My unusual hobby (stephanboyer.com)

856 points by curryhoward 7 months ago | hide | past | web | favorite | 147 comments

j2kun 7 months ago [-]

> What's really amazing to me is that Stephen Kleene probably proved this without the help of a computer, but I was able to verify it with the highest possible scrutiny. It's as if he wrote an impressive program without ever having run it, and it turned out not to have any bugs!

This offhand comment (which we can all forgive) makes it seem like mathematics happens in a vacuum. I can understand the temptation to think that way, and I hope I can make it clear that this is not the case.

General theorems typically grow out of a set of representative examples of interesting behavior (unit tests, if you will), are judged by their relationship to other mathematical objects and results (integration tests), and are reviewed by other mathematicians (user acceptance tests).

Mathematicians don't just write down a proof and cross their fingers. The proof is often one of the last steps. There is a high-level intuition that explains why a theorem or proof sketch is probably true/correct before a formal proof is found. These explanations are usually compact enough to help one re-derive proof ("X doesn't have enough 'space' to fill Y, and Z tracks that gap", or something like that), but aren't formal enough to put into any formal logic system I know of.

curryhoward 7 months ago [-]

(OP here.) Fair enough—thanks for the thoughtful clarification.

By the way, I'm a big fan of your "Math ∩ Programming" articles. People who found my blog interesting will certainly think the same of yours: https://jeremykun.com/

1000units 7 months ago [-]

You're a class act.

aswanson 7 months ago [-]

Big fan of his, and your work as well.

edanm 7 months ago [-]

One very interesting experience for me was recently relearning set theory from a textbook. It talked a bit about the history of the Schröder–Bernstein theorem, a fundamental (and relatively "simple" sounding) result in Set theory.

What I found interesting was that this originally stated, but not proved, by Cantor. Then a few semi-flawed attempts at proofs were made (flawed in the sense that they relied on other axioms like choice, or that they had errors).

These are leading mathematicians, including Cantor, who basically invented Set Theory from scratch. And they went through many years of struggle to arrive at a proof of a result. Which you now learn in an introductory class.

It really put into perspective for me just how *hard* some of these things are, and made me feel less bad when I struggle with some mathematics.

jacobolus 7 months ago [-]

The reasons such proofs are non-obvious is because they are proofs about the relative “size” of hypothetical infinite objects of a type we can never actually grapple with in any physical way even in principle, but only posit as a thought experiment, based on invented axioms in an invented logical system. There’s no concrete computation involved (or even possible) in this kind of context, and no practical examples. So everything must be done in a purely abstract and formal way. Without any examples to test, it’s hard to notice gaps in logic.

According to Wikipedia this particular theorem was proved almost immediately (but not published) by Dedekind, and then 10 years later proved by a 19 year old student in Cantor’s seminar. It’s not clear whether more than a handful of people cared or worked on it in between.

As for Cantor being a leading mathematician of the day: most of the other mathematicians of his time regarded this whole mess to not be mathematics at all. Poincaré called Cantor’s set theory a “disease”. It wasn’t until several decades later that a broader group of mathematicians decided that roping in set theory allowed them to conveniently hand-wave away (“oh, the set theorists will take care of that part”) a lot of thorny questions, letting them get on with their work as they had before, but now with an deflective answer whenever anyone asked such questions. ;-)

daxfohl 7 months ago [-]

Which even amplifies the observation. Not only is the process of proving a theorem more complex than it looks in retrospect, but the whole process of deciding whether a theorem or even a whole field of math is even useful or not can be driven by chance over a time span of decades. What is now "topology" was kicked around for many years with many approaches before a general agreement that open sets formed the most useful basis (heh). Advanced textbook authors regularly have to cut topics from subsequent editions of the book, because what seemed promising 30 years ago turned into a dead end. So math is far from the discrete list of topics and answers that it seems to be when you're picking out classes in your syllabus.

kkylin 7 months ago [-]

For anyone interested this seems to be a reasonable summary: https://en.wikipedia.org/wiki/Controversy_over_Cantor%27s_th...

runeks 6 months ago [-]

If something that looks obviously true is hard to prove using some given language, couldn’t that be a sign that we’re using the wrong language to construct the proof?

Could there exist a language where these sorts of proofs become easier to write, because that language captures the problem in a better way?

eqmvii 7 months ago [-]

> Mathematicians don't just write down a proof and cross their fingers.

On the flip side, this is exactly what a lot of lawyers do when writing contracts. Not necessarily maliciously or negligently, but it can still make for fun questions of interpretation when things don't go as expected.

jacquesm 7 months ago [-]

There is a nice parallel between the legal profession and programming. Programmers write software in such a way that they try (if they're any good) to reduce the number of assumptions made and the number of bugs and the ambiguities in their code. Failure to do so results in undefined behavior, crashing code and in internet facing code in possibly being hacked.

Lawyers write code into contracts. Good lawyers try to do so by removing ambiguity where ever they can. Failure to do so results in court cases where that ambiguity is resolved, and in some cases will result in damage for the party that originally contracted them to write the code.

Neither lawyers nor programmers are legally liable for such fuck-ups.

sheltron 7 months ago [-]

I used to think the same thing, until I learned that there are entire chapters of contracts textbooks on how to use and manage ambiguities effectively. a well-written contract many times has intentional ambiguities meant to limit liability in certain cases, or to use as fodder in negotiations.

inopinatus 7 months ago [-]

The law is not a programming language. Believing so is a common misconception amongst engineers, but representing it as such is likely (as I have said in this forum before) to lead to disappointment, frustration, anger, needless bickering, extended conflict, and vexatiously long, hard to read, and mostly unenforceable contracts.

> amazingman 6 months ago [-]
>
> It's a useful metaphor, if used judiciously. The law may not be programming, but it could be argued that it is engineering.

> mcguire 7 months ago [-]
>
> ... like a lot of meetings with engineers. Particularly standards meetings.

Splines 7 months ago [-]

I've always wondered if it were possible to express legal things via code. I imagine that there's a lot of ambiguity that needs "filling in" by a human, but there must be some set of legal arguments that can be literally codified.

Being able to run test cases through such a construct would be immensely useful - for example, how would changes to health care law impact someone? They could have unit tests that compare different outcomes and make it easier to understand how a new law would change things. People could ask very precise questions about how things would impact them.

> marcoperaza 7 months ago [-]
>
> Very few things in law are truly reducible to some hard, bright-line rule. There are conflicting interests all worthy of consideration and that therefore defy a single objective rule. So even in the presence of extensive precedent, there is usually a grey area requiring case-by-case judgment based on intuitions of fairness and equity.
>
> Say you own an apartment building with a view of the ocean and I own the plot directly between it and the ocean. I threaten to build a wall that blocks your building's view of the ocean unless you buy my plot for 10x more than its market worth. Should that be legal? Say you succumb to my demand. Should you be able to then sue me and recover the excessive payment? I think many would have the intuition that my threat should not be legal and any resulting contract not valid, since it was extracted by duress.
>
> But how about Firefox selling its search bar to Google for billions of dollars, carrying the implicit threat to use a different default provider should Google not pay. That seems more fair than the previous example, but why?
>
> And how do you formulate that difference into a hard rule, without relying on human beings' (i.e. judge and jury) intuitions of fairness and equity?
>
> Here is what the Restatement of Contracts has to say:
>
> ```
> (1) A threat is improper if
>    (a) what is threatened is a crime or a tort, or the
>        threat itself would be a crime or a tort if it
>        resulted in obtaining property,
>    (b) what is threatened is a criminal prosecution,
>    (c) what is threatened is the use of civil process
>        and the threat is made in bad faith, or
>    (d) the threat is a breach of the duty of good faith
>        and fair dealing under a contract with the
>        recipient.
> (2) A threat is improper if the resulting exchange is
>     not on fair terms, and
>    (a) the threatened act would harm the recipient and
>        would not significantly benefit the party making
>        the threat,
>    (b) the effectiveness of the threat in inducing the
>        manifestation of assent is significantly
>        increased by prior unfair dealing by the party
>        making the threat, or
>    (c) what is threatened is otherwise a use of power
>        for illegitimate ends.
> ```
>
> Look at all of the subjective terminology: "bad faith", "good faith", "fair dealing", "fair terms", "significantly".
>
> And look at (2)(c), "otherwise a use of power for illegitimate ends". What does that even mean? It's basically a surrender, acknowledging that it is impossible to formulate a rule *ex ante* that totally captures our notions of fairness as they should be applied in every possible situation.
>
> > marcoperaza 7 months ago [-]
> >
> > >*Very few things in law are truly reducible to some hard, bright-line rule.*
> >
> > My language is a little sloppy here. There are plenty of legal rules that lawyers would call bright-line rules. But they'll usually still fall short of something that can evaluated in software, as a programmer might expect of a rule so-described.

evincarofautumn 7 months ago [-]

On a flight to London, I once spoke with an intellectual property lawyer about my pet idea for machine-checked law, at least limited to the domain of contract law.

He explained some of the many reasons why that's an exceedingly difficult problem. Together we reasoned that it would be possible to implement *some* degree of machine checking and automation, but that an expert human would always need to review the results, because it's humans who interpret the laws & contracts.

> pdm55 7 months ago [-]
>
> One legal term that has always stumped me is the meaning of "beyond reasonable doubt". It's not a phrase that we use in everyday speech, so its meaning doesn't come readily to mind. Apparently, it means something less than "absolutely certain" and something more than "reasonably certain". What's worse is that judges cannot attempt to define it for the jury. Doing so can lead to a mistrial. https://www.ruleoflaw.org.au/beyond-reasonable-doubt/

mkagenius 7 months ago [-]

A small step towards that is Smart Contracts built on Ethereum.

marcoperaza 7 months ago [-]

>*Neither lawyers nor programmers are legally liable for such fuck-ups.*

Lawyers can be held liable if they do a horrible enough job. Likewise for programmers, and there are a lot of people advocating for strict liability for software defects in commercial products/services.

jacquesm 7 months ago [-]

> Lawyers can be held liable if they do a horrible enough job.

They can be but it rarely happens. The most likely outcome is that another lawyer (or even the same one) will send you another large bill to try to fix the problem. This too is roughly analogous to what happens in programming.

whatshisface 7 months ago [-]

Right, but convincing explanations can also be invented for things that aren't true. The true power is in the formalism, which allows you to use your untamable and untrustworthy intution to discover *facts*.

mcguire 7 months ago [-]

And this...

"*To give you an idea of what an actual proof looks like in Coq, below is a proof of one of the easier lemmas above. The proof is virtually impossible to read without stepping through it interactively, so don't worry if it doesn't make any sense to you here.*"

is why Coq is not one of my favorite tools.

A proof consists of two things: the "proof state"---a statement of what you know at a given point--and a sequence of "proof operations"---the manipaulations that move from one proof state to the next.

Most mathematical proofs you have ever seen have been presented as a list of proof states, with the operations either explained in comments or taken to be obvious.

Coq (and most other similar systems) present the proof as a sequence of proof operations, with the state invisible unless you're going through it in a Coq IDE.

On one hand, that's the same thing as a program: a Coq proof is a program for creating the proof. Unfortunately, on the other, I've never been able to read Coq proofs the way I can read programs. Probably a lack of experience, I guess.

On the other hand, Coq is a *damn* fun computer game.

curryhoward 7 months ago [-]

> Unfortunately, on the other, I've never been able to read Coq proofs the way I can read programs. Probably a lack of experience, I guess.

I don't think it's a lack of experience on your part. I worked with some people at MIT who were far better at Coq than I was, and they admitted the same thing. Coq proofs are unreadable, even when all the hypotheses and variables are given thoughtful names.

But, as you found, Coq proofs are not meant to be read; they are meant to be stepped through. I'm not sure if there is really a better alternative: if the formal proofs were made easier to read, they'd probably be much longer. And they can already get quite long compared to traditional paper proofs.

I think the right strategy is to break large developments into small lemmas, such that each proof is short and borderline obvious (similar to structuring a computer program to be self-documenting). Also, more experienced Coq users heavily invest in proof automation (building "tactics" in Coq) to make proofs short and sweet. I don't have much experience on that front, though.

carterschonwald 7 months ago [-]

I worked with Adam one summer before he had that coq textbook, he's definitely a wizard of proof automation in coq. I only much more recently learned how ltac case expressions also backtrack!

kmill 7 months ago [-]

For the benefit of people without specific knowledge of people at MIT: Adam is Adam Chlipala http://adam.chlipala.net/

codeflo 7 months ago [-]

Is that partially a tooling problem, i.e. would it help or even be possible to show those intermediate "proof states" (as used by the grandparent) live in the IDE while you type?

mcguire 7 months ago [-]

Stepping through the proofs in the IDE is the only way to understand them. But that is like stepping through a program in a debugger---it's inefficient if you have a lot to understand.

abecedarius 7 months ago [-]

It sounds like a similar difficulty to reading Forth, and you're proposing a solution like what Forth programmers do. I can't help wondering if we can't make up a more readable language instead.

mcguire 7 months ago [-]

Unfortunately, smart tactics make proofs even harder to read. :-)

qznc 7 months ago [-]

I you prefer readable proofs, try Isabelle or Lean.

Zimm_i48 6 months ago [-]

You can certainly do readable proofs in Coq the way you would do them in Isabelle, but this requires stating intermediate states explicitly (with assert e.g.) and Coq users rarely do that (mainly because readability is not their goal, I guess).

pron 7 months ago [-]

Or TLA+.

mcguire 7 months ago [-]

Lamport's structured proofs... should be easier. But I haven't had enough experience to say.

runeks 6 months ago [-]

Are there languages that model proofs as pure functions? I.e. instead of modifying some implicit state, you pass existing proofs as arguments to other proofs that define these existing proofs as assumptions to itself.

So, a proof would be a pure function, taking a number of assumptions as arguments, and in order to write this proof you must first pass proofs of the assumptions to it as arguments (and then you can reference these proven assumptions in the body of the proof when needed).

Wizek 7 months ago [-]

Couldn't it be possible to unfold a proof by printing all the subsequent proof states one after the other? Wouldn't that be easier to follow?

jmite 7 months ago [-]

What is your favorite tool? Agda? I've been finding it much nicer to work with, but the ecosystem is just not there yet.

david-gpu 7 months ago [-]

I had to take a semester on formal proofs using Coq, the same tool the article talks about.

Putting aside their steep learning curve, formal proof methods do not guarantee that the code you've written is bug free. They only guarantee that the code follows the requirements you defined given the conditions you also set on your inputs. You can think of it as a mathematical proof of your postconditions will hold given that your preconditions are true.

And you know what? People do make mistakes establishing those preconditions and postconditions all the time, and the tool is not going to be of any help. You've proven your code, but it may not do what you actually wanted it to do.

During a lecture I saw a tenured professor, who was a well-respected mathematician and expert in Coq, make some rather obvious mistakes while establishing those preconditions and postconditions. His proof was valid but worthless.

When you add that writing those proofs is actually rather time consuming, you end up with a niche product with little real-world applicability, except possibly for safety-critical environments.

seanwilson 7 months ago [-]

> Putting aside their steep learning curve, formal proof methods do not guarantee that the code you've written is bug free.

People seem to always bring this up but what's better? Verified code is about as close as you're ever going to get to bug free.

If you're doing a large proof, getting the specification wrong and not eventually noticing while working on the proof isn't common. You'll likely have something that isn't provable and get proof goals that don't make any sense. I don't think it's a common problem for mathematicians either that they end up proving the wrong thing.

I haven't heard of any examples of complex verified programs where later someone found a huge flaw in the specification. It would be like writing a set of unit tests and a complex program that passed them which had behaviour that goes against what you wanted.

wolfgke 7 months ago [-]

> I haven't heard of any examples of complex verified programs where later someone found a huge flaw in the specification.

Look at

> https://www.krackattacks.com/

"The 4-way handshake was mathematically proven as secure. How is your attack possible?

The brief answer is that the formal proof does not assure a key is installed only once. Instead, it merely assures the negotiated key remains secret, and that handshake messages cannot be forged.

The longer answer is mentioned in the introduction of our research paper: our attacks do not violate the security properties proven in formal analysis of the 4-way handshake. In particular, these proofs state that the negotiated encryption key remains private, and that the identity of both the client and Access Point (AP) is confirmed. Our attacks do not leak the encryption key. Additionally, although normal data frames can be forged if TKIP or GCMP is used, an attacker cannot forge handshake messages and hence cannot impersonate the client or AP during handshakes. Therefore, the properties that were proven in formal analysis of the 4-way handshake remain true. However, the problem is that the proofs do not model key installation. Put differently, the formal models did not define when a negotiated key should be installed. In practice, this means the same key can be installed multiple times, thereby resetting nonces and replay counters used by the encryption protocol (e.g. by WPA-TKIP or AES-CCMP)."

I personally would consider this as a clear example of a huge flaw in a specification where a correctness proof was done on.

seanwilson 7 months ago [-]

> I personally would consider this as a clear example of a huge flaw in a specification where a correctness proof was done on.

Awesome, thanks for the recent example. My point is people seem to always bring up that specifications may have flaws as if this makes verifying code pointless. The guarantee of correctness is still an order of magnitude better than e.g. using unit tests and I'd wager it isn't common that specifications are wrong in a critical way. It's difficult to make a mistake specifying that a program should not have buffer overflows and out of range errors for instance.

The WPA2 flaw is an example of a specification missing an important property as opposed to a property being described incorrectly. People seem to jump on the latter the most I find as if it's common.

wolfgke 7 months ago [-]

> The WPA2 flaw is an example of a specification missing an important property as opposed to a property being described incorrectly.

If a specification misses an important property it is in my opinion by definition an incorrect description (in particular in this case when this missing property indeed leads to a security problem).

> I'd wager it isn't common that specifications are wrong in a critical way.

> People seem to jump on the latter the most I find as if it's common.

Let me present you an hypothesis: Currenly few things are really formally specified. And if it is done, there are usually very smart people behind the project. Which by definition makes it rare that they contain lots of critical errors. On the other hand, if these methods were applied "large scale" (i.e. also by much less skilled programmers who don't have a lot of deeper knowledge about more than basic computer science topics), the rates of errors in specification would be much higher and I believe this could become a not-that-uncommon topic.

Of course I nevertheless believe that if a lot more properties were formally specified and correctness proofs were done, the error rates of programs would nevertheless go down by magnitudes. But I don't believe that formal specifications and formal correctness proofs are a panacea. What I rather consider as sad is that it seems to be that if formal specification and machine-checked proofs were applied "by nearly every program", we actually slowly get out of ideas what kind of tools we can develop to decrease the error rates even more...

seanwilson 7 months ago [-]

> If a specification misses an important property it is in my opinion by definition an incorrect description (in particular in this case when this missing property indeed leads to a security problem).

I agree that missing important properties is very bad, but incorrectly stating a property, writing a complex program that matches it and completing the proof all without noticing sounds unlikely to me.

> On the other hand, if these methods were applied "large scale" (i.e. also by much less skilled programmers who don't have a lot of deeper knowledge about more than basic computer science topics), the rates of errors in specification would be much higher and I believe this could become a not-that-uncommon topic.

Hmm, so from my experience, if you're having trouble writing correct specifications you quickly end up getting stuck when writing proofs. Your specification won't match your program and your program won't match your specification so you'll be unable to finish your proof. If you somehow luck out through this a few times, you eventually won't be able to prove something else because a previous specification/theorem is wrong.

> But I don't believe that formal specifications and formal correctness proofs are a panacea. What I rather consider as sad is that it seems to be that if formal specification and machine-checked proofs were applied "by nearly every program", we actually slowly get out of ideas what kind of tools we can develop to decrease the error rates even more...

Besides having machines read our minds, I'm not sure where you can go from formal methods except inventing languages that make specifications easier to accurately describe. When it's still a ongoing debate if strongly typed languages are safer than dynamically typed languages we have a very, very long way to go so this isn't an issue yet...

taneq 7 months ago [-]

> Verified code is about as close as you're ever going to get to bug free.

Barring compiler errors (which despite frequent protestations here, you're highly unlikely to encounter in day-to-day coding) the output of a compiler is a 'verified' translation of your source code. This doesn't translate to any guarantee of correctness and neither do proofs.

Turing_Machine 7 months ago [-]

"People seem to always bring this up but what's better?"

"Better" is a slippery term. Code doesn't exist in a vacuum, but rather in an external world that has other constraints (e.g., time and budget).

If using formal proofs means that you go through one development cycle while your competitor is iterating three, four, or five times, your milkshake is going to be drunk.

marcosdumay 7 months ago [-]

> Verified code is about as close as you're ever going to get to bug free.

Code verified by legible proofs is safer than merely verified. The quality of the verification language matters, it's not only the programming language.

seanwilson 7 months ago [-]

> Code verified by legible proofs is safer than merely verified. The quality of the verification language matters, it's not only the programming language.

If the specifications are human legible though, why do you care if the proof (which is checked for you by the machine) is legible? The main point of formal methods is to remove any doubt there are errors in the proof so it's an order of magnitude safer than relying on a readable hand written proof.

If you want a formal proof that is as readable as a hand written proof, you'd have to rely on a lot of automation going on in the background as every proof step must be traced back to basic axioms so complete proofs are pretty much always going to verbose and unreadable.

wolfgke 7 months ago [-]

> If the specifications are human legible though, why do you care if the proof (which is checked for you by the machine) is legible?

Because it is a goal of any science not to make a correct model, but also to get a high-level understanding what this model is about. Feel free to call this high-level understanding "what it is really about a "meta-property" of the model, which we also would like to have.

voxl 7 months ago [-]

Nothing guarantees that code is bug free. We will always have to interact with hardware, and although we can have high confidence in the hardware the physical world has a habit of changing out from under us.

Your argument is very strange. We use type systems not because we think we'll write perfect code, but because we know it will reduce the likelihood of making mistakes in our code (at least, when that type system is sound).

It seems like you're against writing proofs that your code is correct because you think you'll do better. I have a very hard time believing that, seeing as you probably aren't writing your own assembly (that beats a compiler) most if any of the time.

auggierose 7 months ago [-]

Formally proving something is just very expensive. The difference between the cost of an informal understanding of the correctness of your code, and the cost of formally proving it in a theorem prover is usually at least one order of magnitude (if you are lucky).

A compiler though with a type system actually REDUCES your cost. So these are very different beasts.

somenewacc 7 months ago [-]

Type checking your code is, in a sense, a formal proof. That is, your types encode theorems, and the compiler check them.

The article actually alludes to this, linking to this: https://en.wikipedia.org/wiki/Curry%E2%80%93Howard_correspon...

auggierose 7 months ago [-]

Yeah, I should have been more specific in my comment: I don't mean type systems like in Agda or Idris or Coq ..., but where writing down the types is not a big deal, because otherwise you are doing theorem proving with all the associated costs.

In short: I like both my type systems and my theorem proving without any involvement of Curry-Howard :D

seanc 7 months ago [-]

"Beware of bugs in the above code; I have only proved it correct, not tried it. " - Don Knuth

dllthomas 7 months ago [-]

Of course, Knuth's proof was not mechanically checked...

david-gpu 7 months ago [-]

The problem is that these formal proof systems are often marketed as tools to write "bug free code". You will see statements along the lines of "this code is proven to be bug-free", etc. They over-sell and under-deliver.

When you also consider that performing these proofs takes a very significant amount of time, you end up with something with very limited applicability for the vast majority of software development scenarios.

That's all I'm saying. Please don't argue against a strawman.

icebraining 7 months ago [-]

Can you provide an example of a system being marketed as a tool to write "bug free code"? I saw nothing of the sort in the sites of Coq, TLA+ or Isabelle. Are you sure you're not the one arguing against a strawman?

david-gpu 7 months ago [-]

In these comments there are several examples of folks selling an algorithm as "proven" and later turning out to be buggy.

icebraining 7 months ago [-]

Right, because they are proven. It's your assumption that proven == bug-free that is incorrect.

dtornabene 7 months ago [-]

You mean like, say, seL4? Pretty sure that kernel is seeing some very real "applicability".

qznc 7 months ago [-]

Related: Hoare looks back at formally proving programs correct https://m.cacm.acm.org/magazines/2009/10/42360-retrospective...

Someone 7 months ago [-]

"formal proof methods do not guarantee that the code you've written is bug free. They only guarantee that the code follows the requirements you defined given the conditions you also set on your inputs."

I think you are mixing “bug” with “out of spec”. If you ask me to write a perfect chess program, you can’t say it’s a bug if it can’t play go.

Bad or incomplete requirements are a problem, but that’s a different problem from “bugs”.

dwaltrip 7 months ago [-]

People often use the term "bug" in a more broad sense to mean "the system is behaving in an unexpected or undesirable manner". The "spec" may have not even covered the issue in question -- no one thought of that edge case.

Either way, for Coq programs, it sounds to me like specifying the preconditions is part of the implementation, so the term "bug" seems especially apt

ImSkeptical 7 months ago [-]

If the purpose of the program is to entertain go players, then I'd say the bug is in the spec, which describes the wrong game.

mcguire 7 months ago [-]

Yes and no.

Verification != validation, and there is no technique on the "make better code" side that will help with the validation side, not proofs, not unit tests, nada.

On the other hand, there is a certain amount of benefit of "double entry bookkeeping"; having a statement of what it's supposed to do separate from what it actually does. And you get bonus points if they're linked so that you know what it does is what it says it does.

It may be that there is 'little real-world" applicability' for the full deal as things stand now. But things like fancy type systems are trying to continually push usability towards proofiness.

frik 7 months ago [-]

I had to take such a formal proofs course too, just with a different tool that had even steeper learning curve and rough edges (prof used us as alpha tester of his new tool), and came to the same conclusion as you. It seems very valuable to formal proof specific small sub-sets or rather critical components like device drivers, like e.g. Microsoft is doing it for some time. I am sure it will become more popular, when the tools and languages get more mature and easier to grasp for average joe developers in the next years.

pron 7 months ago [-]

> And you know what? People do make mistakes establishing those preconditions and postconditions all the time, and the tool is not going to be of any help. You've proven your code, but it may not do what you actually wanted it to do.

This always comes up when formal methods are discussed, and I've never been able to understand this. The goal isn't really to create a 100% correct program, but to significantly reduce the chance of bugs.

And while writing pre- and post-condition is one way, another is to write global correctness propositions for the entire program that are easy to inspect for correctness (e.g, a two-line formula saying "the database is always consistent", or "no data is lost even if a node crashes"). Usually, these properties are verified not against the program itself, but against some high-level spec serving as a blueprint, and while there is no formal proof that the code follows the blueprint, the same is true for buildings, and blueprints can help a lot. Use those blue-prints alongside code-level pre-/post-conditions, and you have yourself a highly reliable piece of software.

The vision is to be able to verify those global correctness properties all the way down to the code (and even to machine code). This is called end-to-end verification, and has only been accomplished for programs of (very) modest size and complexity, and even then with great effort. But even without end-to-end verification, you can greatly increase your confidence in your program, at a cost that is not only affordable, but may even save you some money.

nlperguiy 7 months ago [-]

Yet there's a movement here https://github.com/HoTT/HoTT where they do everything in Coq.

throwawayjava 7 months ago [-]

"useless except when it really fucking matters"

semi-extrinsic 7 months ago [-]

There's many physical examples of systems like this as well.

Take e.g. a redundant interlock system that disallows opening a chassis unless the power is off and you've inserted a special key. Useless on a juice press #cough#juiceroo#cough#, but very important on a 20 W UV laser.

Or a six-point safety harness. I'm not going to put them in my daily driver, but I wouldn't get into a racecar without them.

yaseer 7 months ago [-]

If you're a programmer that's not tried theorem proving before, I highly recommend giving it a go. It gave me the same 'thrill' I got when I first started programming as a 14-year old and realised a new way of thinking. I find it sharpens your mind in a new, and different way - that's really an exciting feeling.

I prefer Isabelle myself, as I find its proofs more readable in a traditional mathematical sense. Coq however, is probably easier for most programmers to experiment with.

curryhoward 7 months ago [-]

> I prefer Isabelle myself, as I find its proofs more readable in a traditional mathematical sense.

Just curious, what makes Isabelle easier to read? I haven't tried it.

yaseer 7 months ago [-]

Isabelle comes with a proof language layer Isar, designed specifically for readability: https://isabelle.in.tum.de/Isar/

The structure of Isar proofs then more closely resemble a traditional mathematical proof, with a formalisation close to natural language. You can see the example from Wikipedia:

https://en.wikipedia.org/wiki/Isabelle_(proof_assistant)

As you can see, the overall structure of the proof is clearer, and you have less dependency of seeing the proof state to understand the proof.

The downside to Isabelle is that it introduces a dual syntax, with certain sections written in quotes. There's more to learn syntax and concept wise, but I think it's worth the extra effort for a clearer proof.

EDIT: Linked to Wikipedia page for formatting.

kelukelugames 7 months ago [-]

Please use a more informative title and name the hobby. The current title "My unusual hobby" is click bait.

zyxzevn 7 months ago [flagged] [+3]

amelius 7 months ago [-]

Reminds me of Principia Mathematica, where the author (Bertrand Russell) needs 379 pages to prove that 1+1=2. [1]

> "From this proposition it will follow, when arithmetical addition has been defined, that 1 + 1 = 2." —Volume I, 1st edition, page 379.

[1] https://en.wikipedia.org/wiki/Principia_Mathematica

mcguire 7 months ago [-]

I heartily recommend *Mathematics Made Difficult* by Carl Linderholm.

https://en.wikipedia.org/wiki/Mathematics_Made_Difficult?wpr...

Imagine an introduction to math in that style.

fizixer 7 months ago [-]

Fun aside: I'd love to see the reaction of an applied-math class, like an engineering class, in which the instructor after having introduced the course, says, "let's go through some math prequisites for this class" and essentially starts reciting something equivalent to Principia Mathematica (without mentioning what he's trying to prove, meaning in a bottom-up fashion),

- "If A is true, and B is true, then C is true"

- "We know A is true, therefore if B is true, C must be true"

- "B is true if and only if D is true and E is true"

- ...

- Therefore C is true.

- (on and on and on for a whole hour).

rocqua 7 months ago [-]

Do that to any pure math class and they will similarly walk out. You need to know where something abstract like this is going to follow at all.

Bromskloss 7 months ago [-]

> You need to know where something abstract like this is going to follow at all.

Indeed, and do remind your audience from time to time about where you're going. Otherwise, your whole line of reasoning might become useless to someone who lost track of the point, or just didn't hear or grasp it the first time you said it.

kawyut 7 months ago [-]

I've read a fair bit about functional programming but the right-associative nesting of -> still bothers me.

(first precondition IMPLIES (second precondition IMPLIES (implication))

is inherently harder for me to understand than

(first precondition AND second precondition) IMPLIES implication

even though logically they are the same. The first version carries more mental overhead because it seems to indicate that there is some sort of hierarchy/assymetry to the two preconditions.

It's the same thing when I try to read a Haskell function type signature:

f :: a -> b -> c

f is a function taking a and returning (a function taking b and returning (c))

vs.

f is a function taking two parameters a and b and returning c

There is no hierarchy anyhow because the parameters can be swapped:

f' = \b a -> f a b

Curious to know if it's just me or there are others who experience the same.

mikekchar 7 months ago [-]

I am not a particularly experienced functional programmer. What functional programming I do is mainly relegated to code I write as a hobby, but in the last year or so I've started using more functional techniques at work (and luckily my teammates haven't lynched me yet). These days at work I use primarily Ruby and I've been surprised at how often I write curried procs. The language neither requires nor encourages that behaviour, so why am I doing it?

I think I started with the same assumption that you have: the order or parameters is arbitrary. In functional programming, though, this is not the case. For example, the "map" function takes a function and a functor (you can think of a functor as a collection like a list if you aren't familiar with the term). The order is important because if I partially apply a parameter to "map" I get a completely different meaning.

If you partially apply the function to "map", then the result is a function that applies that function to any kind of functor. If you partially apply the functor to "map", then the result is a function that applies any kind of function to that specific functor. Because I can only partially apply the first parameter in the function, this helps define the meaning of the function. In this example, the first definition is more correct (because "map" always takes the function first and the functor second). This becomes increasingly important as you build applicatives and monads.

In case that was unclear, here's a quick example. Let's say I have a list of integers and I have a function that adds 5 to an integer. If I do something like "map addFive list", it will go through each element in the list and add 5, resulting in a new list. If I partially apply the function to map with "map addFive", this will return a function that loops through any list and adds five to each element. So I might say "let addFiveToEach = map addFive".

If we had a different kind of map function (let's call it "mapNew") maybe we can do "mapNew list addFive" and it will do the same as the "map" code above. But if I partially apply list to mapNew I get a function that applies a function to my specific list. I might do "let eachInList = mapNew list".

The result is two very different functions. Either I will do "addFiveToEach list" or I will do "eachInList addFive". What meaning I prefer depends on the situation.

It's this idea that partially applied functions have a meaning that is powerful. When you are writing the code, which meaning do you want? As you note, both are possible, but I think you'll find that only one is actually appropriate in most cases. For example, instead of calling the function with both parameters, what if you assigned the result of partially applying the function to a variable? What would the variable be called? Does that help the reader understand the code?

Like I said, I was surprised at how often it makes a difference in my code. In fact, where it doesn't make a difference, it's usually an indication that I've made a mistake somewhere. It is even to the point where you start to want to pass partially applied functions around because they are more meaningful than the data that they contain. In fact a function is literally just another container in a functional programming language. For me, that was a kind of mind blowing realisation.

BoppreH 7 months ago [-]

I think that's just an artifact of how currying is usually specified. Imagine a language with no automatic currying, but the special name "_" makes the function partially applied: "let addFiveToEach = map addFive _" (or "map (+ 5 _) _"). Then you can also do "let eachInList = map _ list", and you don't have to choose between them.

It does sacrifice some use cases, such as not knowing the total number of parameters, and does add more line noise. But I would be pretty happy with this tradeoff.

kawyut 7 months ago [-]

> In functional programming, though, this is not the case.

That's an artifact of the language, not inherent to functional programming.

(I had a longer response in mind, might write it up later when I get time.)

fizixer 7 months ago [-]

So I'm wondering what is the bridge between "proof-assistant" and "automated (or partially automated) theorem prover".

As someone with "journalistic" (but fairly in-depth) knowledge about these topics, my guess is that something like a proof-assistant will have to be paired up with some kind of logic programming system (like Prolog, or a kanren-derived system or something).

I guess then the problem of combinatorial explosion in logic search translates directly to the combinatorial explosion of proof-steps in a proof assistant. Hence my mention of partially-automated (more doable than fully-automated).

Probably some kind of a "database of facts" would also be involved, making it an expert system (reminds me of metamath [1]).

Is it fair to say, partially-automated theorem proving is a fairly advanced version (and not without added complexity) of a proof-assistant, and will help ease some of the grunt work involved in using a proof-assistant by hand?

[1] http://us.metamath.org/

dwheeler 7 months ago [-]

I also enjoy creating proofs as a hobby. When I'm doing it for fun, I tend to use Metamath, in particular, the "Metamath Proof Explorer" (MPE, aka set.mm) which formalizes the most common mathematical foundation (classical logic + ZFC). You can see that here: http://us.metamath.org/mpeuni/mmset.html

Here's a short video intro about MPE that I made: https://www.youtube.com/watch?v=8WH4Rd4UKGE

The thing I like about Metamath is that every step is directly visible. Coq, in contrast, shows "how to write a program to find a proof" - and not the proof itself. So you can see absolutely every step, and follow the details using simple hyperlinks.

Of course, all approaches have pros and cons. Metamath has some automation; the metamath and mmj2 programs can automatically determine some things, and there's been some experiments with machine learning that are promising. However, Coq has more automation, in part because it operates at a different level.

raphlinus 7 months ago [-]

I've done a bit of work on a "next generation" Metamath, Ghilbert[1], but have come to the conclusion that it's too ambitious to do as a part-time project. People might find it interesting, though, because it attempts to take interactivity to a higher level than Metamath, and is (imho) a cleaner language.

[1] http://ghilbert.org/

fizixer 7 months ago [-]

Hey glad to find someone involved in metamath related project.

I guess if I could simplify my discussion to one question, it would be: Does metamath or ghilbert use a logic inference engine to automate parts of a proof?

The way I understand metamath (the website, not the language) is that it's like an encyclopedia of mathematical knowledge. The encyclopedia is in the form of theorems, lemmas, corollaries, and their proof. But unlike an encyclopedia or disconnected articles, all the propositions build on top of one another. As a result, it builds (or tries to build) the whole mathematical edifice starting from core axioms (well, I guess it didn't build itself, but hundreds of mathematicians contributed to the database, like people contributing to wikipedia in the form of article-writing. As to how automated this process is, I don't know).

Now the question is, can we use this edifice as a database, to be input to a logic inference engine. What this would accomplish is that:

- If I make a query about a mathematical proposition (e.g., Are there infinite prime numbers), the logic inference engine would use the database to conclude that there are if it is able to access all the relevant propositions.

- If it's not able to reach a conclusion, e.g., if my query was "Are there infinite twin primes?" it would start a process of exploring new propositions from old propositions. Then either the search would conclude very quickly (if the proposition is only one or two steps away from the knowledge in the database) or it would give up after a few million, or billion attempts (e.g., this is more likely the case with the twin prime query). There will have to be a manually-specified cutoff point otherwise it could go into an infinite loop (I guess it may have something to do with decidability, incompleteness theorem, halting problem, etc, etc. Though even many decidable problems are computationally intractable due to very very large search space).

- Suffice to say, when I say "query" I don't mean a question in english language but a well-crafted mathematical statement in language of metamath itself.

P.S.: As for your mention of this work not doable as part-time, I fully agree with you. It would have enough cognitive load to be a very complex project even if done full time.

raphlinus 7 months ago [-]

Metamath basically goes in the other direction than most proof assistants, it relies on human ingenuity. There is a bit of search in the original C metamath (the "improve" feature) and also just a tiny amount in mmj2 (unification), but if the user doesn't have a clear picture what the fine-grained steps are, they're going to have a bad time.

One of the interesting things about Metamath is that proofs tend to be very concise. In more automated proof systems, you use automated tactics and proof search techniques, and if you expanded out their output, they'd generally be quite verbose. I think this has led a lot of people to believe that automation is more necessary than it actually is.

All that said, you absolutely could build an inference engine of similar scope as the other big proof assistants, and use the existing Metamath database as a rich foundation.

Happy to answer other questions.

auggierose 7 months ago [-]

These notions are actually becoming more and more intermixed. You could say today that interactive theorem proving (what is done in a proof assistant) is an extension of automated theorem proving, as for example the Isabelle proof assistant has very strong support for proving intermediate theorems automatically.

fizixer 7 months ago [-]

Thanks, I think this is a good example of what I'm looking for.

One key ingredient is the presence of a logic inference engine (and not for type inference only, as in Coq, Haskell).

- If your system doesn't have a logic inference engine, it's a vanilla proof-assistant (Coq).

- If it has one, it's one of two:

```
- a partially-automated theorem prover, (Isabelle)

- a fully-automated theorem prover
```

P.S.: Wait a minute. It just occurred to me, is Coq using its Hindley-Milner Type Inference engine as a Logic Inference engine for verifying constructed proofs of theorems and propositions? (is that the killer idea?)

zaptheimpaler 7 months ago [-]

Cool post! The stuff about Curry-Howard was really interesting and relates to a question i have been thinking about. But I'm a novice when it comes to the theory of type systems or theorem provers. Maybe someone here can enlighten me?

As type systems become more complex (like say in Scala or Haskell), they move towards having specialized syntax and weird hacks to express what are arguably simpler logical theorems/invariants about values in a type.. so why can't we have a programming language that gives you a full blown logic system(Coq? or Microsoft Z3? a SAT Solver?.. i don't understand the differences in detail) rather than part of one? Are there attempts at that, or does it turn out to be unworkable?

curryhoward 7 months ago [-]

> so why can't we have a programming language that gives you a full blown logic system

Most dependently typed programming languages, including Coq, give you the ability to basically "do logic" in a mostly unrestricted way.

There are a few challenges that come to mind with making a practical programming language that can also do arbitrary math:

a) For the language to be useful for proving things, it needs to be strongly normalizing. That means all programs halt—otherwise, you could trivially prove any proposition. Unfortunately that limits its utility as a general purpose programming language.

b) The languages which attempt to be useful for theorem proving are generally pure functional languages (due to the strong correspondence between functional programming and logic). Most programmers don't like working in pure functional languages, although the success of Haskell shows that there are certainly programmers who do (myself included).

c) The more expressive a type system is, the more difficult type inference becomes. Yet, paradoxically, I find myself writing more types in languages like C++ and Java than I do in equivalent Coq programs. I believe Coq uses a variant of a bidirectional type inference engine, which is quite usable in practice.

Idris is an attempt to make a practical programming language with the full power of dependent types. So it's probably the closest real language to what you're describing.

Z3 and other SAT/SMT solvers take a different approach: they try to search for proofs automatically, and for certain kinds of math they are very successful. But these are generally only for first-order theories or other restricted kinds of propositions, and not great for general theorem proving. They are good for things like proving array accesses are within bounds, or that you never try to pop from an empty stack (at least that's my understanding of their capabilities).

iamrecursion 7 months ago [-]

I should add that something like FStar [0] combines the capabilities of automated theorem proving and a more manual-proof-based dependent type system like that in Idris.

It doesn't have some of the power of Idris' elaborator reflection, for example, but it can eliminate many long-winded manual proofs via the SMT solver.

[0] https://www.fstar-lang.org/

mietek 7 months ago [-]

> For the language to be useful for proving things, it needs to be strongly normalizing. That means all programs halt—otherwise, you could trivially prove any proposition. Unfortunately that limits its utility as a general purpose programming language.

It's not true that a strongly normalising language must be limited in utility. See section 4 on codata of Turner's 2004 "Total functional programmming" for more information.

https://github.com/mietek/total-functional-programming/blob/...

curryhoward 7 months ago [-]

> I believe Coq uses a variant of a bidirectional type inference engine, which is quite usable in practice.

Edit (too late to edit the original comment): After thinking about the kinds of type errors I've seen in my proofs, I think Coq's inference algorithm is based on unification (not bidirectional?). Whatever it is, it's quite nice to work with in any case.

Kutta 7 months ago [-]

> The more expressive a type system is, the more difficult type inference becomes.

This is always mentioned and I always fail to see the relevance. Inferable terms stay inferable when we add dependent types.

rocqua 7 months ago [-]

The basic issue lies with `decidability'. Generally we want type systems to tell us whether our program is well typed. However as your type-system becomes `too expressive' it starts to become much harder to determine if a program is well typed.

When finally a type system becomes so expressive as to be Turing complete, you get 'undecidable types'. Those are the types that correspond to undecidable programs for Turing machines.

A consequence is that any `sound' typing system will have to reject some programs that actually are `well typed'. This is why many type systems have a "Just trust me, this is fine" option. Like casting in C/C++ and unsafe in rust.

To fully eliminate those "Just trust me" and still accept all well typed programs you'd need to accept potentially never ending type-checking (or else you'd have solved the halting problem).

__s 7 months ago [-]

Various difficulties arise in how to have the expressive power play well with type inference

[0]: http://smallcultfollowing.com/babysteps/blog/2016/11/04/asso...

pron 7 months ago [-]

We do have such languages. For example, Java has JML [1], C has the very similar ACSL, and a programming language like Idris has a type system that can express arbitrary logical propositions (there are various tradeoffs that may influence the choice between a rich type system a-la Idris or a rich contract system a-la JML, but many of the principles are the same).

The problem is that we do not yet know how to make effective use of such code-level specifications. Finding a cost/benefit sweet spot is not easy (I personally prefer the JML approach precisely because I think it allows for greater flexibility in the choice of tradeoffs vis-a-vis confidence vs. cost).

[1]: http://www.openjml.org/

mcguire 7 months ago [-]

Don't forget Ada/SPARK and Dafney. The former probably has the most actual experience and documentation and the latter is pretty well integrated.

ivan_ah 7 months ago [-]

Here is a paper that shows Coq proofs of two famous theorems by Shannon: the source compression rate of H(X) and the channel capacity of I(X,Y).
https://staff.aist.go.jp/reynald.affeldt/documents/affeldt-i...

I wonder if simplified versions of these proofs couldn't be used with students. Right now it seems like too much details, but maybe there can be a common "stdlib" of standard math arguments that we assume to be true, and then focus on the specific machinery for each proof?

j2kun 7 months ago [-]

My question for the author: does implementing a proof formally in Coq honestly give you a better understanding of the theorem/proof?

It would seem to me that, if you're filling in details from an existing proof or making some existing formal (in the math sense) intuition more formal (in the Coq sense), there's not much deep insight to gain from implementing a proof in Coq.

curryhoward 7 months ago [-]

My experience is: formally proving something in Coq is not a prerequisite for understanding it, but understanding it is a prerequisite for formally proving it in Coq. So if you get Coq to accept your proof, it's a good sign that you understand every detail.

For the domain theory proof I discussed in the article, formalizing it in Coq absolutely helped me understand it because:

a) Merely formalizing the definitions was valuable for me, because there are tricky subtleties to them (e.g., the supremum of a subset doesn't need to be in the subset, the various ways to define a "complete partial order", etc.).

b) When you do a formal proof, you have to refer to the definitions quite often. This just reinforces them in your brain.

c) Of course, when you formalize things in Coq, any misunderstandings you had about the material will be brought to your attention when your proof doesn't work.

On the other hand, I didn't gain much insight from the soundness proof for the simply typed lambda calculus. The definitions for that development were more straightforward. But I was already fairly familiar with the ideas of progress, preservation, etc. so I didn't expect to learn much from doing that. If anything, the main thing I gained from that was experience using Coq.

bordercases 7 months ago [-]

> My experience is: formally proving something in Coq is not a prerequisite for understanding it, but understanding it is a prerequisite for formally proving it in Coq. So if you get Coq to accept your proof, it's a good sign that you understand every detail.

Such tasks are called "orienting tasks" in the cognitive science/human learning literature, and are useful because what it means to "understand" something is quite nebulous and we often fool ourselves when we actually think we understand something. They also allow us to test understanding outside of the domain it's supposed to be applied in, which is useful if the real situation is noisy, messy or costly.

Scott H Young explains the psychology of the problem well: https://www.scotthyoung.com/blog/2017/06/13/how-much-do-you-...

sundarurfriend 7 months ago [-]

Thanks for the link, that was quite interesting.

I love the format that the article takes, as well. I liked the superscript note numbers that xkcd's whatif blog uses, this is another implementation of that, executed very well.

bordercases 6 months ago [-]

It makes printing a bitch but overall I wouldn't mind if more articles were formatted like that, yeah.

adrusi 7 months ago [-]

I think reading about a theorem and a it's proof and then implementing it in a formal language probably similar to watching a professor prove it on the board, then going home and proving similar things on the homework. Writing it in a formal language forces you to think about it at a level of detail that's easy to skip if you just read it.

losvedir 7 months ago [-]

Whoa, very cool post! It's interesting that "small but novel" languages can be formally proved. Do any mainstream languages have a sound foundation like that? If someone is interested in creating a new programming language, is it worth getting up to speed on proof writing and using that to build the language? It's not really a topic that comes up in the Dragon Book or mainstream general compiler study, AFAIK.

curryhoward 7 months ago [-]

> Do any mainstream languages have a sound foundation like that?

I can think of a few languages that started from a formal theory (e.g., Eff [1]), but none of them are "mainstream". There was a mostly successful attempt [2] at formally specifying C and implementing a verified compiler for it. Standard ML has a very comprehensive spec [3], perhaps the best of any "mainstream" language. It's not quite at the level of a Coq formalization, but also not too removed from it. Haskell is based on a lot of peer-reviewed papers, but there isn't one comprehensive spec for it at this level of formalism. Also, the RustBelt project [4] is worth mentioning. It's an initiative to develop a formal theory of Rust.

> If someone is interested in creating a new programming language, is it worth getting up to speed on proof writing and using that to build the language?

If your new programming language contains a novel idea that you want to have published in a peer-reviewed journal/conference, then yes, I think it is absolutely worth it. It's likely your paper will be rejected without having formal proofs.

But if you don't intend to publish your ideas, it's really quite a lot of work and might not be worth it. Depends on how dedicated you are, and how much you care about things like soundness.

[1] http://www.eff-lang.org/

[2] http://compcert.inria.fr/

[3] http://sml-family.org/sml97-defn.pdf

[4] http://plv.mpi-sws.org/rustbelt/

pron 7 months ago [-]

Terrific!

I took the liberty of translating your module to TLA+, my formal tool of choice (recently, I've been learning Lean, which is similar to Coq, but I find it much harder than TLA+). I tried to stick to your naming and style (which deviate somewhat from TLA+'s idiomatic styling), and, as you can see, the result is extremely similar, but I guess that when I try writing the proofs, some changes to the definitions would need to be made.

One major difference is that proofs in TLA+ are completely declarative. You just list the target and the required axioms, lemmas and definitions that require expanding. Usually, the proof requires breaking the theorem apart to multiple intermediary steps, but in the case of the proof you listed, TLA+ is able to find the proof completely automatically:

```
LEMMA supremumUniqueness ≜
  ∀ P ∈ SUBSET T : ∀ x1, x2 ∈ P : supremum(P, x1) ∧ supremum(P, x2) ⇒ x1 = x2
PROOF BY antisym DEF supremum (* we tell the prover to use the antisym axiom and expand the definition of supremum *)
```

The natDiff lemma doesn't even need to be stated, as it's automatically deduced from the built-in axioms/theorems:

```
LEMMA natDiff ≜ ∀ n1, n2 ∈ Nat : ∃ n3 ∈ Nat : n1 = n2 + n3 ∨ n2 = n1 + n3
PROOF OBVIOUS (* this is automatically verified using just built-in axioms/theorems *)
```

Another difference is that TLA+ is untyped (which makes the notation more similar to ordinary math), but, as you can see, this doesn't make much of a difference. The only things that are different from ordinary math notation is that function application uses square brackets (parentheses are used for operator substitution; operators are different from functions, but that's a subtelty; you can think of operators as polymorphic functions or as macros), set comprehension uses a colon instead of a vertical bar, a colon is also used in lieu of parentheses after quantifiers, and aligned lists of connectives (conjunctions and disjunctions) are read as if there were parentheses surrounding each aligned clause. Also `SUBSET T` means the powerset of T.

Here's the module (without proofs, except for the one above):

```
------------------------------ MODULE Kleene ------------------------------
EXTENDS Naturals

(*
  Assumption: Let (T, leq) be a partially ordered set, or poset. A poset is
  a set with a binary relation which is reflexive, transitive, and
  antisymmetric.
*)

CONSTANT T
CONSTANT _ ≼ _

AXIOM refl    ≜ ∀ x ∈ T : x ≼ x
AXIOM trans   ≜ ∀ x, y, z ∈ T : x ≼ y ∧ y ≼ z ⇒ x ≼ z
AXIOM antisym ≜ ∀ x, y ∈ T : x ≼ y ∧ y ≼ x ⇒ x = y

(*
  A supremum of a subset of T is a least element of T which is greater than
  or equal to every element in the subset. This is also called a join or least
  upper bound.
*)

supremum(P, x1) ≜ ∧ x1 ∈ P
                  ∧ ∀ x2 ∈ P : x2 ≼ x1
                  ∧ ∀ x3 ∈ P : ∀ x2 ∈ P : x2 ≼ x3 ⇒ x1 ≼ x3

(*
  A directed subset of T is a non-empty subset of T such that any two elements
  in the subset have an upper bound in the subset.
*)

directed(P) ≜ ∧ P ≠ {}
              ∧ ∀ x1, x2 ∈ P : ∃ x3 ∈ P : x1 ≼ x3 ∧ x2 ≼ x3

(*
  Assumption: Let the partial order be directed-complete. That means every
  directed subset has a supremum.
*)

AXIOM directedComplete ≜ ∀ P ∈ SUBSET T : directed(P) ⇒ ∃ x : supremum(P, x)

(*
  Assumption: Let T have a least element called bottom. This makes our partial
  order a pointed directed-complete partial order.
*)

CONSTANT bottom

AXIOM bottomLeast ≜ bottom ∈ T ∧ ∀ x ∈ T : bottom ≼ x

(*
  A monotone function is one which preserves order. We only need to consider
  functions for which the domain and codomain are identical and have the same
  order relation, but this need not be the case for monotone functions in
  general.
*)

monotone(f) ≜ ∀ x1, x2 ∈ DOMAIN f : x1 ≼ x2 ⇒ f[x1] ≼ f[x2]

(*
  A function is Scott-continuous if it preserves suprema of directed subsets.
  We only need to consider functions for which the domain and codomain are
  identical and have the same order relation, but this need not be the case for
```

```
  continuous functions in general.
*)

Range(f) ≜ { f[x] : x ∈ DOMAIN f }

continuous(f) ≜
  ∀ P ∈ SUBSET T: ∀ x1 ∈ P :
    directed(P) ∧ supremum(P, x1) ⇒ supremum(Range(f), f[x1])

(* This function performs iterated application of a function to bottom. *)

RECURSIVE approx(_, _)
approx(f, n) ≜ IF n = 0 THEN bottom ELSE f[approx(f, n-1)]

(* We will need this simple lemma about pairs of natural numbers. *)

LEMMA natDiff ≜ ∀ n1, n2 ∈ Nat : ∃ n3 ∈ Nat : n1 = n2 + n3 ∨ n2 = n1 + n3

(* The supremum of a subset of T, if it exists, is unique. *)

LEMMA supremumUniqueness ≜
  ∀ P ∈ SUBSET T : ∀ x1, x2 ∈ P : supremum(P, x1) ∧ supremum(P, x2) ⇒ x1 = x2
PROOF BY antisym DEF supremum

(* Scott-continuity implies monotonicity. *)

LEMMA continuousImpliesMonotone ≜ ∀ f : continuous(f) ⇒ monotone(f)

(*
  Iterated applications of a monotone function f to bottom form an ω-chain,
  which means they are a totally ordered subset of T. This ω-chain is called
  the ascending Kleene chain of f.
*)

LEMMA omegaChain ≜
  ∀ f : ∀ n, m ∈ Nat :
    monotone(f) ⇒
        approx(f, n) ≼ approx(f, m) ∨ approx(f, m) ≼ approx(f, n)


(* The ascending Kleene chain of f is directed. *)

LEMMA kleeneChainDirected ≜
  ∀ f : monotone(f) ⇒ directed({ approx(f, n) : n ∈ Nat })

(*
  The Kleene fixed-point theorem states that the least fixed-point of a Scott-
  continuous function over a pointed directed-complete partial order exists and
  is the supremum of the ascending Kleene chain.
*)

THEOREM kleene ≜
  ∀ f : continuous(f) ⇒
    ∃ x1 ∈ T :
        ∧ supremum({ approx(f, n) : n ∈ Nat }, x1)
        ∧ f[x1] = x1
        ∧ ∀ x2 : f[x2] = x2 ⇒ x1 ≼ x2

============================================================================
```

curryhoward 7 months ago [-]

Wow, this is amazing. Can you share a link to the full module with proofs? I'd love to compare it to the Coq version.

> pron 7 months ago [-]
>
> Sure — if and when I get around to writing them :) Looking at your proofs will probably make this a lot easier. After all, the principles of proof are very similar in pretty much every system. But if you want to get a sense of what a methematical proof in TLA+ looks like (as opposed to algorithm proofs, which make up most TLA+ examples), see the appendix to this paper: https://lamport.azurewebsites.net/pubs/proof.pdf
>
> But it's very important to know that TLA+ has very different usage goals from Coq/Lean/Isabelle, and so makes very different design tradeoffs. The tools are certainly *not* interchangeable.
>
> Coq, Lean and Isabelle are first-and-foremost research tools. As such, they are completely general purpose, very powerful, but also very, very complicated. TLA+, on the other hand, was designed for use by ordinary engineers to specify and verify large, real-world digital (software or hardware) systems. This means that TLA+ is not generally appropriate for doing high math (e.g., because it's untyped, it doesn't even have operator overloading, which alone would make it unappealing for serious math; then again, not much serious math is done formally, anyway), and isn't designed for logicians to explore various logics (it's classical logic only). In addition, it's syntactic/symbol manipulation capabilities are very limited, so while you *could* specify a programming language, as you've done, the result is going to be much messier (although specifying low-level languages with simple syntax, like machine- or byte code, can be very elegant).
>
> OTOH, TLA+ is truly exceptional in its ease of learning and its capabilities when it comes to specifying and verifying systems and algorithms. In many respects, it's computational theory is richer than Coq's (although Coq is so general that some people have embedded TLA, the computational theory of TLA+ in Coq; but TLA+ has it out of the box). In addition, when used in industry, people hardly ever use TLA+'s proof system. Not because it's inadequate for proving the correctness of algorithms — it truly excels at that — but because formal proofs are, in general and regardless of the choice of tool, not very cost effective. Instead, TLA+ has a model checker that lets you verify your propositions about your system completely automatically — you just state them and press a button — albeit, only on small, finite instances. This is more than sufficient for industry use, as, while there have been multiple cases of informal (published and peer-reviewed) correctness proofs later found to be wrong with a model-checker, I am not aware of any cases where a model checker result was found wrong, despite running on small instances, and so not qualifying as proof.
>
> So the choice between TLA+ and Coq/Isabelle/Lean is usually simple. If you're an academic researching new logics, new programming languages, or complex formal mathematics, you'll choose Coq/Isabelle/Lean. If you're an engineer trying to design a complex or subtle system or algorithm, you'll likely choose TLA+.

fizixer 7 months ago [-]

If what you say is true, then this is one more reason for me to learn TLA+.

As I asked in a different question, is TLA+ able to do the proofs declaratively and automatically because it has an internal 'logic inference' engine?

pron 7 months ago [-]

Well, TLA+'s proof system, called TLAPS [1], is not an independent proof assistant, but rather a frontend, which uses the proof-assistant Isabelle, combined with various SMT and tableau solvers for proof-search (like Z3, Yices, Zenon) under the cover.

Isabelle and Lean also make use of automatic provers for proof automation.

BTW, TLA+-style declarative proofs are not always better than Coq-style imperative proofs. They are certainly easier to read (the proof language was designed to be easily readable by humans), but they're sometimes harder to write, because it's largely an iterative trial-and-error process: you first try to prove the entire proposition automatically. If it fails, you look at the error message, try to see the difficulty, and then break the proof down into smaller steps, and so on.

[1]: https://tla.msr-inria.inria.fr/tlaps/

fizixer 7 months ago [-]

> BTW, TLA+-style declarative proofs are not always better than Coq-style imperative proofs. They are certainly easier to read (the proof language was designed to be easily readable by humans), but they're sometimes harder to write, because it's largely an iterative trial-and-error process: you first try to prove the entire proposition automatically. If it fails, you look at the error message, try to see the difficulty, and then break the proof down into smaller steps, and so on.

I'd say it still appears to be better than fully-impertive "manual" process. (though I have used neither Coq nor TLA+).

- In Coq, you have to construct the whole proof yourself. Coq system merely verifies what you constructed.

- In TLA+, you need to do 1 or more iterations, but you have the full-weight of the computation behind you, sharing your cognitive load. Your only cognitive load is to think about the critical parts of the proof, if any, and the system will do the rest.

I wonder if the TLA+ system also uses a "database of propositions" behind the scenes to help with faster inference, e.g., something equivalent to Russell and Whitehead's Principia Mathematica (but in a coded form), or like metamath.

pron 7 months ago [-]

> I'd say it still appears to be better than fully-impertive "manual" process.

I really don't know. It may largely depend on what you want to do, and I believe Coq also has powerful proof automation, albeit one that works differently. Both take quite a bit of practice to write. The main thing I can say is that TLA+ proofs are easy to read. In any event, formal proofs in any style are far from a walk in the park (which is why when using TLA+ in practice we try to avoid them altogether and just use the model-checker).

Also, TLA+ is a much gentler introduction to formal methods, as it's so, so, so much easier to learn, and the availability of a model checker means that, when learning, you can concentrate on writing specifications, a much more important skill than writing proofs (if you're interested in software, that is).

> I wonder if the TLA+ system also uses a "database of propositions" behind the scenes to help with faster inference, e.g., something equivalent to Russell and Whitehead's Principia Mathematica (but in a coded form), or like metamath.

It automatically uses the axioms of the core logic (essentially ZFC set theory with a Hilbert's choice operator, some simple theorems on the built-in sets, like the natural numbers [0] plus TLA, which is a simple yet very powerful temporal logic for computational reasoning). Other than that, it is a core principle of the system that you manually list which theorems are to be used for each step of the proof. It doesn't only help the provers -- it also helps the human reader. But, to save you typing, you can write "USE lemma1, lemma2", and have them automatically added to all following steps.

The proof language (like the entire specification language) was designed mostly by Leslie Lamport, based on his "structured proof" style detailed here [1]. The appendix of the paper lists some formal calculus proofs, checked with TLAPS, if you want to get a sense of what those proofs look like.

[0]: For example, here's the proof for the natDiff lemma from the post:

```
LEMMA natDiff ≜ ∀ n1, n2 ∈ Nat : ∃ n3 ∈ Nat : n1 = n2 + n3 ∨ n2 = n1 + n3
PROOF OBVIOUS (* this is automatically verified using just built-in axioms/theorems *)
```

So, actually, it doesn't even need to be stated.

[1]: https://lamport.azurewebsites.net/pubs/proof.pdf

mcguire 7 months ago [-]

With verification systems like Frama-C and SPARK, which have similar difficulties, I like to think of it as playing "what do I know at this point" until the proof goes through.

carapace 7 months ago [-]

I've been working in a CPS Joy variant, deriving recursive functions on trees and such. I've noticed that the code always works the first time. So far my "data" on this phenomenon is still too anecdotal to get excited, but I wanted to mention it (here where people might be interested.) It surprised me at first, but then I realized that this is part of the promise of Backus' Functional Programming: being able to do algebra to derive programs gives you error-free code that doesn't require tests.

It leads me to speculate... what if I don't need types to write correct software? I assumed I would implement typing and type inference (like Factor language already has) at some point. But if the code has no bugs without it... YAGNI?

My reasoning goes like this: Let's postulate that we have a library of functions, each of which has the quality Q of being error-free. If we derive new functions from the existing ones by a process that preserves Q, so that new functions are all also error-free, then we cannot have errors in our code, *just user error*. I.e. the user can present bad inputs, but no other source of error exists (other than mechanical faults.) This would be a desirable state of affairs. Now if the Q-preserving new-function-deriving process doesn't require types to work, then we don't need them. Whether or not typing helps prevent user error is another question. Personally, I lean toward defining user error as UI error, but I am ranging far from my original point now.

(If you're interested: https://github.com/calroc/joypy/tree/master/docs Still very DRAFT versions. https://github.com/calroc/joypy/blob/master/docs/3.%20Develo...)

pagnol 7 months ago [-]

I'm currently working through the material here: https://github.com/pigworker/CS410-17

yodsanklai 7 months ago [-]

I've never managed to have fun with Coq even though I really tried. I did tons of exercises and labs. On the other hand, I do love (functional) programming, logic and maths in general.

It's almost like you get the bad side of programming (time-consuming, tedious) without the reward (seeing your program doing cool things). And you don't get the satisfaction of doing maths either. At least I don't. Maybe I didn't try hard enough.

What is very interesting though is to learn about type theory and the Curry Howard isomorphism.

phkahler 7 months ago [-]

Is it possible for one of these "programs" to run forever? If so, does that illustrate a direct correspondence between the halting problem and godels incompleteness theorem?

curryhoward 7 months ago [-]

There are two languages to discuss here:

1) The underlying pure functional language that forms the logical basis for Coq (called "Gallina"). It is not possible to write a program which runs forever in this language, because all recursive calls must be "well founded". Coq uses a very simple set of rules to determine when recursive calls are guaranteed to terminate (slightly more flexible than primitive recursion), but it rules out some legitimate programs which do not terminate. For example, writing merge sort is difficult in Coq because the type checker can't automatically prove that the recursion terminates. For these cases, you can provide your own ordering relation and prove that it is well-founded. So tldr, you have to prove that your programs terminate (and this is automated in many cases). The fact that all Gallina programs terminate is crucial; otherwise, you could trivially prove any proposition.

2) Usually in Coq you do not write most proof terms in Gallina manually (because it is quite unnatural). Instead, you write scripts called "tactics" to generate the terms for you. These scripts are written in a dynamically typed language called Ltac, and programs in this language can run forever. But all proofs generated by these tactics are type checked in Gallina, so it's impossible for a tactic to generate a bad proof term without it being detected by Gallina's type checker.

curryhoward 7 months ago [-]

* but it rules out some legitimate programs which terminate.

(Typo, but too late to edit the original comment.)

maxhallinan 6 months ago [-]

A lot of these comments are focused on the merits of Coq and some of the mathematical details of your post. As someone who has no experience with Coq and little experience with mathematical proofs, I want to add an additional perspective. This is a pretty cool hobby. It's really nice to hear about hobbies that aren't motivated by ambitions of profit or fame. Thanks for sharing!

zitterbewegung 7 months ago [-]

Anyone else here do recreational mathematics ? I mainly try to answer questions on CSTheory.stackexchange.com

tluyben2 7 months ago [-]

I have the same hobby; I play with Coq, Idris, Agda, F* and pen and paper. I do not have time to do it for work most of the code, however all critical parts I do. One of my dreams is to do this full time.

lordleft 7 months ago [-]

Are there mathematical proofs that are incapable of being expressed in Coq? Isn't there a limit to kind of ideas a computer could verify? Curious about the boundaries of this language.

yaseer 7 months ago [-]

This question gets at the heart of computation, logic and provability.

Coq is based on a logic system called the "Calculus of constructions" https://en.wikipedia.org/wiki/Calculus_of_constructions. As such it has the same limitations as that logical system.

Most notably, Gödel's incompleteness theorems place a limit on what's provable in Coq, or indeed many other systems.

There are various kinds of logic systems, each with different kinds of 'power'. Power is a double-edge sword in logic systems, as with power comes the capacity to introduce paradoxes and inconsistencies.

As with programming languages, certain logic systems and automated logics are suited for particular cases. For example, there is a system 'MetiTarski' like a Domain Specific Language, designed to prove theorems about real-valued special functions.(https://www.cl.cam.ac.uk/~lp15/papers/Arith/).

curryhoward 7 months ago [-]

Practically all of mathematics can be formalized in Coq. One notable exception is the internal consistency of Coq itself (due to Gödel's first incompleteness theorem, I believe).

Coq's logic is intuitionistic, which means it's impossible to prove the following: forall P, P or ~P. But this actually makes Coq more powerful than classical logic, not less, because:

a) If you want, you can add that proposition (called the law of the excluded middle) as an axiom. Now you have classical logic.

b) By not adding that axiom, you can show that certain results do not depend on it. You couldn't do this if it were built-in.

pbhjpbhj 7 months ago [-]

>"Are there mathematical proofs that are incapable of being expressed in Coq?" //

Like the proof that there are proofs incapable of being expressed in Coq?!

Bromskloss 7 months ago [-]

> The CI system runs Coq on my code and prevents me from merging it into the master branch if there is a mistake.

How does one make Travis CI prevent a merge?

curryhoward 7 months ago [-]

By making the master branch protected in GitHub and making the CI check for Travis required. Docs: https://help.github.com/articles/enabling-required-status-ch...

solotronics 7 months ago [-]

thinking about this same topic there is probably amazing opportunities out there for programmer mathematicians. Don't get me wrong, of course there are quants and such for trading firms but intuitively I would think there are an amazing number of applications for this kind of advanced mathematical programming that seem relatively under utilized.

libeclipse 7 months ago [-]

This is honestly beautiful.

dclowd9901 7 months ago [-]

This is how I imagine Flowtype works for Javascript (more or less). What's interesting to me, is that this could be the foundation for AIs taking over programming from developers.

sillysaurus3 7 months ago [-]

How do you pronounce Coq? I searched it but all I found was https://m.youtube.com/watch?v=TAHH83n2dmY

jloughry 7 months ago [-]

It's pronounced like "cock". It's named after the animal, and that's how you say it in French. It's how people who use the tool say it.

sls 7 months ago [-]

Not really, the English word "cock" has a very different vowel than the French "coq" - https://youtu.be/bIraCzL3dtI?t=4s is a pretty fair exemplar. It's a lot more like the "o" in "joke".

(edit: maybe you're English? Some English accents of England would articulate "cock" very close to French "coq" and "joke" would be far away. Apologies, one really does need to specify which "English" is meant, and I should have seen that at once. First cup of coffee still in progress.)

pbhjpbhj 7 months ago [-]

Quick look on YouTube, this guy says "cock" [short staccato 'k' at the end] https://www.youtube.com/watch?v=WFen8x66ETY and this guy https://www.youtube.com/watch?v=sW4uki0Fr8I - both apparently French (though the guy on the second one really sounds like he's faking it!).

In your example the 'o' is more precise and the 'k' is swallowed a little.

I'd expect they're different accents, perhaps your example is more Northern French (I really don't know).

mcguire 7 months ago [-]

Perhaps more like "Coke"?

dwheeler 7 months ago [-]

I pronounce it as "coke", for fairly obvious reasons :-).

guizzmo 7 months ago [-]

So there is this rumor hanging around the french community that use Coq, that the name was deliberately chosen to sound like "cock" in english. This way, french researcher could brag to other collegues about going to a conference about "cock", in all seriousness.

ThePawnBreak 7 months ago [-]

Something like this: https://www.youtube.com/watch?v=XKUE62hOvHM