pron on Nov 30, 2015 | parent | favorite | on: The C standard formalized in Coq

I would strongly encourage anyone interested in formal methods (and formal proofs in particular) but may be intimidated by dependent types, which serve as the basis for Coq (and are foreign not only to software developers but to most mathematicians as well) to give TLA+[1] a look. It is a decade younger than Coq (16 vs. 26), but simpler to use, much more gradual, and has had industry adoption pretty much since its inception.

Instead of dependent types (in fact, TLA+ is untyped[2]) It is based on simple ZF set theory and logic, and should be immediately familiar to anyone who's taken even one logic (or discrete math) course in college. In addition to a proof assistant it has a good model checker, comes bundled in a nice (relatively speaking) IDE, and is easy to learn (with a thorough tutorial written by Leslie Lamport himself).

TLA+ is also more widely used in the industry -- both hardware and software -- perhaps most notably in the software industry by Amazon, who use TLA+ to specify many of their AWS services. TLA+ is a formal specification (and verification) tool designed for engineers and used by engineers. Although, to be fair, neither tool is used by a large portion of the industry.

EDIT: Removed potentially incendiary Lamport quote.

EDIT: Changed an incorrect statement about Coq's use in the industry

[1]: http://research.microsoft.com/en-us/um/people/lamport/tla/tl...

[2]: See *Should Your Specification Language Be Typed?* by Lamport and Paulson: http://research.microsoft.com/en-us/um/people/lamport/pubs/l...

nickpsecurity on Nov 30, 2015 [-]

Your TLA+ comment isn't relevant to this discussion and seems to only promote TLA+. The topic is verification of compilers or imperative programs in C language. I don't know of anyone using TLA+ for that or advocating for it. It's mainly used as a lightweight approach in protocol verification. So, distributed apps not C code or compilers.

Now, Coq has *plenty* of history in certified programming and compilation. There exists formalisms and models of most pieces of the puzzle. There are worked examples of certified compilers whose details and sometimes source are available for re-use. There's even already a verified C compiler other works are building on. It was used in EAL7-class JavaCard interpreters, database structures I believe, web services, and so on. Even outputs to robust Ocaml code. People needing to learn this stuff even have free books like Chlipala's:

http://adam.chlipala.net/cpdt/

So, anyone trying to specify or formally verify compilers or algorithms have plenty of reason to choose Coq. It's actually one of the best tools for the job with HOL or Isabelle the other style. Field kind of diverges doing parallel work with them in imperative verification. Coq is possibly easier to use and seems to have more prior work to draw on in this. So, it's either the best or one of best tools for the job.

TLA+ is for totally different jobs. Not relevant here.

pron on Nov 30, 2015 [-]

As I said in another comment, stories like this make people interested in formal methods, and Coq, IMO, would be daunting for most beginners. TLA+ is a good, beginner-friendly entry point to formal methods (and you can slowly progress from specification to model-checking and only then proofs), even if you later choose to move to dependent-type approaches and tools like Coq or F*.

> TLA+ is for totally different jobs. Not relevant here.

I beg to differ. TLA+ is commonly (as far as the word "commonly" can be used when formal methods are discussed) used to specify and verify programs in C or any other language.

It is true that Coq has been used to verify compilers more than TLA+, while TLA+ has been used to verify algorithms (esp. concurrent/distributed) more than Coq, but the reason is not because neither could do the other's job, but because of their respective progeny. Coq (as all type-theory tools) is closely associated with PL-theory people, who tend to be more attracted to compilers, while TLA+ originated from the software-verification world (mostly Pnueli's temporal logic), and created and promoted by Leslie Lamport, most certainly an "algorithms" (specifically distributed/concurrent) guy rather than a "PL" guy.

BTW, Isabelle is one of the proof engines used by the TLA+ proof system.

nickpsecurity on Nov 30, 2015 [-]

" stories like this make people interested in formal methods, and Coq, IMO, would be daunting for most beginners. TLA+ is a good, beginner-friendly entry point to formal methods (and you can slowly progress from specification to model-checking and only then proofs), even if you later choose to move to dependent-type approaches and tools like Coq or F*."

TLA+ is an easier intro to verification. However, the verification of C programs to any strong degree will require tools like Coq, possibly esoterica like separation logic, tough specs, tough proofs, and so on. There's no easy route for it for beginners except to straight up learn the necessary knowledge with Chlipala's book, Software Foundations, or something similar. Then plenty of practice and study of how successful work was done.

There's a reason we celebrate every time someone comes close to specifying or verifying a significant chunk of C language. It's because it's Just That Hard (TM). ;)

"I beg to differ. TLA+ is commonly (as far as the word "commonly" can be used when formal methods are discussed) used to specify and verify programs in C or any other language."

Hmm. Interesting. I should probably do another survey of TLA+ to see what uses it's had in sequential, low-level algorithms like the OP post is on. May be more going on than I was previously aware. All the top results are using Coq and Isabelle/HOL, though.

pron on Nov 30, 2015 [-]

> There's no easy route for it for beginners

And if this post had appeared on a Coq forum, a verification forum or even LtU, I would have said nothing. However, as this is HN, and there are probably many beginners reading this, I thought it very relevant to point out a more beginner-friendly tool as a foray into this interesting subject.

> what uses it's had in sequential, low-level algorithms

This is a fairly recent paper that describes a tool that translates C to TLA+ for verification: http://www.lsv.ens-cachan.fr/Publis/PAPERS/PDF/MLBHB-ftscs15... It works for concurrent C programs as well as sequential ones.

This is an old (old) spec by Lamport of the Win32 Threads API: http://research.microsoft.com/en-us/um/people/lamport/tla/th...

And again, the choice of TLA+/Coq often has much more to do with the community the author belongs to (PL/conc-dist algorithms) rather than some specific strengths of the tool.

nickpsecurity on Nov 30, 2015 [-]

Appreciate the paper as I didn't have that interesting work. Once again, though, they're focused on the details of concurrent (eg protocol-like) work instead of verifying concrete, sequential programs. They claim in the paper to be able to prove absence of runtime errors but don't give much detail. They also aim to do more on that leveraging Frama-C but again no detail. So, your example shows it a work in progress for TLA+ rather than work done several times over as in Coq or Isabelle work.

It does support the use of TLA+ for some aspects of C verification or specification along with integrating into the right tool (Frama-C). I'll probably do the re-survey on TLA+ use I mentioned earlier due to this paper.

OopsCriticality on Nov 30, 2015 [-]

I've been wondering for a while, and since this thread seems to have attracted people who may have answers…

Can you (or anyone) provide any insight as to why ACL2 doesn't get mentioned much in discussions of formal methods? I'm interested in formal methods and ACL2 seems like a nice tool, but at least on HN all I usually see is Coq this and Coq that, with occasional mentions of TLA+ and Isabelle.

nickpsecurity on Nov 30, 2015 [-]

I'm not sure why it doesn't get mentioned a lot. I do know that both LISP and best use of ACL2 (i.e. hardware verification) both get little mention in general. A variant of one to do the other should be similar.

The best work I've seen in ACL2 comes from Rockwell-Collins and their SHADE verification suite. They've verified raw hardware, microcode, processor ISA, security policies, crypto systems... you name it. They give plenty of detail in their papers in how they do that albeit some are paywalled. The resulting AAMP7G processor is used in commercial applications needing reliability and security.

I haven't done a straight survey on ACL2 in a while, though. Might be worth a reader doing and posting in case something interesting was overlooked. Rockwell's stuff seems to be best use of it, though.

OopsCriticality on Dec 2, 2015 [-]

Not sure if you'll see this, but I did run across a preprint with a few Rockwell authors that might be of interest, "Development of a Translator from LLVM to ACL2" (http://arxiv.org/abs/1406.1566)

nickpsecurity on Dec 2, 2015 [-]

That's interesting work. Hardin is the name that was on the other ones. No surprise they dragged LLVM intermediate form into ACL2 to leverage their prior work. Good that it's fast but I'm more interested in what analysis they can do and how easily. Wish it had more of that.

Still added it to my collection as it might come in handy for future ACL2 work. :)

throwaway999888 on Nov 30, 2015 [-]

> Your TLA+ comment isn't relevant to this discussion and seems to only promote TLA+.

pron has this habit of being negative and dismissive towards functional programming and related things, recommending non-FP alternatives instead. Which is fine, only that sometimes he's too trigger happy and just gives unsolicited advice without any build up or with weak topical relevance. Like here.

pron on Nov 30, 2015 [-]

I am not dismissive towards Coq (how can anyone be?), and my advice is neither less solicited nor less relevant than many other HN comments.

But you are right that I sometimes try to correct what I see as a (strange) imbalance here on HN (which reflects neither the prevailing opinions in academia nor in the industry) regarding PFP and type theory. People unfamiliar with the topic might get the impression that PFP+type-theory is either the only or the most prevalent technique for software verification (or software correctness in general), while neither is the case. In exchange, I get to be the lucky target of various ad hominem attacks, so I guess I've earned my right to speak up :)

throwaway999888 on Nov 30, 2015 [-]

The phrase "ad hominem" has become really watered-out.

nickpsecurity on Nov 30, 2015 [-]

I noticed along with no supporting evidence presented in relevancy for this topic. Hence my comment. ;)

nmrm2 on Nov 30, 2015 [-]

A note for the unitiated: There is a tension between TLA+ and Coq et al. (typified by statements such as "weird computer-science math") that is similar and maybe even rooted in to the sorts of culture wars that sprout up around programming languages (see [2] of parent's post for a perfect example of what I'm talking about...)

The characterization in pron's post is decidedly biased toward TLA+ in a way that I think is unnecessary. A few correctives:

* Coq has seen plenty of industry adoption. To say that type-theoretic theorem provers (or the Coq system specifically) are merely academic toys compared to TLA+ is wrong.

* "Normal" mathematicians (not just "weird Computer Science-type" mathematicians, whatever that means...) have used and extolled the virtues of Coq as a tool as well as its theoretical foundations. See e.g., the univalent foundations project. Regardless, I'm not sure I understand why "Computer Science-type math" is an unreasonable foundations for a *program specification tool*, which is what TLA+ is. And if the goal is formalized mathematics, you're probably better off with Coq (and it's not even clear that the TLA+ community is interested in that sort of thing, independnet of results useful in the course of a program specification/verification task.)

* If you're familiar with (esp. typed) functional programming, you might find Coq even easier to get into than TLA+. On both sides of the type theory vs sets/axioms divide, people should be careful not to confuse one's predispositions based upon educational background with the actual mental load of learning to use a tool.

There is room for both of these tools (and more!) in the world. I don't understand why they have to be in competition. A formalization of the C standard is a HUGE undertaking, independent of what tool was used. Let's celebrate that achievement!

unboxed_type on Nov 30, 2015 [-]

I would like to mention that Coq is also used to extract _verified implementations_ rather than model checking abstract algorithm design (as in TLA+). This property might be very desirable depending on situation. As for me this property looks very appealing.

pron on Nov 30, 2015 [-]

> To say that type-theoretic theorem provers (or the Coq system specifically) are merely academic toys compared to TLA+ is disingenious.

I never said that Coq is an academic toy. Some projects done in academia and used in the industry have been verified with Coq, but I am not aware of actual (direct) use of Coq in the industry (but would be interested to learn of any).

> I'm not sure I understand why "Computer Science-type math" is an unreasonable foundations for a program specification tool, which is what TLA+ is

I've never said it is unreasonable, only that it is foreign to most engineers (as well as most academic CS people), or rather, more foreign than basic logic/set theory.

I would say again that if you're comfortable with type theory, you might enjoy Coq better. If not, TLA+ would probably be more approachable. TLA+ has the added advantage of having a model checker, which is very useful when you don't have the resources required for a full proof, but still want a good confidence in the correctness of your algorithm.

> A formalization of the C standard is a HUGE undertaking, independent of what tool was used. Let's celebrate that achievement!

Oh, absolutely! It's just that stories like this make people interested in formal methods, and Coq, IMO, would be daunting for most. TLA+ is a good entry point to formal methods, even if you later choose to move to dependent-type approaches and tools.

yaantc on Nov 30, 2015 [-]

> ... but I am not aware of actual (direct) use of Coq in the industry (but would be interested to learn of any).

The first EAL7 certified smart card certification used coq, see: https://www.commoncriteriaportal.org/iccc/9iccc/pdf/B2404.pd...

pron on Nov 30, 2015 [-]

Oh, wow, they have verified a (JavaCard) JVM. Impressive! Thanks.

They mention > 117,000 lines of Coq with > 1600 proofs to get EAL7 for the JVM.

brohee on Nov 30, 2015 [-]

This link is about an EAL4+ evaluation

wolfgke on Nov 30, 2015 [-]

If you read slide 4, you'll see that the SIM card was evaluated at EAL4+ level and the design of the Java Card System at EAL7 level.

pron on Nov 30, 2015 [-]

BTW, what is the justification for using dependent type rather than stating assumptions/theorems directly in predicate logic over set theory (as in TLA+)?

Lamport has this to say on the matter:

The main virtue of these type theories is precisely that they are constructive. A constructive proof that two arbitrary numbers always have a greatest common divisor provides an algorithm for computing it [Thompson 1991]. Researchers, using tools such as Coq and Nuprl are investigating whether this can lead to a practical method of synthesizing programs.

You can perform classical reasoning in a constructive type theory by adding P ∨ ¬P as an axiom. The resulting system will probably be strong enough to handle any specification problem likely to arise. However, it will be no stronger than ZF, and it will be much more cumbersome to use.

In the end, he opted for set theory because it is simpler, more familiar, and more flexible. Are there any advantages to using dependent types? Is the main advantage indeed synthesizing programs from proofs?

nmrm2 on Nov 30, 2015 [-]

> Is the main advantage indeed synthesizing programs from proofs?

It's one major advantage, yes. And not just for the sake of software verification. Sometimes when proving a non-CS-type-math theorem you still want an efficient implementation of an existence proof.

But code synthesis is a lot harder than merely being constructive, and efficient code synthesis even more so. Use cases for Coq that depend upon code synthesis don't come for free just by avoiding LEM. So although it's fair to say that you can do classical reasoning in Coq, I'm not sure saying you can do synthesis from TLA+ if only you avoid LEM is fair at the moment (I might be wrong).

> Are there any advantages to using dependent types?

Here are a few other reasons for using Coq:

* Proof terms. There are two major disagreements here. The first grounds out in philosophy and the second basically amounts to "you really want to compose the proofs and make sure they are still a proof instead of just throwing away the proof and using the theorem" which has lots of good systems-y justifications

* Specifically wrt TLA+, Modus Operandi matters. If you want a formal proof in the typical sense of the word (rather than a verification of fact based upon model checker output), then Coq userlands's tactics and theorems are useful.

* Lots of people like to disparage the amount of algebra that FPers talk about. But it turns out that if studying those algebraic structures is your day job, the relationships between types and algebraic structures makes type theory convienant.

>simpler, more familiar

This is almost definitionally subjective. Which isn't meant as a criticism, in-so-far as one realizes that other people might have different experiences and so might (just as validly) consider other formalisms more familiar.

> and more flexible

There are (obviously) a lot of people who fundamentally disagree with the concrete arguments Lamport makes in this paper. I wish someone would annotate the paper with references to literature from the type theory community that address some of his concerns.

For the record, I don't disagree with Lamport. I think Lamport is *right* -- for some use cases, it certainly makes sense to stick with ZF. But he's also *wrong* -- sometimes, the type theoretic approach really is simpler, more familiar, and more flexible. E.g., in the case of this dissertation.

pron on Nov 30, 2015 [-]

> If you want a formal proof in the typical sense of the word (rather than a verification of fact based upon model checker output), then Coq userlands's tactics and theorems are useful.

TLA+ has them, too.

> in-so-far as one realizes that other people might have different experiences and so might (just as validly) consider other formalisms more familiar.

So you think some people are more familiar with type theory than with undergrad-level set theory (ZF)?

> sometimes, the type theoretic approach really is simpler, more familiar, and more flexible. E.g., in the case of this dissertation

I don't see it (probably because my type theory is very, very basic; I'm an algorithm's guy and the thought that a data-structure of mine might need type theory to be specified makes me shudder), as in I fail to see anything that isn't readily expressible in simple classical logic + temporal logic. Could you explain?

nmrm2 on Nov 30, 2015 [-]

> TLA+ has them, too.

Like I said, modus operandi matters. If I have to perform another model checking routine (possibly on an infinite state space) to slightly generalize a theorem, then in many cases that means I never really had a useful proof of it in the first place.

Also, Coq's mathematics library is extensive and often closely follows the proofs found in graduate mathematics text books. Last I looked into TLA+, it was squarely focused on algorithms and hardware (not even CS-related mathematics more generally -- specifically algorithms and hardware).

> So you think some people are more familiar with type theory than with undergrad-level set theory (ZF)?

Do I think there exists some population of people who are "more familiar" with type theory than with ZF? Yes, starting, for example, with every researcher whose primary area of study is type theory. In fact, I think that anyone who doesn't honestly believe there are such people must have an extremely low opinion of quite a number of very smart people.

(Also, set theory qua set theory gets really disgusting and kludgey really quickly, and is something that almost no undergraduate (or even phd student) is exposed to these days. The "set theory" you see in a discrete math course kind of barely scrapes definitions. I think most people who posit that "set theory" is nice and simple must've never really dove deem into the actual theory of set theory...)

(Also, most real math these days quickly abstracts away from germanely set theoretic constructions. Mathematicians don't work in ZF. They do not. Most will *tell you* they do not, if they even know what ZF is. Which a lot won't, because their undergrad discrete math course is probably the last time they saw a truly formal derivation. And the mathematicians who do know what ZF is often can't regurgitate an axiomatization of ZF. If you tell a mathematician that all of math is just set theory, most will chuckle and wink and say "why, yes, yes it is", and hopefully you're in on the joke...)

> I don't see it... I fail to see anything that isn't readily expressible in simple classical logic + temporal logic. Could you explain?

Did you read this dissertation's explanation of why separation logic is important?

To the extent that this observation is remotely reasonable, it's only in the sense that you can code up one logic in another. Which, well, all the world's a TM, as they say...

Anyways, I think the argument over whether type theory is a successful foundations for CS research and practice is already a closed issue. It demonstrably is. So is set theory.

pron on Dec 1, 2015 [-]

> If I have to perform another model checking routine

Why model checking? TLA+ is not a model checker. It's a specification and proof language. There *is* a model checker that can check TLA+ specs, just as there is a proof assistant that can check TLA+ proofs.

> must have an extremely low opinion of quite a number of very smart people.

Why low opinion? Type theory isn't "smarter", it's just more obscure.

> The "set theory" you see in a discrete math course kind of barely scrapes definitions. I think most people who posit that "set theory" is nice and simple must've never really dove deem into the actual theory of set theory...

Thankfully, that basic set theory -- the one that barely scrapes the definition -- is all that's required to specify real-world programs.

> Did you read this dissertation's explanation of why separation logic is important?

Skimmed it. Unless I'm missing something (which is quite possible) it seems very straightforward to use in TLA+: you just define the operators just as they're defined in the paper. There's no need to "code" the logic in any arcane or non-obvious way; just define it. In fact, this short paper describes doing just that and it seems pretty basic: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.570...

> Anyways, I think the argument over whether type theory is a successful foundations for CS research and practice is already a closed issue. It demonstrably is.

It never even crossed my mind to make the case that it isn't. That type theory is an interesting topic for research is obvious. I also never implied that dependent-type tools can't do the job; as you say, they demonstrably can. I simply asked if I -- a programmer working in the industry -- had a program I wished to verify, whether there was a reason why I would reach for a dependent-type tool vs. a classical logic tool. So far, I don't see why I should. It seems to basically boil down (at least according to the responses in this thread) to whether I prefer working with direct, classical logic or with dependent types.

nmrm2 on Dec 1, 2015 [-]

> TLA+ is not a model checker. It's a specification and proof language. There is a model checker that can check TLA+ specs, just as there is a proof assistant that can check TLA+ proofs.

For the third (!) time, modus operandi matters.

Coq's huge library of tactics and proofs of facts that TLA+ might just throw at a model checker are, in my experience, very useful starting points for things that can't be model checked.

Ostentibly TLA+ could build up an equivalent library of tactics and proofs.

> Why low opinion?

Even supposing type theory is "more obscure", it's pretty hard to believe that someone could study it for 30 years and not understand it better (in some sense) than ZF. That's a pretty extraordinary claim.

> basic set theory -- the one that barely scrapes the definition -- is all that's required to specify real-world programs.

Not really -- you often need non-trivial constructions when proving properties about rewrite systems within set theory. Those constructions are *much* harder to understand than induction on algebraic datatypes, which is why type theoretic approaches have dominated in formalization of programming languages.

>Unless I'm missing something...

I'm disinclined the dismiss separation logic just because you can code up some of its ideas in another logic (in a way that counts as a novel contribution worthy of publication, no less).

The clarity with which separation logic presents the idea and use of the separating conjunction has obviously influenced the paper you cited. I would think this is reason enough to summarily dismiss these "everything's a TM"-style arguments you're making here. The authors of the paper you cited are only not using separation logic in a kind of meaningless sense IMO.

> So far, I don't see why I should.

This sentiment kind of reminds me of people who have never typeset mathematics complaining that LaTeX is useless... which is to say, perhaps you simply aren't interested in and so don't care about the use cases where Coq shines? Lots of people -- including me -- have already pointed out a lot of these use-cases in this comment section.

But really, based on the arc of this conversation, I would be enormously surprised if you chose to use Coq to prove X, even if there were a proveX tactic already written in Coq and the TLA+ implementation would take all day ;-)

pron on Dec 1, 2015 [-]

> Coq's huge library of tactics and proofs of facts that TLA+ might just throw at a model checker are, in my experience, very useful starting points for things that can't be model checked.

I don't understand what you're saying. What does TLA+ have to do with the model checker? TLA+ is a language, while TLAPS (the proof system) and TLC (a model checker) are two different products (each supporting different subsets of TLA+, BTW). TLAPS comes with its own library of proofs. I don't know how extensive they are.

AFAIK, you can't (and certainly you don't) take a fact verified by the model checker and state it as an axiom of your proof. The interaction between the two -- if any -- is that Lamport encourages you to model check your spec before trying to prove it (assuming you want to prove it, or even model-check it) because proving things that are false may prove hard. However, the result of the model checker play no part in the proof.

When it comes to proofs, TLA+ is intended to be declarative and independent of the proof engines used (by default, TLAPS uses SMT (any of several solvers), Isabelle and another couple of tool, passing every proof step to every tool unless a tactic specifies a particular one).

This is a nice introduction for the design of the TLA+ proof language: http://research.microsoft.com/en-us/um/people/lamport/pubs/p... which includes (in the appendix) a few proofs in calculus.

> it's pretty hard to believe that someone could study it for 30 years and not understand it better (in some sense) than ZF. That's a pretty extraordinary claim.

I did not mean to claim that type theory researchers don't understand it better than basic college set-theory. I meant that *in general*, engineers and academics are more familiar with basic set theory.

> just because you can code up some of its ideas in another logic (in a way that counts as a novel contribution worthy of publication, no less).

That was a technical report that mentioned in passing using an approach similar to separation logic.

> "everything's a TM"-style arguments you're making here

I am not making that argument. My argument is that most engineers and CS academics (except those in PLT) would be more familiar -- and therefore more likely to use -- a tool based on basic math they know well. In addition, I haven't seen anything to suggest that the complexity difference between specifying a system using dependent types and specifying it in TLA+ is significant at all in either direction. Learning Coq, however, requires much more effort, and AFIK, it doesn't support formal verification mechanisms other than proofs (e.g. model checking)

> perhaps you simply aren't interested in and so don't care about the use cases where Coq shines?

Perhaps. I design data structures (mostly concurrent), and my interests lie in specifying and verifying those as well as distributed systems, and less in verifying compilers. My current TLA+ spec is too complex to be proven anyway (regardless of the tool used) in any feasible amount of time, so I rely on just the spec as well as some model checking.

It is also true that I may be biased. As an "algorithms guy" I feel more at home working with tools and languages designed by other algorithm people (and more importantly *for* algorithm people) rather than PLT people.

Nevertheless, I'm interested in learning the difference between the two.

> But really, based on the arc of this conversation, I would be enormously surprised if you chose to use Coq to prove X, even if there were a proveX tactic already written in Coq and the TLA+ implementation would take all day

Considering that learning Coq would take me much longer than a day, that would be a wise decision.

ScottBurson on Nov 30, 2015 [-]

I'm guessing LEM = Law of the Excluded Middle.

nmrm2 on Nov 30, 2015 [-]

Yes.

pacala on Nov 30, 2015 [-]

FWIW, https://en.wikipedia.org/wiki/Mizar_system is based on set theory, and has traction in the math community.

fmap on Nov 30, 2015 [-]

Ok, there are a number of points you can make comparing set theory and type theory... but the most important point here is that it is a false dichotomy! From a logical point of view "type theory" is a particular formalism for describing logical frameworks. ZFC is the "type theory" of first-order predicate logic plus a few axioms which add a type of sets, a predicate for set membership and assert certain properties of set membership. You can add function types to ZFC and you do obtain a strictly stronger system (Higher-Order ZFC). The main practical advantage is that you do not need to encode your problem into first-order logic.

For instance, in higher-order set theory a (very strong) form of the axiom of choice asserts that there is a function "epsilon : set -> set", such that "epsilon x \in x" for x not empty. Expressing the same in first-order ZFC turns the whole statement on its head since you need to encode this function somehow. This leads to (almost) equivalent but misleading statements, such as "the intersection of a set all of whose inhabitants are non-empty is non-empty". Now if you look into TLA+, you will find that this choice operator epsilon has been added to the language, because it is so useful in practice! And this is just one instance of something that "type-theory" can add over classical ZFC. More to the point, higher-order specifications are crucial for the verification of higher-order programs. Higher-order programs are in turn one of the most useful tools for writing modular code.

Sorry for the nitpicking; let me get back on topic.

The main advantage of Martin-Löf style type theory is that it makes it easy to build new abstractions. For instance, the Bedrock project (http://plv.csail.mit.edu/bedrock/) builds an extensible program verification environment within Coq. This is only possible because the underlying logic is so expressive that you can build internal models for whichever pet logic you want to reason in. From a practical point of view dependent types and the conversion rule - computations in proofs - allows you to build scalable (reflective) proof automation while retaining a small trusted base.

The constructive aspect of Martin-Löf type theory is very pleasing from a philosophical point of view, but again in practice it's not the most relevant aspect. The main reason for me to use something other than classical logic is that there are more models of intuitionistic logic than of classical logic. This gives you more freedom in designing a logic which makes it easy to reason about your particular problem domain. For instance, if you work with Cyberphysical systems, you might like to work within the framework of synthetic differential geometry where all functions are smooth and the real numbers are as simple as they can ever be.

What so many people seem to be ignoring is that the logic you use for reasoning is part of your design space. Saying "I'll just use ZFC" is exactly the same as a programmer exclaiming "I'll just use COBOL". Leslie Lamport designed TLA+ to make it easy to reason about distributed systems. There may well be a nicer logic for reasoning about distributed systems, but you will probably not find it among the models of classical ZFC, since there aren't really that many of them...

On the other hand, there are many different flavors of type theory, just waiting to be discovered. And in my view that's the main advantage of looking beyond set theory.

pron on Nov 30, 2015 [-]

> For instance, in higher-order set theory a (very strong) form of the axiom of choice asserts that there is a function "epsilon : set -> set", such that "epsilon x \in x" for x not empty

Such higher-order statements are easily expressible in TLA+ as long as you specify the domain of the sets on which you operate, which you should always be able to do when specifying real programs, even higher-order programs. Here it is in TLA+:

```
∃ epsilon ∈ [SUBSET S → S] : ∀x ∈ SUBSET S : epsilon[x] ∈ x
```

You just need to specify S (or parameterize it).

> Saying "I'll just use ZFC" is exactly the same as a programmer exclaiming "I'll just use COBOL".

I don't think it's the same at all, considering that 99.9% of math is done with "just ZFC". I don't doubt that type theory may let us explore other logics, but as a practical way for specifying real programs *today* I fail to see a case in favor of dependent types. Sure, I believe dependent types may let you define all sorts of modal logic, but in practice, temporal logic has proven a good way to specify algorithms. So I think that a more suitable analogy would be, "Why confine your thinking to von-Neumann machines when you can use a meta-logic to specify programs for Quantum computers and other computational models". In that case, "I'll just use a von-Neumann machine" seems like a very reasonable choice for any work that isn't theoretical.

fmap on Nov 30, 2015 [-]

I am honestly unconvinced on "most of mathematics" being just ZFC. You see this claim a lot, but when reading formal developments in mathematics I often see justifications involving "small categories" or statements which should hold in ZFC with "enough universes a la Tarski".

But this is missing the point. There is a lot of active research work on program logics for reasoning about e.g., concurrent programs in weak memory models. You may be able to encode the model of something like Iris (http://plv.mpi-sws.org/iris/) in TLA, but the challenge lies in doing this in a usable format...

pron on Dec 1, 2015 [-]

I am working on specifying a weak memory model system in TLA+ right now, and have yet to encounter any issue.

fmap on Dec 1, 2015 [-]

Neat! Do you have some pointers to your work? I would be especially interested in example verifications of lock free data structures and in comparing the proof effort to logics custom built for this purpose.

All the constructions I know use at least second order logic and something like Iris is - as far as I know - most elegantly described internal to the topos of trees. If you have found a good way to avoid this complexity then that's a significant contribution.

pron on Dec 1, 2015 [-]

> Do you have some pointers to your work?

At the moment we're not open-sourcing the spec (I'm an engineer, not an academic) but we might in the future. We've opted not to even try to prove the system's correctness, as that might take decades (it's far more complex than, say, seL4). Instead, we're just formally specifying it, and model-checking it with a finite model. The software in question, however, is itself open-source and is described here: http://blog.paralleluniverse.co/2012/08/20/distributed-b-tre...

> All the constructions I know use at least second order logic

Concurrent algorithms are specified with TLA+ (in the industry, I mean) quite frequently. Also, specifying the memory consistency properties does require second order logic, but as I've shown in another comment, TLA+ handles second-order logic gracefully and in a very straightforward matter (trivially, I would say). You simply write:

∃ ordering ∈ [operations → SequenceOf(Op)] : ConsistencyProperty(ordering[operations])

pron on Dec 1, 2015 [-]

Sorry, that should be:

```
    ∃ epsilon ∈ [SUBSET S → S] : ∀x ∈ SUBSET S : x ≠ {} ⇒ epsilon[x] ∈ x
```

As a matter of fact, TLA+ also allows unbounded quantifications, so you could write:

```
    ∃ epsilon : ∀x : x ≠ {} ⇒ epsilon[x] ∈ x
```

However, this form is rarely used because TLC, (TLA+'s model checker, which only supports a subset of TLA+), doesn't support unbounded quantifiers, and they are not necessary when modeling real programs. I'm not sure about TLAPS (the proof assistant), but I believe it handles it fine.

fuklief on Nov 30, 2015 [-]

Coq is also used in industry, in particular by Airbus with the CompCert C compiler.

feider on Nov 30, 2015 [-]

From CompCert site: "What sets CompCert C apart from any other production compiler, is that it is formally verified, using machine-assisted mathematical proofs, to be exempt from miscompilation issues."

I'm not very into C language - what are these miscompilation issues? What is the practical impact that CompCert C brings?

witty_username on Nov 30, 2015 [-]

Miscompilation issues means bugs in the compiler causing incorrect assembly to be generated.

feider on Nov 30, 2015 [-]

I understood that. Sorry if my question was bit unclear. What I wanted to know was _specifically_ what these bugs are / what causes them / what is practical impact (i.e. probability of application crashing or other serious event due to use of GCC etc)

sanxiyn on Nov 30, 2015 [-]

Check out "Finding and Understanding Bugs in C Compilers": http://www.cs.utah.edu/~regehr/papers/pldi11-preprint.pdf.

The paper explains what-is-bug/what-causes-it/what-is-impact for selected bugs. Complete list of 282 compiler bugs found (79 GCC, 203 LLVM) is also available online: http://embed.cs.utah.edu/csmith/.

feider on Nov 30, 2015 [-]

Thank you, that paper seems to be surprisingly easy to read (being an academic paper that is).

nickpsecurity on Nov 30, 2015 [-]

Especially see the CompCert section on page 6 of the Csmith PDF to see what a difference formal verification made. Note that compilers don't have to go that far: typed, functional, thoroughly-tested code in something like Ocaml or Haskell can achieve 80-90% of that quality with relatively little effort. Hence, why I promote rewriting and further extending things like LLVM in languages like ML, Ada, or Eiffel that support advanced checks.

pron on Nov 30, 2015 [-]

Do Airbus engineers write Coq specifications or just use CompCert (a C compiler formally verified with Coq in academia)?

nmrm2 on Nov 30, 2015 [-]

This distinction is disingenious (*edit: I probably mean spurious*).

The number of people in industry who use *either* tool is incredibly small and is likely to remain so; developers spend 20-50% of their time on test suites and *STILL* don't feel it's cost-effective to write down formal specs, even in TLA+-style specifications.

What matters is that important libraries and frameworks can be verified, not that everyone and his brother can use formal methods for every WordPress website they churn out.

eli_gottlieb on Nov 30, 2015 [-]

>The number of people in industry who use either tool is incredibly small and is likely to remain so; developers spend 20-50% of their time on test suites and STILL don't feel it's cost-effective to write down formal specs, even in TLA+-style specifications.

How many zero-day security flaws have to occur in core infrastructure like, for instance, OpenSSH before the laziness of engineers about learning "academic" tools stops being a cost-effective excuse *not* to use formal methods?

jessaustin on Nov 30, 2015 [-]

...disingenious.

English usage note: if the word you meant to use was "disingenuous", I would suggest "spurious" instead, as it expresses similar feelings about "this distinction", without *also* attributing unsavory motives to 'pron.

ScottBurson on Nov 30, 2015 [-]

Hey, I like this word "disingenious". I think it could mean "a bad idea being promulgated to displace (what the speaker considers) a much better idea". We need a word for that :-)

pron on Nov 30, 2015 [-]

> The number of people in industry who use either tool is incredibly small and is likely to remain so

I agree, but I see some chance of TLA+ of getting wide(r) adoption, precisely because it is rather easy to learn and uses math that most engineers are already familiar with. Also, it allows gradual verification: specification -> model-checking -> proof (with each additional step being completely optional)

> not that everyone and his brother can use formal methods for every WordPress website they churn out.

I don't know about WordPress websites, but Amazon engineers do use TLA+ to specify many (most?) AWS services.

nmrm2 on Nov 30, 2015 [-]

My point was just that formal methods can have a tremendous positive impact even if academics are the only ones using them, as long as the output from those efforts *do* get used. So I'm not sure distinguishing between "using formally verified software" and "using a formal methods tool directly" is a helpful distinction when measuring industrial impact.

Also, formalizing architectural properties for the world's most popular cloud service is probably even rarer a task than formalizing language specifications.

pascal_cuoq on Nov 30, 2015 [-]

Both. “They” (and by they I do not necessarily imply a large number of engineers at Airbus, but some people who are engineers at Airbus) use CompCert and “they” also use Coq in order to finish up the correctness arguments for part of the properties that part of the software is supposed to have:

https://www.lri.fr/~marche/M2-TIP-Airbus.pdf

unboxed_type on Nov 30, 2015 [-]

Also take a look at this thread, were I briefly describe our experience of using all the spectre of verification tools. https://news.ycombinator.com/item?id=10220264

random778 on Nov 30, 2015 [-]

Would you advise one to start directly with TLA+ or work through some logic/discrete math seperately beforehand?

agentultra on Nov 30, 2015 [-]

Not the author but a recent book I've read on the subject served as a gentle introduction for me: The Little Prover [0]. If you can get past the lisp syntax you can get through this book with an informal and incomplete education in set theory and an intuitive understanding of logic.

[0] https://mitpress.mit.edu/index.php?q=books/little-prover

pron on Nov 30, 2015 [-]

Lamport's tutorial (the *TLA+ Hyperbook*) as well as the old TLA+ book (*Specifying Systems*) -- both available freely on the TLA+ homepage -- cover all the math you need (which is little more than logical conjunction/disjunction/implication, the quantifiers for all/exists, and set membership/union/intersection). But it is perhaps better to be familiar with those simple concepts beforehand, although note that what you need is very, very basic. As nmrm2 said, there's no need for a full Discrete Math course (or a whole book).

nmrm2 on Nov 30, 2015 [-]

If you have had the equivalent of an undergraduate course on Discrete Mathematics you are probably fine; an introduction to logic course (covering e.g., soundness and completeness of propositional/first order logics) probably isn't necessary.

There is a self-contained book; IMO chapter 1 gives a good overview of what you should be comfortable with:

http://research.microsoft.com/en-us/um/people/lamport/tla/bo...