The Proof Assistant as an Integrated Development Environment

Nick Benton

Microsoft Research
nick@microsoft.com

Abstract. We discuss the potential of doing program development, code generation, application-specific modelling, and verification entirely within a proof assistant.

Managing the interaction between programming and proving creates challenging problems in the design of languages, logics and user interfaces. Almost independent from the deep research problems associated with designing reasoning methods for programs, it is not clear what a 'ideal' environment or tool-chain for producing verified software might even look like.

Many automated verification tools start with conventional languages, compilers and development environments, which are extended to allow Hoare-style assertions and invariants to be added as annotations, for example as structured comments. Assertions are verified behind the scenes, for example by an SMT solver, and failures reported by 'red squigglies' and textual error messages, just like conventional syntax and type errors. This comfortably familiar approach is minimally disruptive to development practices and can work very well, particularly for comparatively simple or specific properties. But there are limitations.

Firstly, when the automation fails (which is the common case), the programmer has to change the program, change the specification, or add further annotations as hints to the prover. Making such changes can be hard: the programmer is, at least morally, interacting with the prover to try to construct a proof, without any direct feedback of what the proof looks like and only one mechanism – stating new lemmas – to guide its construction. The alternative of generating verification conditions for residual obligations and shipping them to a proof assistant allows some interactivity, but often generates large, incomprehensible goals, with no clear link back to the original program. Accurately relating the annotation language, prover language, proof assistant language and, in the case of interesting properties, the semantics of the programming language is challenging. Even arranging to persist the VC proofs along with the program isn't entirely trival.

Secondly, the idea of just marking up a conventional program with a few well-chosen pre/postconditions and invariants and then pressing a magic button only gets one so far. There are simple syntactic limitations on what one can say if specification level constructs must align with programming language ones (e.g.

insisting that each procedure has a single specification, or disallowing quantification scoped over more than one procedure). More fundamentally, to verify 'deep' functional correctness properties one needs a rich logic just to write the specifications. Really verifying a compiler, linear algebra library, crypto protocol or video decoder involves a model of the desired behaviour that can only be formally captured in a language that is powerful enough to express, essentially, arbitrary mathematics, especially if we want the specifications to be comprehensible and modular. The necessary reasoning about such models is not generally fully automatable, so some form of explicit representation of proofs seems unavoidable.[1] As the amount of text involved in defining the model, proving properties of it, writing specifications and establishing that they hold of the program (even with the help of some automation) can easily be at least as great as that associated with the actual program (and almost always takes at least as long to write), an environment's support for convenient proving is arguably more important than its support for programming.

An attractive alternative to Hoare-style verification is to write one's software in a dependently-typed language, such as Coq or Agda, in the first place. Dependent type theories do have the power to express the mathematically rich specifications one needs for full functional correctness (as well as a full range of simpler ones, of course). And dependently typed languages provide an elegant integration of programming and proving, such that programs can be said to be 'correct by construction' (even if achieving correctness requires a more elaborate construction). There are actually two slightly different styles of writing verified software in dependently-typed systems: one one can either use the full power of dependency to capture specifications directly in the types of functions, or write conventionally-typed programs about which one separately, though still all in the same system, proves correctness theorems. The latter style is common in Coq – e.g. for CompCert [13] – as there is good tactical support for proving, strongly-dependent programming is trickier than it should be, and because extraction of the computationally relevant parts of a program to OCaml for actual execution is thereby more straightforward.

Beautiful though it often is, programming directly in type theory comes with its own trade-offs. Firstly, one must program in a rather fundamentalist pure functional language, in which only provably total functions can be written. Such a language is not obviously the most natural choice for writing *all* the kinds of software component one might wish to verify. Secondly, compilation is typically via extraction/translation to a more conventional functional language, such as OCaml or Haskell. This is convenient both for reusing existing infrastructure (optimising compilation, runtime systems, etc.) and for interfacing verified and unverified code, for example to add impure IO and UI code to a verified algorithmic core. But if we insist on a very high level assurance (which we often

[1] For particular applications, such as cryptography, or forms of property, such as memory safety, one can usefully use an interactive prover to verify the metatheory associated with specialized automation, but in the general case it is impossible to insulate the developer of seriously verified software from thinking about proofs.

don't), then we might have reservations about including a sophisticated compiler, its runtime system and libraries in our trusted computing base.[2] And if the theorems we really want to prove make non-trivial statements about the IO behaviour of our program, that behaviour really needs to be brought within the scope of our formal reasoning.

An popular alternative approach is to use a proof assistant to reason about a program written in a conventional language, such as C or ML. Given a formalized semantics for the programming language and a representation of the program code as an object in the prover's internal language, one can prove program properties by many methods, including Hoare-style reasoning, with the soundness of the reasoning principles being formally justified in terms of the semantics of the specific language, which may be high or low level, pure or impure. Actual executable code is generated by running the textual version of the program through a conventional compiler. This approach is particularly appropriate for verifying low-level code and has been successfully used in many significant projects, such as the sel4 verification [10]. As with program extraction, an external compiler becomes part of the TCB, though in both cases that concern can be removed by a separate formal verification of the compiler itself.[3] The state of the art here is the Verified Software Toolchain [1], which builds provably sound separation logic reasoning (and other program analysis tools) for a version of C on top of the CompCert semantics and compiler.

As previously mentioned, interactive verification of programs written in standard programming languages has huge advantages. In many practical situations, it will, clearly, be the only kind of verification that is acceptable. And yet, as well as the potential weakening of the guarantees that are obtained if one does not also use a verified compiler, it is inherently a little clumsy. One's workflow involves tools to translate between the programming language syntax and that of whatever representation is chosen in the prover's language. Naive representation choices (of the sort that would mesh well with a compiler correctness proof and are much more acceptable in metatheoretic work) lead to theorems being proved about objects that are rather more unwieldy than the original source, such as abstract syntax trees with their own encoded representations of definitions, scope, etc. Even slightly more idiomatic translations (that, for example, map definitions in the language to definitions in the prover) make it trickier to verify formally the correctness of the workflow as a whole. There are numerous complexities, from differences in identifier naming rules, through dealing with preprocessors to linking, build scripts, code generation, and keeping the program and proof in sync.

[2] And whilst the story about the relation between Coq semantics and OCaml semantics might be reasonably clear for whole programs, it seems less so for components, as the OCaml type system can't express the purity constraints assumed by Coq functions.

[3] Or alleviated by testing, though even that really ought to be strongly connected with the formalized semantics. One reason for verifying programs written in an existing language is to be able to use existing tools.

Now, proof assistants are powerful tools, verification researchers are smart, and code is malleable stuff. Many ingenious strategies for reducing the pain of impedence matching have been devised, including the circular verification of the F* typechecker [21] and Myreen et al.'s automatic extraction of HOL functions, together with correspondence proofs, from low-level programs [17, 16]. But ultimately, a proof assistant like Coq is in many respects simply a better programming language than most conventional ones. Ideally, we'd like to keep the expressive structuring and integration between programming and proving that we get from programming in type theory and *also* have high assurance compilation down to the machine, the ability to do low-level programming, and effects, including general recursion.

Various projects have involved writing imperative (or other non-purely-functional) programs directly in a proof assistant. But this is often just to do examples for papers on metatheoretic work: a small program is coded up as a term in an AST type, or directly into some more semantic representation, and something interesting is proved about it, but it is not often suggested that this is a way of writing programs one would ever want to actually run. A notable exception is Ynot: an ML-like, higher-order, imperative language, with Hoare-style assertions in types [18]. Ynot is built as a library in Coq, from which it inherits structuring and proving mechanisms. The model of effectful programs is axiomatised rather than formally verified in Coq, and programs are run via extraction to OCaml, so Ynot may not quite achieve the highest level of assurance, but the programming model of 'type theory with effects' is an attractive one. Verified programs that have been written in Ynot include web services [23] and a simple database [14].

Over the last few years, Andrew Kennedy and I, together with a number of collaborators, have been investigating formally verified compilation and reasoning principles for low-level level languages. Like many others, we aim to verify systems-level code right down to the hardware and so, having done several bits of work involving very idealized assembly code and abstract machines, we embarked on constructing a Coq model of (a sequential subset of) x86 machine code. The model is foundational in style: starting from bits, bytes and words, on top of which we model the machine architecture, instruction encoding and decoding, and the operational semantics.

The scientifically 'deep' part of the project involves the design and semantics of a separation logic for unstructured machine code, supporting both first- and higher-order frame rules, a full range of intuitionistic connectives and a 'later' modality, all with good logical properties [8]. Along the way, however, we also discovered that Coq was not *such* a bad system for actually writing the programs we wanted to verify. Being based on type theory, Coq can do more than contemplate eternal verities: it can actually compute. We implemented an assembler in Coq which, thanks to user-defined notations (including custom binding forms), is syntactically compatible with existing assemblers. The assembler has been proved correct, but also actually runs inside Coq: we can extract a bootable

binary image or a runnable .exe file from Coq, with no other external dependencies.[4]

Moreover, Coq's powerful abstraction mechanisms allow us to define conveniently parameterized higher-level programming constructs, including control structures and a variety of calling conventions, as macros. These macros also come with their own verified specifications, allowing one to move up the abstraction hierarchy in both programming and proving [9]. Chlipala's Bedrock system [5] takes a similar approach, allowing imperative programs to be written within Coq using very conventional-looking macro syntax that incorporates pre/post conditions and invariants and supports a very high degree of Coq automation for producing foundational proofs about the generated (machine-independent) low-level code.

One could keep going up until one had recreated most of the features of a general-purpose high-level language, but we are more excited about the prospect of producing verified code via much more explicit orchestration of staging and metaprogramming within the prover, combining general purpose programming abstractions with the use of embedded domain-specific languages; sharing pieces, but each with their own metatheory and code generation strategies. As a small example of the kind of thing we have in mind, we implemented a verified compiler for regular expressions that builds on an existing third-party formalization [4] of the theory of Kleene algebras – not merely at the specification level, but reusing a verified and computable function from regular languages to finite state machines as part of the compilation process.

Such an approach is not universally applicable, but for producing, say, a small, foundationally verified operating system kernel, the use of a conventional language seems eminently avoidable. Indeed, combining domain-specific compilers specialized to packet processing, scheduling, protocol definitions, event processing, policy checking, and so on, and able to describe those domains and their theory declaratively, using the full power of type theory, rather than indirectly through an implementation of some aspect of them in a general-purpose, low-level programming language, is very appealing. There is much talk about 'model-driven' software engineering, but it would be good to do it for real, using tools that can both actually build meaningful models and generate code that provably implements them. Mixing up different programming and specification paradigms might seem (or even be) a recipe for chaos, but the need to verify composed systems at least keeps us honest, and not only were we going to have to think carefully about the specifications of boundaries in any case, but those specifications provide guidance for designing the combination.

Even for unverified software, there is a trend towards integrating special purpose sub-languages into mainstream, general-purpose ones. $C\sharp$ has been extended with (amongst other things) asynchronous concurrency, reactive programming, database queries and parallelism, while $F\sharp$ goes even further by integrating heterogenous external data sources and models [22]. Trying to make such a range of features and paradigms fit together into a single coherent language design is

[4] Well, apart from hex2bin and a small bootloader borrowed from Singularity.

challenging, to say the least, and conventional type systems are often not up to the job of expressing domain-specific invariants (e.g. noninterference between parallel tasks). If we are going to verify multiparadigm programs, we do have to capture those invariants in our specification language, so perhaps that could take the place of, or augment, types at the programming stage too.

Furthermore, the potential performance advantages of metaprogramming and domain-specific code generation are considerable. Serious software (presumably the only kind on which one would spend the effort of verification) already often makes use of code generation techniques, from parser generators and template metaprogramming to the custom approach of, for example, FFTW [7]. Composing code generators, rather than working with a monolithic compiler, would allow many optimizations (such as smoothly combining manual memory management and garbage collection), limited only by the effort one is willing to put into establishing their safety.

Of course, there are many obstacles, both large and enormous, to be overcome before the rather utopian vision of UNCOL-with-extra-maths [6] can be realized: Computation within Coq is comparatively slow, and naive definitions often don't compute at all. Despite much research, it is still hard to work comfortably with object languages with binding. Interfacing different styles of specification may prove impossibly hard. It's not really clear how to factor definitions to allow sharing of important optimizations like register allocation, or to get the right degree of machine-independence. We've given no thought to debugging or profiling. And so on. Nevertheless, for a restricted range of high-assurance verification tasks, multiparadigm code generation directly from a proof assistant is an exciting and promising research direction.

More broadly, programming languages have, in a sense, lost control of their environments. However good we are at compiling and verifying individual languages, modern software components increasingly live in a complex, heterogenous world, with rich interfaces to other components, libraries and services. Building genuinely trustworthy *systems* means that the scope of specification and verification has to extend beyond closed programs written in a single language. Modern proof assistants are simply the only tools we have in which *all* the artefacts in which we are interested (programs, languages, models, specifications, proofs, compilers . . .) can coexist and be formally related.

The research community is beginning to build up quite a collection of machine formalizations of important artefacts, including network protocols [3], machine architectures, languages, logics, and programming-related theory [2]. These are expensive to construct, and it is a shame that many are abandoned after a couple of papers. However, there are encouraging signs that reuse is not only possible, but is really happening: VST [1] builds on CompCert [13], Bedrock [5] on XCAP [19], and the CakeML compiler [12] on components also used in other projects, including a (tested) model of x86-64 [20, 15] and (a translation of) a formalization of Parsing Expression Grammars [11]. As such formalizations mature, it should be possible to integrate them into the process of software development, rather than just post-hoc verification.

A mature high assurance development environment will probably look more like Visual Studio than an Emacs buffer with blue highlights. But the underlying technology that ties it all together should be logic and type theory.

Acknowledgements

My thanks go to Andrew Kennedy and to the other friends who've contributed and collaborated with us on many aspects of mechanized reasoning about software: Pierre-Evariste Dagand, Chris Hawblitzel, Chung-Kil Hur, Guilhem Jaber, Jonas Jensen, Neel Krishnaswami, Conor McBride, Georg Neis, Marco Paviotti, Valentin Robert, Nicolas Tabareau, Carsten Varming, Uri Zarfaty.

References

1. A W Appel, R Dockins, A Hobor, L Beringer, J Dodds, G Stewart, S Blazy, and X Leroy. *Program Logics for Certified Compilers*. CUP, 2014.
2. N Benton, A Kennedy, and C Varming. Some domain theory and denotational semantics in Coq. In *22nd International Conference on Theorem Proving in Higher Order Logics (TPHOLs)*, volume 5674 of *LNCS*, 2009.
3. S Bishop, M Fairbairn, M Norrish, P Sewell, M Smith, and K Wansbrough. TCP, UDP, and Sockets: Rigorous and experimentally-validated behavioural specification. volume 2: The specification. Technical Report 625, University of Cambridge Computer Laboratory, 2005.
4. T Braibant and D Pous. An efficient Coq tactic for deciding Kleene algebras. In *International Conference on Interactive Theorem Proving (ITP)*, volume 6172 of *LNCS*. Springer, 2010.
5. A Chlipala. The Bedrock structured programming system: Combining generative metaprogramming and Hoare logic in an extensible program verifier. In *ACM International Conference on Functional Programming (ICFP)*, 2013.
6. M E Conway. Proposal for an UNCOL. *Communications of the ACM*, 1(10):5–8, 1958.
7. M Frigo. A fast Fourier transform compiler. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 1999.
8. J Jensen, N Benton, and A Kennedy. High-level separation logic for low-level code. In *ACM Symposium on Principles of Programming Languages (POPL)*, 2013.
9. A Kennedy, N Benton, J Jensen, and P Dagand. Coq: The world's best macro assembler? In *International Symposium on Principles and Practice of Declarative Programming (PPDP)*, 2013.
10. G Klein, K Elphinstone, G Heiser, J Andronick, D Cock, P Derrin, D Elkaduwe, K Engelhardt, R Kolanski, M Norrish, T Sewell, H Tuch, and Simon Winwood. sel4: Formal verification of an OS kernel. In *22nd ACM Symposium on Operating Systems Principles (SOSP)*, 2009.
11. A Koprowski and H Binsztok. TRX: A formally verified parser interpreter. *Logical Methods in Computer Science*, 7(2), 2011.
12. R Kumar, M O Myreen, S A Owens, and M Norrish. CakeML: A verified implementation of ML. In *ACM Symposium on Principles of Programming Languages (POPL)*, 2014.

13. X Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009.
14. G Malecha, G Morrisett, A Shinnar, and R Wisnesky. Toward a verified relational database management system. In *ACM Symposium on Principles of Programming Languages (POPL)*, 2010.
15. M O Myreen. Verified just-in-time compiler on x86. In *ACM Symposium on Principles of Programming Languages (POPL)*, 2010.
16. M O Myreen and M J C Gordon. Function extraction. *Science of Computer Programming*, 2010.
17. M O Myreen, K Slind, and M J C Gordon. Machine-code verification for multiple architectures - an application of decompilation into logic. In *Formal Methods in Computer-Aided Design (FMCAD)*, 2008.
18. A Nanevski, G Morrisett, A Shinnar, P Govereau, and L Birkedal. Ynot: Reasoning with the awkward squad. In *ACM International Conference on Functional Programming (ICFP)*, 2008.
19. Z Ni and Z Shao. Certified assembly programming with embedded code pointers. In *ACM Symposium on Principles of Programming Languages (POPL)*, 2006.
20. S Sarkar, P Sewell, F Zappa Nardelli, S Owens, T Ridge, T Braibant, M O Myreen, and J Alglave. The semantics of x86-cc multiprocessor machine code. In *ACM Symposium on Principles of Programming Languages (POPL)*, 2009.
21. P-Y Strub, N Swamy, C Fournet, and J Chen. Self-certification: Bootstrapping certified typecheckers in F* with Coq. In *ACM Symposium on Principles of Programming Languages (POPL)*, 2012.
22. D Syme, K Battocchi, K Takeda, D Malayeri, and T Petricek. Themes in information-rich functional programming for internet-scale data sources. In *Workshop on Data Driven Functional Programming (DDFP)*. ACM, 2013.
23. R Wisnesky, G Malecha, and G Morrisett. Certified web services in Ynot. In *International Workshop on Automated Specification and Verification of Web Systems (WWV)*, 2009.