Machine Obstructed Proof

How many months can it take to verify 30 assembly instructions?

Nick Benton
Microsoft Research
nick@microsoft.com

After years doing programming language theory without going near a proof assistant, I was finally convinced by the POPLmark 'buzz' and conversations at ICFP'05 that it was time to try one. I was working on relational reasoning for low-level programs, for which mechanization seemed useful and feasible. The metatheory was tricky, proofs about particular programs were so tedious that I made mistakes doing them by hand and, unusually, there wasn't any α-conversion involved, so maybe I could get something done *before* getting involved in long disputes about substitution.

A scary feature of automated theorem proving is that it constantly forces one to deal with abstruse foundational matters, starting when one decides which prover to use. 'Mathematics floats' won't cut it: you *have* to choose a party or religion to belong to. It'd be nice if it didn't matter, but extant systems do (seem to) have foundational limitations that could trip one up months later. Wondering about the metatheoretic strength of results proved within some internalised version of category theory seems unlikely to help get your work done. I picked Coq: it 'felt' right, it seemed unlikely to leave me painted into a foundational corner, and there was an expert two doors from me.

The workshop description says "the available tools are [...] difficult to learn, inadequately documented, and lacking in specific library facilities required for work in programming languages". I can confirm that I have rarely felt as stupid and frustrated as I did during my first few weeks using Coq.

Tactical theorem proving is like an extreme form of aspect-oriented programming. This is *not* A Good Thing, particularly for beginners. Scripts are unreadable by themselves, as one has no idea what the tactics are doing to the proof state, and the documentation for them is incomprehensible to the novice. The only thing that works is lots and lots of trial and error in an interactive environment, and I still couldn't give a coherent general description of what some of the tactics I've used many times actually *do*, or how they differ from half a dozen apparently similar ones. And basic ones are still missing; I spent days fighting with `elim`, `case`, `destruct` and variations on `induction` and still kept finding myself having done case splits without the information about which branch I was in. This was so frustrating I gave up on Coq (and spent a week playing with HOL Light) until Georges Gonthier showed me the magic, and frankly bizarre, incantation `generalize (refl_equal x); pattern x at -1; case x.`

Metalogical vertigo lasts a long time. `Set` or `Type`? `Bool` or `Prop`? It was cool to define record types combining sets with propositions, e.g. a relation on states plus a function from states to sets of locations and a proof that the relation is invariant under changes to the state outside the range of the function. Less cool to have to then make my own copy of the standard list library because that's only parameterized over `Set`, whilst records including `Prop`s live in `Type`. Any new user will surely ask about extensional equality immediately: "why isn't it there?" and "can I just assume it?". It's hard to get a straight answer to either of these questions. The book doesn't mention extensionality in its 450-odd pages! The FAQ makes mysterious remarks about extraction. Experts on the mailing list have sophisticated disagreements. But the poor user is none the wiser about whether asserting `forall f g, (forall a, f a = g a) -> f=g.` will make the sky fall down.

There are bugs. On Windows, CoqIDE falls over, whilst Proof General only works with the original 2004 version of Coq 8. I spent ages defining `Setoid` structures on everything, only to find Setoid rewriting throws an 'anomaly' exception in interesting contexts.

I had many similar difficulties, but then started to make progress. Just having intermediate stages of the work in a computerized form rather than on many pages of paper proved a major benefit. Far more often than I'd expected, one can alter definitions and then mildly tweak the previous version of a proof to keep it up to date. On paper, I tend to keep going back to the top and doing everything from scratch to be sure everything is still consistent.

Constantly permuting large separated conjunctions of assertions to bring the active one to the head was painful. I first attacked this by defining some tactics, but then discovered the joy of computational reflection: replacing a shallow embedding of assertions with a more intensional representation using lists allowed me to permute goals and hypotheses by reducing recursive functions that were proved to preserve the semantics. A similar change in representation of machine code fragments allowed me to replace custom tactics with computation and rewriting when extracting instructions.

Automated proving is not just a slightly more fussy version of paper proving and neither (Curry-Howard notwithstanding) is it really like programming. It's a strange new skill, much harder to learn than a new programming language or application, or even many bits of mathematics. I'm resistant to investing significant effort in tools (I don't write clever TeX or Emacs macros), but the payoff really came the second time I used Coq: I was able to prove some elementary but delicate results for a different paper in just a day or so. Coq is worth the bother and it, or something like it, is the future, if only we could make the initial learning experience a few thousand times less painful.

The answer is about four [1].

References

[1] N. Benton. Abstracting Allocation: The New `new` Thing. 2006.