p ≠ 0 ∧ p ≠ 1 ∧ ∀ k : ℕ, (k | p) ⟹ (k = 1 ∨ k = p)
```

However, some of the auxiliary concepts used in the proofs leading to the fundamental theorem are given operational definitions. Most notably, I use Lean's built-in algorithmic definitions of natural number division and the greatest common divisor. It is perhaps surprising that the statement of the fundamental theorem of arithmetic does not depend on the definition of natural number division. The source of confusion is the distinction between natural number division and the "divides" relation. While division is given an operational definition, the notion that a number k divides another number n is given the following extensional definition.

Listing 13. The definition of "divides," extracted from the Lean library: library/init/algebra/ring.lean

```
instance comm_semiring_has_dvd : has_dvd α := has_dvd.mk (λ a b, ∃ c, b = a * c)
```

With this policy, extensional definitions for the terms in the theorems we care about and operational definitions for auxiliary results, we can be confident that our theorems encoded in Lean say what we think they do. And we save a lot of work in the auxiliary results. We should be mindfully suspicious of the auxiliary lemmas that state facts about operationally defined concepts. But this suspicion does not cast doubt on the proof of the main results; the claims made by the auxiliary lemmas are technically correct and useful; they just may not be interpretable in the most natural sense.

6. DEDUCTION BY GENERALIZATION

To prove a difficult theorem, mathematicians sometimes find that it is easier to prove a more general result and deduce the desired result as a special case. For example, Alexander Grothendieck settled the Weil conjectures by essentially re-writing the entire field of algebraic topology in the general setting of étale cohomology. Andrew Wiles proved Fermat's last theorem as a corollary of a far more general modularity theorem. The effectiveness of this meta-mathematical move should appear at least a little surprising: why should it be easier to prove a strictly more general statement?

This phenomenon appears in a more elementary context in the construction of rigorous induction proofs. When constructing an inductive proof, it is often useful to show a more powerful claim than the required result, because this yields a more powerful inductive hypothesis. Intuitively, if the claim to be show is to weak then it may not be strong enough to propagate itself in the inductive step; strengthening the claim makes our goal in the inductive step more difficult, but it also gives us a stronger hypothesis to work with. This phenomenon is common knowledge in the formal verification community, where proofs are formal and induction is the most fundamental tool for reasoning about inductive datatypes and recursive functions. As an example, consider a recursive definition of distance between two natural numbers.

Listing 14. A definition of distance for natural numbers, extracted from distance.lean

```
def dist : ℕ → ℕ → ℕ
| 0       m       := m
| n       0       := n
| (n+1)  (m+1)  := dist n m
```

Suppose we want to prove that $\mathrm{dist}(m, n) = \mathrm{dist}(n, m)$ (symmetry; see `dist_symm` in distance.lean for a formal proof). The only tool we have is the definition, and it is a recursive definition, so our proof must proceed by induction. By induction on m, we have the inductive hypothesis $\mathrm{dist}(m, n) = \mathrm{dist}(n, m)$ and we need to show that $\mathrm{dist}(m + 1, n) = \mathrm{dist}(n, m + 1)$. If $n = 0$ the result is easy (both sides equal $m + 1$) so suppose $n = k + 1$ for some $k \in \mathbb{N}$. Now our hypothesis becomes $\mathrm{dist}(m, k + 1) = \mathrm{dist}(k + 1, m)$ and we must show that

$$\mathrm{dist}(m + 1, k + 1) = \mathrm{dist}(k + 1, m + 1).$$

Applying the definition of distance (the third constructor) this is equivalent to showing that

$$\mathrm{dist}(m, k) = \mathrm{dist}(k, m).$$

But it is unclear now how to apply our inductive hypothesis.

Suppose instead we had taken the inductive hypothesis $\forall n, \mathrm{dist}(m, n) = \mathrm{dist}(n, m)$. The distinction between this generalized claim and the claim for an arbitrary n in the preceding paragraph is easy to miss in informal arguments. We now need to show that $\forall n, \mathrm{dist}(m+1, n) = \mathrm{dist}(n, m + 1)$. We can do this by demonstrating that $\mathrm{dist}(m + 1, n) = \mathrm{dist}(n, m + 1)$ for an arbitrary n. And as before, after considering the case $n = 0$, it remains to show that $\mathrm{dist}(m, k) = \mathrm{dist}(k, m)$. But now the stronger inductive hypothesis applies, and we are done. See distance.lean for many more examples of this style of inductive proof.

Here is another interesting and elementary example of the power of generalization that was exposed to me only by the careful reasoning enforced by formal verification. Consider the following proposition, where the $\% n$ operator denotes remainder mod n.

Proposition. *For all natural numbers x, y,and n, $[x \,\%\, n + y \,\%\, n] \,\%\, n = (x + y) \,\%\, n$*

Proof. By definition of natural number division, $a = n\left(\frac{a}{n}\right) + a \,\%\, n$. Using this identity to eliminate all terms with modulo operations and calculating, we have

$$[x \,\%\, n + y \,\%\, n] \,\%\, n = x - \left(\frac{x}{n}\right) + y - \left(\frac{y}{n}\right) - n\left[\frac{x - \left(\frac{x}{n}\right) + y - \left(\frac{y}{n}\right)}{n}\right]$$

$$= x - \left(\frac{x}{n}\right) + y - \left(\frac{y}{n}\right) - n\left[\frac{x + y - \left(\frac{x}{n}\right) - \left(\frac{y}{n}\right)}{n}\right] \tag{1}$$

$$= x - \left(\frac{x}{n}\right) + y - \left(\frac{y}{n}\right) - n\left(\frac{x + y}{n}\right) - \left(\frac{x}{n}\right) - \left(\frac{y}{n}\right)$$

$$= x + y - n\left(\frac{x + y}{n}\right) = (x + y) \,\%\, n.$$

□

This is a valid proof, in the sense that with sufficient care one can formalize an argument in a proof assistant that follows the path of these computations. But it is surprisingly tricky, requiring many auxiliary lemmas such as the following (this particular lemma is required to justify Equation 1).

Listing 15. An unexpectedly important algebraic identity

```
lemma add_sub {m n a b : ℕ} (hm : m ≥ a) (hn : n ≥ b)
 : m − a + n − b = m + n − a − b
```

What's going on here is that our proof attempts to do algebra with natural numbers and, as any mathematician will tell you, algebra works best in a closed number field. If our numbers were integers–if we had additive inverses–then the identity above would follow trivially by commutativity of addition. In informal arguments, we frequently pass to a more general number system when convenient, usually without even noticing. Historically this practice led to the introduction of complex numbers, when the mathematicians Cardano and Bombelli found it useful to reason with imaginary values as intermediate terms in algebraic computations, in pursuit of (real) solutions to cubic equations. To prove the proposition above, we would do better to prove it for integers and derive the statement for natural numbers as a special case.

7. UNSOUND DEDUCTION

When proving more complicated results in Lean, I fell into a workflow where I would start writing a proof, write down helper lemmas (without proof) that I needed for various subgoals, and return later to prove the helper lemmas. This usually worked out well, but the risk of introducing lemmas without proof is that it is possible to make assertions that are false. Consider the following definition of a sorted list.

Listing 16. A bad definition of a sorted list

```
def lmin : list ℕ → ℕ
| []          := 0
| (h :: t)    := min h (lmin t)

def sorted : list ℕ → bool
| []          := tt
| (h :: t)    := (min h (lmin t) = h) && sorted t
```

There is a bug in the definition of `lmin`: it always returns zero. This bug propagates into the definition of `sorted`, which will return true iff its input is `[]`. I didn't discover this until much later because I immediately wrote down the following lemma, with the intent to come back later and fill in the proof.

Listing 17. A simple lemma about sorted lists, extracted from list_extra.lean

```
lemma sorted_singleton (x : ℕ) : sorted [x] = tt
```

This (false!) lemma enabled me to prove a lot of results by reducing to a single-element list, which meant I didn't notice the bug with the empty list. So the bad definition stood for a long time and proliferated throughout my codebase. When I finally returned to prove `sorted_singleton` and realized the problem, I was pretty horrified.

But it turned out that the problem wasn't bad at all. First, I changed the definitions:

Listing 18. A correct definition of a sorted list

```
def lmax : list ℕ → ℕ
| []         := 0
| (h :: t)   := max h (lmax t)

def sorted : list ℕ → bool
| []         := tt
| (h :: t)   := (max h (lmax t) = h) && sorted t
```

Of course, this broke all my proofs, but the fix to the proofs was almost mechanical: I went through and replaced every max with a min, every lmax with lmin, flipped a bunch of $\leq$ inequalities to $\geq$, and replaced a few standard library invocations of lemmas regarding min with the corresponding ones regarding max. And all the proofs went through again. Recalling the discussion of interfaces from Section 4, I had essentially hidden the buggy definition `lmin` behind the dual interface `sorted_singleton`. Once I replaced `lmin` with a correct implementation `lmax`, `sorted_singleton` held true and so did the rest of my results.

This suggests that inconsistency isn't always the disaster implied by the principle of explosion. Under the false assumption of `sorted_singleton`, the deductive calculus permitted me to prove any claim I wanted. But I somehow managed to avoid writing any proofs that depended on this explosion of consequences; once I patched up the buggy definition, all my proofs became valid. This observation is congruous with Wittgenstein's comments about paradoxes. It's also congruous with the habits of the mathematical community, which generally works in a naive and implicit axiomatic framework, without concern for the foundational difficulties that motivate careful formalisms like Zermelo-Frankel.

A bold prognosticator might infer from these observations that if we ever found an inconsistency in ZF, it would be a non-event. Set theorist would work furiously for a few years to patch it up, and the broader mathematical community would proceed unfazed. I don't completely understand why this is the case, but I speculate that the answer has to do with semantics in some broad classical, or even Platonic, sense. Unlike a SAT-solver, a mathematician's proof-writing isn't directed by random or heuristic application of the deductive calculus. Proof-writing activity is largely undertaken in support of claims that we already believe for semantic reasons, making it less likely that we accidentally stumble into a contradiction and prove a false claim.[2] This also helps explain why it is possible to recover most of our reasoning when we "patch up" an unsound argument; there is nothing wrong with the objects under consideration, we just need to a better definitions and logic to correctly discuss them.

8. SEMANTICS

I'd like to wrap up with some thoughts about intuitionistic logic. Understanding this logic–and more generally, the ideas and motivation behind non-classical logics–was one of my main objectives for this project. I still don't fully understand it, so this section is likely the least coherent part of this paper. But I'd like to at least try to say something about logic.

Let's take a broad view of the constituents of a logic. A logic has a syntax consisting of well-formed formulas, axioms, and a deductive calculus for deriving theorems from axioms.

[2] Of course, the method of proof by contradiction inverts this situation. And, as every novice mathematician has learned, it is extremely easy to make a mistake in these arguments.

It also has a semantics consisting of models, and a satisfaction criterion that links models to formulas. Most logics seem to have some notion of completeness, a semantic concept that is defined with respect to a class of models: a well-formed formula is a tautology iff it is satisfied by every model, and a logic is complete iff it deduces every tautology. Logics also have a notion of soundness. We can define soundness semantically: a logic is sound iff every deduction is a tautology. But in many logics (e.g. classical, intuitionistic) we can get away with a purely syntactic definition of soundness: a logic is sound iff its deductive calculus derives no contradictions.

Classical first-order predicate logic is complete with respect to the classical model theory of structures. Intuitionistic first-order is obviously not complete with respect to these models (e.g. it does not derive "$p \vee \neg p$," which is satisfied by every structure). But it is complete with respect to Kripke models, which constructivists seem to view as a "more appropriate" semantics for constructive logic. (Interestingly, this completeness proof is apparently not non-constructive; Beth models are an alternative semantics for intuitionistic first-order logic, for which the completeness proof is itself constructive.)

I've been struggling to understand what could be meant by an "appropriate semantics" for a (sound) deductive calculus. One seemingly natural interpretation is that a semantics is "appropriate" for a deductive calculus if you can prove a completeness theorem for that deductive calculus for the given semantics. This view would endow any deductive calculus with an "internal semantics" for which it is, by construction, complete. This seems elegant, but I don't know if helps us understand what's going on any better.

As with any research project, I've walked away with more questions than answers. The Kripke models of intuitionistic logic seem frustratingly circular; the semantics of the logic themselves talk about deductions; there seems to be, in a sense, nothing more than deductions. Contrast this with classical model theory, which at least gestures at a broader Platonic universe of structures. As discussed in Section 7, some form of Platonism seems important for keeping our reasoning in check and intuitionistic semantics don't seem to help us understand this phenomenon. I don't know if this is a reasonable criticism of intuitionism, or if I'm asking the semantics to do too much. I also don't understand the connection between operational semantics with its notions of soundness (any typed program halts) and completeness (any program that halts has a type) and logical semantics with the soundness and completeness notions discussed above; it seems that there ought to be some connection via the Curry-Howard isomorphism. For that matter, I also don't understand how the calculus of constructions offered by Lean compares to other settings like first-order intuitionistic predicate logic. The calculus of constructions seems much stronger.

9. CONCLUSION

Let's try to draw some broad constructive lessons from the observations in this paper, with the aim of making proof assistants a better and more popular tool. For outreach to the mathematical community, it seems important to develop strong tactic libraries that minimize the language barrier between written and formal proofs. For mathematicians themselves: there ought to be a discussion about the workflow we want to foster for a future generation of mathematicians. Do we imagine that future mathematicians will spend their working lives reading and writing mathematics in a proof assistant? Or do we imagine proof assistants as ancillary tools for attaining results that we then export to other media? These alternate visions of the future have profound influence on how we will want to optimize the language of proof assistants for readability. For outreach to programmers, there needs to be a clear methodology for constructing dual interfaces. Until we can convincingly demonstrate an ability to construct these interfaces, and offer programmers a power standard library equipped with a dual interface, widespread adoption of formal verification practices seems unlikely: it simply does not scale.

On a less concrete note, I would like to see more convincing stories about some of the more speculative topics discussed in this paper. What should we make of the distinction between operational and extensional definitions? Under what circumstances is one or the other more appropriate? It is really weird that we can sometimes prove a more general claim more easily than special case–why is that? Sometimes, we can "project down" a proof of a general claim to a more specific claim, by reducing each step of the argument to a specific claim; other times, as for formal inductive proofs, increased generality seems essential: what is going on here? Finally, is there a more convincing story to be told about the semantics of logic, where logic is understood broadly to include classical, intuitionistic, and other deductive reasoning systems? Is there a story that explains why humans are able to perform logical and mathematical reasoning so reliably?