Coq's Vibrant Ecosystem for Verification Engineering (Invited Talk)

Andrew W. Appel
Princeton University
Princeton, NJ, USA
appel@princeton.edu

Abstract

Program verification in the large is not only a matter of mechanizing a program logic to handle the semantics of your programming language. You must reason in the mathematics of your application domain—and there are many application domains, each with their own community of domain experts. So you will need to import mechanized proof theories from many domains, and they must all interoperate. Such an ecosystem is not only a matter of mathematics, it is a matter of software process engineering and social engineering. Coq's ecosystem has been maturing nicely in these senses.

CCS Concepts: • **Software and its engineering** → **Software libraries and repositories**; **Correctness**.

Keywords: None, nil

ACM Reference Format:
Andrew W. Appel. 2022. Coq's Vibrant Ecosystem for Verification Engineering (Invited Talk). In *Proceedings of the 11th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP '22), January 17–18, 2022, Philadelphia, PA, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3497775.3503951

1 Introduction

> "The key design principles in *engineering* a large program verification effort are not the focus of the most well-known publications of the effort. Instead, they can be in less standard references such as . . . invited talks . . . ." [45]

I am interested in the formal verification of software and hardware at scale, of their functional correctness and other important properties. This often requires the verification of many different components, working in application domains that require different kinds of mathematical reasoning, connected together at interfaces.

The logical framework for verification should support this by allowing proofs about different components to be composed, and by allowing proofs about the same component at different levels of abstraction. As a practical matter, that means all the proofs should be conducted in the same implementation of the same logic—exporting nontrivial theorems from one logical framework to another is difficult to sustain at scale [31].

The logic should support *data abstraction* in the conventional software-engineering sense, which is also the conventional mathematical sense [39]. It should permit high-level specifications to be stated formally in a way that we can be reasonably confident that we've proved the right theorem [42]. It should support enough abstraction to clearly express these interfaces between different mathematical application domains, different levels of abstraction, different components, and the interface between human and proof-checker. These desiderata are not novel, and all are reasonably well satisfied by proof assistants based on higher-order logics such as HOL4, Isabelle/HOL, Coq, and others.

Much of this is explained in the survey by Ringer *et al.* [45]. But what happens now that these infrastructures have been in place for several years, and people have been using them?

We begin to see the combination of large and diverse components and tools, embodying very different collections of expertise, whose authors may never have communicated with each other. In this respect, verification follows trends in software engineering: today's engineer combines many preexisting packages, frameworks, and subsystems and layers them into one big useful system. And the same for hardware: SOCs (systems-on-chip) and multicomponent computer systems are put together by architects who have little insight into the internal design of the individual components.

In conventional hardware and software engineering, the components are often rather ill-specified. So it's a miracle that big software and hardware systems work at all, with any degree of reliability or predictability. It ought to be easier for us in the world of specification and proof: if we do everything

CPP '22, January 17–18, 2022, Philadelphia, PA, USA

ACM ISBN 978-1-4503-9182-5/22/01.
https://doi.org/10.1145/3497775.3503951

within a single machine-checked logic, we can get end-to-end guarantees about our large-scale theories, proofs, and tools—even when we have composed opaque components.

2 Two-Layer Program Verification

In software verification one must typically reason about the *program* and its data structures at a low level, using the semantics of the programming language; and about the mathematics of the application domain, which explains why the program accomplishes the user's high-level goals. Each of these kinds of reasoning may be complex in itself, and they may require different kinds of expertise. So it is common engineering practice to separate the proof into layers. Instead of proving all at once that the program implements a high-level specification, write a *functional model* (or "low-level specification" or "algorithm") of the program, and separately prove that the program implements the functional model, and that the functional model accomplishes the high-level goal. This approach was outlined by Heitmeyer *et al.* [24] (although they had no mechanized way of doing the program-implements-low-level-spec proof); and was used in the seL4 verification [29] and in many others.

The tool for proving that the *program* implements the *functional model* is typically a program logic (or an operational-semantic forward-simulation methodology), for a particular programming language, embedded in a general-purpose proof assistant. There are several such tools, some of which I will describe in the next section.

I am particularly interested in tools that are embedded in general-purpose logical frameworks, because they can better *connect* both above and below:

- When the tool for verifying that the program refines a functional model is in the same logic, and framework, as the tool for verifying that the functional model satisfies the high-level specification, then these two proofs can *compose*.
- When the program verification tool is in a logic and framework capable of expressing operational semantics of programming languages, then the tool can be *proved sound*. And if there happens to be a compiler that is proved correct w.r.t. that operational semantics, then the soundness proof of the verification tool can compose with the correctness proof of the compiler.

When these proofs *compose*, ideally they do so seamlessly within a single logic. Otherwise there could be unintended semantic gaps as a theorem statement from one logic (or framework) is interpreted in another logic.

2.1 Example of a Layered Verification

Although it's possible in some program logics to directly prove high-level properties of an imperative program, it's often best to prove that the imperative program *refines* a functional model, and then prove high-level properties of the functional model.

I illustrate by an example: binary search trees. The C program has an insert function that takes the address of structure-fields and uses a loop to traverse the tree from top to bottom [17, Fig. 1]. The high-level specification says that insert and lookup interact in the expected way:

$$\begin{aligned} &\text{lookup}(\text{insert}(t,i,v),i) = v \\ i \neq j \;\rightarrow\; &\text{lookup}(\text{insert}(t,i,v),j) = \text{lookup}(t,j) \end{aligned}$$

We can do this verification in VST, the Verified Software Toolchain. Instead of trying to prove *directly* that the C program has this property, we prove that the C program refines a functional model—that is, binary search trees as a functional program in Coq [4, Chapter "SearchTree"].

```
Inductive tree := E : tree | T: tree → key → V → tree → tree.

Fixpoint insert (x: key) (v: V) (s: tree) : tree :=
 match s with
 | E ⇒ T E x v E
 | T a y v' b ⇒ if x <? y then T (insert x v a) y v' b
                else if y <? x then T a y v' (insert x v b)
                else T a x v b
 end.
```

Then we prove that this functional model satisfies the high-level specification for the interaction of insert and lookup. For this we need a "program logic" for functional programs; but since this program is written in Gallina, which *is* Coq's logic, then Coq *is* the program logic. This proof is quite straightforward in Coq.

Given the lower-level refinement proof [17] and the higher-level properties proof [4, Chapter "SearchTree"], we can compose these in VST using the principle of *function specification subsumption* [10]: that is, any function that satisfies specification A must also satisfy specification B.

And really, I wouldn't want to try verifying this imperative search-tree program in a single layer. In a Hoare logic proof, what loop invariants for insert or lookup would directly express the two equations of the high-level specification?

3 Tools for Verifying Imperative Programs

The lower-level refinement proof could be done in any of several tools. I cannot possibly list here all the good tools, but here are some that form part of the ecosystems of Coq (and, for comparison, of HOL4 and Isabelle/HOL):

Verified Software Toolchain[1] [1] (for C, embedded in Coq) is a higher-order impredicative separation logic. "Separation logic" means it is an extension of Hoare logic that can reason concisely about pointer (anti)aliasing. "Higher-order" means it can reason about higher-order functions (function pointers in C) and about higher-order types (data abstraction). "Impredicative" means it can reason

[1] https://github.com/PrincetonUniversity/VST

even about contravariant patterns—that occur in programs but cannot be expressed using least fixed points. VST is proved sound in Coq using a semantic model with step-indexed logical relations encapsulated in a "later" modality [6]. The soundness proof is w.r.t. the operational semantics of CompCert Clight, for which CompCert is a proved correct optimizing compiler. VST handles almost the entire C language (but not bitfields, gotos, or whole-struct assignments). VST has been used for several verifications by teams that overlap with the VST developers [2, 5, 8, 11, 30, 38, 48], and also has several external users who are applying it to verify proprietary software that has never been seen by the VST developers.[2]

Iris[3] [27] is a library in Coq for the construction of program logics that (like VST) embody higher-order impredicative separation logic, proved sound in Coq using a semantic model with step-indexed logical relations encapsulated in a "later" modality. The Iris model is constructed differently (and more modularly) than that of VST, but has similar properties and goals.

RustBelt[4] [26] (for Rust, embedded in Coq) is a higher-order impredicative concurrent separation logic, proved sound in Coq using Iris. The Rustbelt verification tool handles a subset of the Rust language. It is not yet engineered into a general-purpose Rust verification tool, or in a form easily accessible to external users.

seL4 proofs [29] (for C, in Isabelle/HOL) of correctness of the seL4 microkernel operating system were done by direct forward simulation between a functional program (written in the pure ML-like language embedded in the HOL logic) and the operational semantics of the C program.

Certified Abstraction Layers [21] (for C, embedded in Coq) is a method of direct forward simulation between a functional program (written in the pure ML-like language embedded in Coq's logic) and the operational semantics of the C program. It was used to prove correctness of the CertiKOS hypervisor kernel. This library is not released in a form easily accessible to external users.

CompCert[5] [35] In order to reason formally about the behavior of C programs, one needs a formalization of the semantics of C. Both VST and Certified Abstraction Layers use CompCert's formalization of the Clight operational semantics as their specification. Users of CAL often use CompCert's operational-semantic rules directly; users of VST use them indirectly via the *Verifiable C* program logic.

BedRock's C++ toolchain[6] (for C++, in Coq) is a program logic for C++, implemented atop Iris but with many design features influenced by VST. It does not have a soundness proof with respect to an operational semantics. It was built for the purpose of proving correctness of a commercial hypervisor operating system.[7]

CFML[8] (for OCaml, in Coq). *Characteristic Formulae for ML* [18] is a program logic for OCaml based on separation logic. ML with mutable references and mutable arrays is a mixed functional-imperative language. In CFML, separation logic (anti)aliasing reasoning is needed only for the mutable part while simpler reasoning can be used for the functional part.

Characteristic Formulae for CakeML[9] [22] (for ML, in HOL4) is an implementation of a similar program logic for Standard ML, proved sound w.r.t. the CakeML [33] verified optimizing compiler.

RefinedC[10] [46] (for C, in Coq) is a program logic based on refinement types and separation logic, implemented via Iris, proved sound w.r.t. a C semantics called Caesium.

Several of these tools target the C programming language, because: C is a least-common-denominator *lingua franca* for systems programming. It's (just barely) small enough to have a formalizable semantics. Its pointers and its spicy-Curry-style type system[11] allow the ability to express almost arbitrary data structures and design patterns.

These tools fall into two camps on the question of whether to support data-pointers and function-pointers in their full generality. Both the seL4 verification and Certified Abstraction Layers support a limited subset of C: no function pointers, very limited use of data pointers, no mutually recursive modules. This straightjacket greatly improves the composability and simplicity of the program logic, to the point where (strictly speaking) they don't even have a program logic, they both reason by forward simulations between logical functions and C operational semantics. And within this straightjacket it's possible to implement (and prove) a small kernel operating system.

On the other hand, both the Verified Software Toolchain (VST) and the Iris framework support data pointers, function pointers, and function pointers within data structures, in an attempt to support any wacky design pattern that the C or Rust programmer might use. They both do this through *modal step-indexed separation logic*; that is, separation logic to express (anti)aliasing patterns of data-structure pointers,

[2] I believe, but am not entirely sure, that Company A has verified orbital-satellite operating systems, Company B is verifying 5G and 6G software-defined radio, and Company C is verifying some blockchain thing, and Company D is doing autonomous vehicle operating systems.
[3] https://gitlab.mpi-sws.org/iris/iris/
[4] https://gitlab.mpi-sws.org/iris/lambda-rust/
[5] https://github.com/AbsInt/CompCert
[6] https://github.com/bedrocksystems/BRiCk
[7] Do not confuse BedRock Systems Inc. with the (earlier and continuing) Bedrock project of Adam Chlipala at MIT.
[8] https://gitlab.inria.fr/charguer/cfml2
[9] https://github.com/CakeML/cakeml/tree/master/characteristic
[10] https://gitlab.mpi-sws.org/iris/refinedc
[11] A Curry-style type system means, "a program can have an execution semantics even though it may not type-check;" I'll define a spicy-Curry-style type system to mean, a program can have semantics even though its type system may be unsound.

step-indexed logical relations to express patterns of (mutual) recursion of function pointers (with data pointers), and a modal logic to encapsulate the step-indexing. This adds a thick and fairly opaque layer between the program logic (in which one proves programs correct) and the operational semantics (with respect to which a compiler might be proved correct); this layer is the *model and soundness proof* of the program logic.

These tools are built as large libraries that contain both lemmas (e.g., proofs relating an axiomatic semantics to an operational semantics) and automation (tactic programming or proof-by-reflection programming to assist in applying the program logic to one's programs). That is, one can implement the VST-Floyd [16] separation-logic proof automation system and Iris Proof Mode [32] because Coq has a tactic language. HOL4 and Isabelle/HOL also support proof metaprogramming; Agda and Twelf, not so much.

In contrast, there are several tools that are entirely external to a proof assistant, including Verifast [25], Dafny [34], and Frama-C [28]. Those tools have some advantages: they are more automated, requiring users "only" to provide function preconditions and postconditions, loop invariants, and (often) if-statement postconditions. They achieve this automation via integration with SMT solvers. They are (typically) quick, (typically) thanks to excellent integration with the underlying solvers. But they have some disadvantages: they are not foundational, i.e. proved sound w.r.t. some underlying operational semantics for the programming language (that could be connected to a proved-correct compiler, for example). Their assertion languages (in which preconditions and invariants are written) must be fairly weak, to accommodate the first-order logics of SMT. The user must sometimes "encode" assertions, preconditions, postconditions into this relatively unexpressive language, making them difficult to comprehend.

The weak assertion language limits what can be expressed in a specification of program correctness. That's not *necessarily* a severe problem when proving that a C or Dafny program satisfies a *low-level* spec, but then there is the problem of proving that the low-level spec (written in the limited assertion language) satisfies some high-level property. As the rest of this paper explains, such proofs cannot generally be done in SMT, so it requires an embedding of the low-level specification language in a higher-order-logic proof assistant. Frama-C has taken some steps to support this, for connecting to Coq.

4 Auxiliary Tools for Proving Refinement

When using a modal separation logic such as VST or (one embedded in) Iris, it's sometimes useful to rely on other Coq libraries or packages meant for special-purpose reasoning.

Iris Proof Mode[12] [32] is a tactical proof system for manipulating formulas in modal separation logic. Some of the modes include "persistence" (for quasistatic properties) and "later" [7] (to encapsulate step-indexed approximation); and these interact logically with separating conjunction $*$, ordinary conjunction $\wedge$, and with the Hoare judgment. It's useful to have a tactical system for managing the application of those proof rules. Iris Proof Mode (IPM) was originally developed for Iris, but is now portable to almost any separation logic, and has been used atop VST [37].

VST-Floyd [16] is the proof automation for VST's separation logic for C; it operates by semiautomatic symbolic execution of C programs, where the symbolic "states" are really separation-logic assertions. In contrast to IPM, it has weak support for modes and much stronger support for forward Hoare-logic reasoning and automated entailment solving.

CertiGraph[13] [47] is the theory of directed-graph data structures embedded in separation logic. Separation logic is (normally) tuned to assertions of the form $\{P * Q\}$, in which the addresses of the data structure satisfying P must be disjoint from the addresses satisfying Q. But directed acyclic graphs, or directed cyclic graphs, naturally express *sharing*. To model and prove algorithms such as graph search, shortest path, minimum spanning tree, union-find, and garbage collectors, we need to reason explicitly about directed graphs. But we still want to do this within the context of separation logic, so we can use logics such as VST and Iris to handle other aspects of the program that naturally want separation. CertiGraph has been successfully used atop VST-Floyd to prove the correctness of a generational copying garbage collector [47] and Dijkstra's, Kruskal's, and Prim's algorithms [40].

5 Hardware

The recipe for comprehensively verifying software is: write a high-level specification of the functional properties you demand; write an algorithm / functional model / low-level specification; prove the functional model satisfies the high-level specification; write a *program* in a language with a formalized semantics; prove that the program refines the functional model; compile the program to a low level implementation in *machine language*; prove that the compiler is correct; compose all those proofs.

Perhaps you have chosen to implement something directly in hardware (VLSI or FPGA) rather than in software. The verification layers are analogous: Replace the word *program* with *circuit design*, and replace *machine language* with *logic gates*, and you have the recipe for proving the functional correctness (and other properties) of hardware.

[12] https://gitlab.mpi-sws.org/iris/iris/-/tree/master/iris/proofmode

[13] https://github.com/Salamari/CertiGraph

And you will want a *tool* for proving that the circuit design implements your functional model:

Kôika[14] [15] is a Coq formalization of Bluespec—a Haskell-like functional language for describing digital circuits. The *compiler* is a program in Coq that compiles these Bluespec-like programs to Verilog, a low-level register-transfer-language for describing digital circuits. This compiler is proved correct in Coq, based on semantics of source and target languages.

6 Proving Properties of the Functional Model

In Section 2.1 I wrote, "Coq *is* the program logic [for proving properties of the functional model] so the proof goes smoothly." If only it were so simple! In fact, this proof may be highly nontrivial. The functional model is the *algorithm*—and the correctness of an algorithm can be a complex proof, and may use different kinds of mathematics in different application domains.

Therefore you should use libraries or packages, Coq theories for your domain-specific mathematics—where a "theory" is not just a set of definitions and lemmas but often includes programs (in Gallina or Ltac) and other forms of proof automation.

Here are some of the Coq packages that have been useful in proving (functional models of) C programs correct:

Coq-std++[15] is a library of functional models of data structures such as lists, trees, search tree lookup tables, binary tries, and so on, along with monad notations, an equality simplifier, a solver for compatibility of functions w.r.t. relations, and so on.

Foundational Cryptography Framework [41] is based on a theory of computational monads over probability spaces. It can be used to prove such things as "the attacker who can only do 2^k steps of computation can guess the next bit that your random-number generator will produce with probability $< 0.5 + 2^{-(170-k)}$" [48].

Flocq [13, 14] is a formalization in Coq of the IEEE-754 floating-point standard used in all modern computers. That standard is so good that it earned a Turing award[16] and was first formalized (in HOL Light) in 1999 [23]. In Flocq one can reason about how 32-bit or 64-bit (or any other size) formats represent binary exponent-mantissa scientific notation, and how that notation injects into the real numbers, and how the arithmetic operations work with their rounding modes, and so on.

Flocq is used in CompCert's specification of the floating-point operations of C, and of the assembly languages to which CompCert is targeted [12]. Flocq's floating-point spec is also used in VST's program logic (which of course is based on CompCert's operational semantics of C). And therefore, Flocq is also needed in correctness proofs (of C programs) built by users of VST [5].

Flocq is built in such a modular way that it can represent a wide variety of floating-point formats, and it can also represent fixed-point arithmetic (where one computes using fixed-precision integers in which that are implicitly scaled in a way that the algorithm designer keeps track of). This is quite useful for some engineering applications.

Coq.Reals is the formalization of the classical real numbers in Coq's standard library. Computer programs cannot compute on the (classical) real numbers—only on computable representations such as the integers, rationals, or floating-point.[17] But we usually need to reason in the reals in proving floating-point programs correct. That is, the program computes on floating-point numbers that *represent* real numbers. If a : float represents $R(a) : \mathbb{R}$ and b represents $R(b)$ then when we floating-point multiply $a \times_{\mathrm{F}} b = c$ we get a number c that represents $R(a) \cdot R(b) + \delta$, where δ is the floating-point rounding error. We reason this way, instead of directly in the floating-point numbers, because the mathematical theory of real analysis is well developed and has reasonable properties.

Interval [14, §4.2] is an *interval arithmetic* package for the Real numbers. It is useful for reasoning about bounds for approximation errors that may arise from discretization (finitary sums rather than integration of infinitesimals) and from floating-point error (computing in the rationals rather than the reals). That is, if we know that $\hat{a}$: float approximates some quantity of interest $a : \mathbb{R}$ with accumulated error δ_a, that is, $a - \delta_a \le R(\hat{a}) \le a + \delta_a$ and similarly for $\hat{b}, \hat{c}$, then we can use interval arithmetic to manage inequalities such as $a + b - (\delta_a + \delta_b) \le R(\hat{a}) + R(\hat{b}) \le a + b + (\delta_a + \delta_b)$.

Gappa [14, §4.3.1] is a package for automating the reasoning about floating-point rounding errors. Gappa is implemented in C++ and generates proofs checkable by Coq. In a proof that a C program correctly *and accurately* implements floating-point square root by Newton's method [5], we use Gappa in the layer between the functional model and the high-level specification.

VCFloat[18] [44] is a package layered atop Interval and Flocq that calculates a reified (syntactic) description of a floating-point expression (for example, from a CompCert Clight syntax tree); allows annotation with some expected properties of the floating-point calculation; then calculates what verification conditions would need to be proved in order to bound the floating-point roundoff error of the

[14] https://github.com/mit-plv/koika
[15] https://gitlab.mpi-sws.org/iris/stdpp
[15] https://github.com/adampetcher/fcf
[16] for William Kahan in 1989

[17] Whether it can be practical to compute on the constructive reals is perhaps still a research question.
[18] https://github.com/reservoirlabs/vcfloat

entire expression; then discharges those verification conditions using the Interval package.

VCFloat and Gappa are intended to solve essentially the same problem, but VCFloat is implemented entirely within Coq. We are experimenting[19] with VCFloat in proving accurate a numerical integrator for ordinary differential equations (ODEs).

Mathematical Components[20] [36] formalizes mathematical structures such as ordinals, groups, fields, matrices, polynomials, and some of the useful theorems and algorithms upon them. Many interesting projects have used Mathcomp; in software verification we have used it in a verification of forward erasure correction (FEC) based on Reed-Solomon codes [19]. In this project, the *algorithm* expresses computation over a finite field modulo a primitive polynomial, via matrix multiplication and Gaussian elimination. We prove using VST that the C program correctly implements the algorithm, in a proof that doesn't understand much mathematics at all (and does not use Mathcomp); we prove using Mathcomp that the algorithm correctly reconstructs missing network packets in a proof that doesn't mention a C program (and does not use VST).

Interaction Trees[21] [30] is a library for reasoning about reactive input/output or for representing functional models of recursive and imperative programs, employing a coinductive datatype.

Sail[22] [9] is a language for describing machine languages (instruction-set architectures), that can generate formal descriptions of the syntax and semantics of ISAs in Isabelle, Coq, and other logics. It comes with well-researched formal descriptions of several widely used machines. A Coq proof about a compiler (targeting an ISA), a CPU design (implementing an ISA), a static analyzer, would find it useful to import these as libraries.

See also

Many packages in the Coq ecosystem are described at https://github.com/coq-community/awesome-coq.

7 The Coq Platform

> "Coq has reached the usage size, where the central maintenance of libraries is no longer feasible. Instead, the Coq library has been factored into hundreds of repositories with a somewhat standardized build process. This allows distributed maintenance of the library. But it also means that not every repository always builds with the latest version of Coq. For example, when we ran our export[23] in early 2019, only around 70 of around 250 repositories could be built, including the MathComp libraries (the situation has improved since then)." [31].

The good news about a vibrant ecosystem of substantial libraries developed and maintained by many different people is that one can compose them to engineer large verifications.

The bad news is that one can now enter *DLL Hell*[24], or more generally *Dependency Hell*[25]. This is a situation observed in software builds where, for example, package D depends on packages B and C, which both depend on package A. But B and C depend on *different incompatible versions* of A; or B and C build only in incompatible versions of the language/compiler in which they're programmed.[26]

So, for example, if you need VST version 2.8 that builds in Coq 8.13.2 and depends on CompCert 3.9 (which uses Flocq 3.4), but you also want to reason about floating-point calculations using VCfloat 1.0 that requires Coq 8.5beta2, CompCert 2.6, Flocq 2.5—then you are in Dependency Hell.

Many technical solutions are proposed[27] for detecting and mitigating Dependency Hell, but most of them will not solve the problem in this particular application, *program verification*. The reason is that if you need to reason about your source program in the same theory of floating point upon which the program logic is proved sound, then VST and VCfloat *must use the same version* of Flocq, the specification of floating point.

The solution that will actually work is the **Coq Platform**, an effort led by Michael Soegtrop with the help of Karl Palmskog, Enrico Tassi, Théo Zimmermann, and others. It is much a work of social engineering as it is of software engineering. That is: with each release of Coq, there must also be a "Platform" release of each of the many libraries (such as I have described in this paper) that are compatible with that release of Coq *and with each other*. Then, an end user can simply download and install a given Platform release, with the confidence that all the libraries will be compatible.

This means that (before each Platform release is constructed) the authors or maintainers of each of these libraries

[19] Ariel Kellison and Andrew W. Appel, work in progress.
[20] https://github.com/math-comp
[21] https://github.com/DeepSpec/InteractionTrees
[22] https://github.com/rems-project/sail

[23] The situation that Kohlhase and Rabe describe was transient: in early 2019 a new release of Coq had just come out, and many of the library maintainers had not yet released their new versions, and eventually there were over 450 compatible packages for that release 8.9.0 of Coq. But even so, the Coq Platform has improved the release process; library maintainers test against release-candidate Coq versions in advance of the Coq release, and the Coq Platform release follows within a month or two of the Coq major-version release.
[24] https://en.wikipedia.org/wiki/DLL_Hell
[25] https://en.wikipedia.org/wiki/Dependency_hell
[26] Furthermore, it is not feasible to test version dependency information in opam, so one cannot assume that it is accurate. Sometimes opam builds fail not because something is impossible, but because opam select wrong versions based on inaccurate dependency information.
[27] https://en.wikipedia.org/wiki/Dependency_hell#Solutions

must coordinate with the maintainers of those libraries that are adjacent on the dependency graph. For example, the maintainer of CompCert must coordinate with the maintainer of Flocq, and the maintainer of VST must coordinate with the maintainer of CompCert.

The managers of the Coq Platform coordinate this effort by using tools such as e-mail, Zulip, Github issues, and opam. In addition a significant part of the Platform effort is to determine how best to use tools such as opam to make installation smoother for users.

The Coq Platform has been in existence only since about mid-2020; the third major Platform release is expected in late 2021. It is enormously useful.

8 Community Tools

How can Coq's library ecosystem be healthy even though it is beyond the scale where a single group could possibly maintain it? Of course, Coq is hardly unique in this respect—many open-source software ecosystems are diverse, interoperable, and healthy in this way.

And it is possible, in part, by using tools and processes developed in the open-source community at large:

Open-source itself, as a legal and social concept.

`git`, a 5th-generation source code control system.[28]

Github or Gitlab, which integrate many project-management interfaces atop git, such as access control, issue reports, pull requests, and distribution packaging.

Public repositories: the idea that every commit to your repo should be visible to the public, that every bug you commit, every discussion you have, should be readable by the world—that is revolutionary, compared to a previous ethos in which a "release" was published every year or two.

Continuous integration, and especially the kind of *linked* continuous integration that can immediately notice inter-package incompatibilities.

Continuous delivery, even in the limited sense that the main branch of package *A*'s repo may import the main branch of *B* as a subrepo, meaning that *B* is continuously delivered even though its maintainers did not make a conscious decision to do that, and may not even be aware that *A* exists. And this is enabled by public repositories.

`opam` and other package-management systems.

The first time I shipped an open-source distribution, Standard ML of New Jersey (jointly with David B. MacQueen in 1988), we did so using the BSD license, on 9-track magnetic tapes sent through snail mail. Other than open-source itself, we had none of these 21st-century affordances. That was more than 30 years ago; but most of these affordances have become important in the Coq ecosystem only in the past 10 years, or the past 5 years.

[28] I write as a user, in the past, of SCCS then RCS then CVS then `subversion`.

9 Difficulties

Building large systems in Coq has its difficulties. Here I will describe a few.

9.1 Division of Labor

Building verified software requires many different kinds of skill and expertise: Software engineering (writing efficient C programs), specification engineering (functional models, high-level specs), C language verification (such as the use of VST), mathematics of the application domain (proving properties of the functional model). In some real verifications that we (and others) have done, these five jobs are done by five different engineers [3]. There is no need to find a single genius who can do everything.

9.2 Proof Dialects

Suppose you say to yourself, "Come, let us build a program and its correctness proof, that reaches unto the heavens, so that we may make a name for ourselves." You hire many talented proof engineers. But some of those engineers write their proofs in Ltac, others in Ltac2, and others in ssreflect. After your engineers scatter over the face of the Earth, you try to maintain the resulting artifact. But you are fluent in just one of these tactic languages; and therefore you find the task very difficult.

The existences of several tactic languages is not a *bug;* it is a *feature* that must be managed just like any other engineering technology. For example, my students verified a program that does erasure correction by matrix multiplication over a finite field modulo a primitive polynomial [19]. To prove that the C program correctly implemented the functional model they used VST. To prove that the functional model correctly decoded in the finite field, they used Mathematical Components. VST uses the Ltac tactic language and uses first-order data structures to represent matrices: lists of lists. MathComp uses the ssreflect tactic language and uses dependently typed data structures to represent matrices. It would be extremely difficult to mix ssreflect and dependently typed matrices into a VST proof; and it would be equally difficult to perform a MathComp proof with Ltac and nondependent types.

The solution was to write *two layers of functional model.* The lower layer expresses Gaussian elimination as a Coq function on lists of lists; the upper layer expresses Gaussian elimination as a Coq function on dependently typed matrices. One proves the C program correct with respect to the lower-level functional model; the other proves the mathematical correctness of the upper-level functional model; and it's a straightforward proof to relate the two functional models. Ltac lives in one land, ssreflect lives in another, and they trade peaceably with each other at arm's length.

You might think, "you still need two engineers, speaking respectively Ltac and ssreflect, to maintain the two parts of

```
Fixpoint big (n: nat) : is_true true :=
 match n with
 | O ⇒ eq_refl
 | S n' ⇒ eq_trans (big n') (big n')
 end.

Definition t := {x: bool | is_true x}.

Definition a (n: nat) : t := exist is_true true (big n).

Time Compute (a 10). (* 0 secs *)
Time Compute (a 20). (* 0.061 secs *)
Time Compute (a 21). (* 0.144 secs *)
Time Compute (a 22). (* 0.279 secs *)
Time Compute (a 23). (* 0.565 secs *)
Time Compute (a 24). (* 1.189 secs *)
Time Compute (a 25). (* 2.115 secs *)
Time Compute (a 26). (* 4.917 secs *)
```

Figure 1. Proof blowup in a dependently typed computation.

the system." But this was already the case. Just let it be that one engineer can maintain the VST proof of the C program, without having to understand MathComp, finite fields, or ssreflect. And let it be that the other engineer can maintain the MathComp proof of the properties of the functional model, without having to understand C programming or VST proofs. "You shall not wear a garment of different sorts, such as wool and linen mixed together."

9.3 Computation in Coq

One can prove a program correct in Coq, extract it to OCaml, and run it. But there are also reasons to run computations *inside* Coq. For example, VST does (partly computational) symbolic execution of C programs, but within the terms on which it computes are Coq values that describe what the C program represents. The types of these values are chosen by the user (as part of the program's specification); the expressions constituting these values are used in the user's interactive functional-correctness proofs. So there is a *mix* of computation and interactive proof, and this could not possibly be done in extraction to OCaml.

Coq can do efficient computation of its native lambda-terms in CiC, using the the tactics compute (call-by-value reduction, reasonably fast), vm_compute (compilation to byte code then interpretation, faster) and native_compute (compilation to native code, fastest). But we run into difficulties when some of the subterms are not meant to be computed or reduced. For example, we want to simplify all the terms having to do with symbolic execution—the semantics of C—but not the user's subterms (within the symbolic execution). If we compute the user's application-specific subterms, that makes proof management difficult for the user (and it can sometimes blow up into huge normal forms).

There are ways to accomplish this in Coq, but they are a bit clumsy, nonmodular, and difficult to make reliable. For example, Coq's cbv delta tactic has a way to specify just which terms to expand, but it's not very modular.

9.4 Computations on Terms with Proofs

Terms with subterms that are proofs can cause difficulty when computing in Coq. Coq's logic permits, for example, packaging a value with a proof of a property of that value: $\{x : T \mid P(x)\}$. The big libraries I describe in Sections 3, 4, and 6 sometimes build such packages, and it leads to trouble with computation in the clients of those libraries.

We have observed this behavior calculating on CompCert's structure-field type definitions, which contain internal proofs about field-alignment; and when computing on Flocq's representation of floating-point numbers, which contain internal proofs about exponent bounds.

Consider the example in Figure 1: The function big(n) produces a proof (of is_true true) of size 2^n. The function a(n) produces a package of the boolean b and a proof that b is true. The size of the term a(n) is $n + 2$: the name a applied to the natural number n. But when we compute, the term blows up to size 2^n.

You might think, "no one would write such silly, deliberately huge proofs." No, not deliberately. What happens is that we have, in CompCert, a structure field with a proof that its offset is aligned; or in Flocq, a floating point number with a proof that its mantissa is in bounds. For example, $5 < 2^{23}$. The shortest proof of that is tiny. But in practice (in CompCert and Flocq) those proofs are done using decision procedures (such as lia for linear integer arithmetic) based on other terms that 5 and 23 are calculated from, and the bound-proofs of those terms. So we do indeed get huge proofs of trivial facts.

And why should we not? The whole point of proving a lemma is to *encapsulate* a mathematical fact, so that you don't have to worry about *how* it was proved. But abstraction can sometimes be the enemy of efficiency. Proofs are proofs, programs are programs, and don't try to use proofs as programs! You may have heard of the Curry-Howard correspondence, but if you want to take it seriously and compute with proofs, then you'll have to optimize all your proofs the same way you optimize your programs, and nobody wants to do that.

Is there a way to achieve true proof irrelevance in the Coq kernel, such that terms that are proofs of propositions (of type Prop) are truly ignored? There may be type-theoretic difficulties here, since we also have terms that pattern-match on proofs of equality and reduce only when those proofs calculate all the way down to eq_refl.

Indeed, this is a research question that's the subject of intense study. Gilbert *et al.* [20] demonstrated a slightly weaker

variant of Prop which is truly computationally irrelevant, but loses the ability to pattern-match on equalities soundly. Pujet and Tabareau [43] describe a type theory that fixes that problem along with many others.

Until these type-theoretic innovations can be integrated into Coq, I suggest that a less radical solution would be to implement a reduction strategy similar to call-by-value but that does not step into terms of type proof-of-Prop.

In the present, my advice to users is: **avoid putting proofs into terms** on which *someone* might wish to compute. It's not enough to avoid putting proofs into terms on which *you* intend to compute! Recall that we have difficulties when we are clients of CompCert and of Flocq, with proof terms inside computational objects built by those systems. The authors of CompCert and Flocq did not intend to compute on these terms. But *we* want to do so.

10 Conclusion

Coq has a vibrant ecosystem of libraries and packages thanks to the many researchers who take the trouble to publish *software* in addition to *papers*, and who take the trouble to maintain that software, and to the many people who help maintain the infrastructure of the ecosystem itself, and the instituts and corporations who employ those people. This builds on itself in a virtuous cycle: with libraries and tools, you can do your research, and build new libraries and tools. And this permits projects of a scale that no single institution, or no single organized research group, could support.

It is not only the number of lines of code; it is the huge variety of expertise embodied and encapsulated within these library packages. One can compose these packages without needing to be expert in how they all work internally. And that is especially true in the world of machine-checked proof, where one can be sure that if theorem A is proved with reliance on theorem B, there has been no misunderstanding about the definitions.

So the individual researcher and tool builder can leverage the ecosystem by finding and using the right tools and packages for the job, allowing software verification at a scale not otherwise possible. Just don't forget to do it in a public repo with continuous integration, and if you build something good, then try to participate in the Platform.

References

[1] Andrew W. Appel. 2011. Verified Software Toolchain. In *ESOP'11: European Symposium on Programming*, Gilles Barthe (Ed.). LNCS, Vol. 6602. Springer, 1–17.

[2] Andrew W. Appel. 2015. Verification of a Cryptographic Primitive: SHA-256. *ACM Trans. on Programming Languages and Systems* 37, 2 (April 2015), 7:1–7:31.

[3] Andrew W. Appel. 2016. Modular Verification for Computer Security. In *CSF 2016: 29th IEEE Computer Security Foundations Symposium*. 1–8.

[4] Andrew W. Appel. 2107. *Verified Functional Algorithms*. Software Foundations, Vol. 3. softwarefoundations.org.

[5] Andrew W. Appel and Yves Bertot. 2020. C-language floating-point proofs layered with VST and Flocq. *Journal of Formalized Reasoning* 13, 1 (Dec. 2020), 1–16. https://doi.org/10.6092/issn.1972-5787/11442

[6] Andrew W. Appel, Robert Dockins, Aquinas Hobor, Lennart Beringer, Josiah Dodds, Gordon Stewart, Sandrine Blazy, and Xavier Leroy. 2014. *Program Logics for Certified Compilers*. Cambridge.

[7] Andrew W. Appel, Paul-Andre Melliès, Christopher D. Richards, and Jerôme Vouillon. 2007. A Very Modal Model of a Modern, Major, General Type System. In *Proc. 34th Annual Symposium on Principles of Programming Languages (POPL'07)*. 109–122.

[8] Andrew W. Appel and David A. Naumann. 2020. Verified Sequential Malloc/Free. In *International Symposium on Memory Management (ISMM)*. 48–59.

[9] Alasdair Armstrong, Thomas Bauereiss, Brian Campbell, Alastair Reid, Kathryn E. Gray, Robert M. Norton, Prashanth Mundkur, Mark Wassell, Jon French, Christopher Pulte, Shaked Flur, Ian Stark, Neel Krishnaswami, and Peter Sewell. 2019. ISA Semantics for ARMv8-a, RISC-v, and CHERI-MIPS. *Proc. ACM Program. Lang.* 3, POPL, Article 71 (Jan. 2019), 31 pages. https://doi.org/10.1145/3290384

[10] Lennart Beringer and Andrew W. Appel. 2021. Abstraction and Subsumption in Modular Verification of C Programs. *Formal Methods in System Design* (2021). https://doi.org/10.1007/s10703-020-00353-1

[11] Lennart Beringer, Adam Petcher, Katherine Q. Ye, and Andrew W. Appel. 2015. Verified Correctness and Security of OpenSSL HMAC. In *24th USENIX Security Symposium*. USENIX Assocation, 207–221.

[12] Sylvie Boldo, Jacques-Henri Jourdan, Xavier Leroy, and Guillaume Melquiond. 2013. A formally-verified C compiler supporting floating-point arithmetic. In *2013 IEEE 21st Symposium on Computer Arithmetic*. IEEE, 107–115.

[13] Sylvie Boldo and Guillaume Melquiond. 2011. Flocq: A unified library for proving floating-point algorithms in Coq. In *2011 IEEE 20th Symposium on Computer Arithmetic*. IEEE, 243–252.

[14] Sylvie Boldo and Guillaume Melquiond. 2017. *Computer Arithmetic and Formal Proofs: Verifying Floating-point Algorithms with the Coq System*. Elsevier.

[15] Thomas Bourgeat, Clément Pit-Claudel, and Adam Chlipala. 2020. The essence of Bluespec: a core language for rule-based hardware design. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. 243–257.

[16] Qinxiang Cao, Lennart Beringer, Samuel Gruetter, Josiah Dodds, and Andrew W. Appel. 2018. VST-Floyd: A Separation Logic Tool to Verify Correctness of C Programs. *J. Autom. Reason.* 61, 1-4 (June 2018), 367–422. https://doi.org/10.1007/s10817-018-9457-5

[17] Qinxiang Cao, Shengyi Wang, Aquinas Hobor, and Andrew W. Appel. 2018. Proof Pearl: Magic Wand as Frame. https://www.cs.princeton.edu/~appel/papers/wand-frame.pdf

[18] Arthur Charguéraud. 2011. Characteristic Formulae for the Verification of Imperative Programs. In *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming* (Tokyo, Japan) *(ICFP '11)*. Association for Computing Machinery, New York, NY, USA, 418–430. https://doi.org/10.1145/2034773.2034828

[19] Joshua M. Cohen, Qinshi Wang, and Andrew W. Appel. 2021. Verified Forward Erasure Correction in Coq. in preparation.

[20] Gaëtan Gilbert, Jesper Cockx, Matthieu Sozeau, and Nicolas Tabareau. 2019. Definitional Proof-Irrelevance without K. *Proc. ACM Program. Lang.* 3, POPL, Article 3 (jan 2019), 28 pages. https://doi.org/10.1145/3290316

[21] Ronghui Gu, Jérémie Koenig, Tahina Ramananandro, Zhong Shao, Xiongnan (Newman) Wu, Shu-Chun Weng, Haozhong Zhang, and Yu Guo. 2015. Deep Specifications and Certified Abstraction Layers. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015* (Mumbai, India, January 15-17, 2015), Sriram K. Rajamani and David Walker (Eds.). ACM, 595–608. https://doi.org/10.1145/2676726.2676975

[22] Armaël Guéneau, Magnus O. Myreen, Ramana Kumar, and Michael Norrish. 2017. Verified Characteristic Formulae for CakeML. In *Programming Languages and Systems: 26th European Symposium on Programming, ESOP 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22–29, 2017, Proceedings*. Springer Berlin Heidelberg, Berlin, Heidelberg, 584–610. https://doi.org/10.1007/978-3-662-54434-1_22

[23] John Harrison. 1999. A Machine-Checked Theory of Floating Point Arithmetic. In *Theorem Proving in Higher Order Logics: 12th International Conference, TPHOLs'99 (LNCS)*, Vol. 1690. Springer-Verlag, 113–130.

[24] Constance Heitmeyer, Myla Archer, Elizabeth Leonard, and John McLean. 2008. Applying formal methods to a certifiably secure software system. *IEEE Transactions on Software Engineering* 34, 1 (2008), 82–98.

[25] Bart Jacobs, Jan Smans, Pieter Philippaerts, Frédéric Vogels, Willem Penninckx, and Frank Piessens. 2011. VeriFast: A powerful, sound, predictable, fast verifier for C and Java. In *NASA Formal Methods Symposium*. Springer, 41–55.

[26] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. 2017. RustBelt: Securing the foundations of the Rust programming language. *Proceedings of the ACM on Programming Languages* 2, POPL (2017), 1–34.

[27] Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Aleš Bizjak, Lars Birkedal, and Derek Dreyer. 2018. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *Journal of Functional Programming* 28 (2018).

[28] Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. 2015. Frama-C: A software analysis perspective. *Formal Aspects of Computing* 27, 3 (01 May 2015), 573–609. https://doi.org/10.1007/s00165-014-0326-7

[29] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, et al. 2009. seL4: Formal verification of an OS kernel. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*. ACM, 207–220.

[30] Nicolas Koh, Yao Li, Yishuai Li, Li-yao Xia, Lennart Beringer, Wolf Honoré, William Mansky, Benjamin C Pierce, and Steve Zdancewic. 2019. From C to interaction trees: specifying, verifying, and testing a networked server. In *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs*. 234–248.

[31] Michael Kohlhase and Florian Rabe. 2021. Experiences from Exporting Major Proof Assistant Libraries. *J. Automated Reasoning* 65 (Aug. 2021), 1265–1298. https://doi.org/10.1007/s10817-021-09604-0

[32] Robbert Krebbers, Jacques-Henri Jourdan, Ralf Jung, Joseph Tassarotti, Jan-Oliver Kaiser, Amin Timany, Arthur Charguéraud, and Derek Dreyer. 2018. MoSeL: A General, Extensible Modal Framework for Interactive Proofs in Separation Logic. *Proc. ACM Program. Lang.* 2, ICFP, Article 77 (July 2018), 30 pages. https://doi.org/10.1145/3236772

[33] Ramana Kumar, Magnus O Myreen, Michael Norrish, and Scott Owens. 2014. CakeML: a verified implementation of ML. *ACM SIGPLAN Notices* 49, 1 (2014), 179–191.

[34] K. Rustan M. Leino. 2010. Dafny: An Automatic Program Verifier for Functional Correctness. In *Logic for Programming, Artificial Intelligence, and Reasoning - 16th International Conference, LPAR-16, Dakar, Senegal, April 25-May 1, 2010, Revised Selected Papers (LNCS 6355)*, Edmund M. Clarke and Andrei Voronkov (Eds.). Springer, 348–370. https://doi.org/10.1007/978-3-642-17511-4_20

[35] Xavier Leroy. 2009. Formal verification of a realistic compiler. *Commun. ACM* 52, 7 (2009), 107–115.

[36] Assia Mahboubi and Enrico Tassi. 2021. *Mathematical Components*. Zenodo. https://doi.org/10.5281/zenodo.4457887

[37] William Mansky. 2022. Bringing Iris into the Verified Software Toolchain. (2022). (in preparation).

[38] William Mansky, Andrew W. Appel, and Aleksey Nogin. 2017. A Verified Messaging System. In *Proceedings of the 2017 ACM International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA '17)*. ACM.

[39] John C. Mitchell and Gordon D. Plotkin. 1988. Abstract Types Have Existential Type. *ACM Trans. on Programming Languages and Systems* 10, 3 (July 1988), 470–502.

[40] Anshuman Mohan, Wei Xiang Leow, and Aquinas Hobor. 2021. Functional Correctness of C Implementations of Dijkstra's, Kruskal's, and Prim's Algorithms. In *International Conference on Computer Aided Verification*. Springer, 801–826.

[41] Adam Petcher and Greg Morrisett. 2015. The Foundational Cryptography Framework. In *Principles of Security and Trust (POST'15) (LNCS)*, Vol. 9036. Springer, 53–72.

[42] Robert Pollack. 1998. How to Believe a Machine-Checked Proof. In *Twenty Five Years of Constructive Type Theory*, G. Sambin and J. Smith (Eds.). Oxford University Press.

[43] Loïc Pujet and Nicolas Tabareau. 2022. Observational Equality: Now For Good. *Proc. ACM Program. Lang.* 5, POPL, Article (to appear) (jan 2022).

[44] Tahina Ramananandro, Paul Mountcastle, Benoît Meister, and Richard Lethin. 2016. A Unified Coq Framework for Verifying C Programs with Floating-Point Computations. In *Proceedings of the 5th ACM SIGPLAN Conference on Certified Programs and Proofs (CPP 2016)*. Association for Computing Machinery, New York, NY, USA, 15–26. https://doi.org/10.1145/2854065.2854066

[45] Talia Ringer, Karl Palmskog, Ilya Sergey, Milos Gligoric, and Zachary Tatlock. 2019. QED at Large: A Survey of Engineering of Formally Verified Software. *Foundations and Trends in Programming Languages* 5, 2-3 (2019), 102–281. https://doi.org/10.1561/2500000045

[46] Michael Sammler, Rodolphe Lepigre, Robbert Krebbers, Kayvan Memarian, Derek Dreyer, and Deepak Garg. 2021. RefinedC: Automating the Foundational Verification of C Code with Refined Ownership Types. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI)*. Association for Computing Machinery, 158–174. https://doi.org/10.1145/3453483.3454036

[47] Shengyi Wang, Qinxiang Cao, Anshuman Mohan, and Aquinas Hobor. 2019. Certifying Graph-Manipulating C Programs via Localizations within Data Structures. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 171 (Oct. 2019), 30 pages. https://doi.org/10.1145/3360597

[48] Katherine Q. Ye, Matthew Green, Naphat Sanguansin, Lennart Beringer, Adam Petcher, and Andrew W. Appel. 2017. Verified Correctness and Security of mbedTLS HMAC-DRBG. In *ACM Conference on Computer and Communications Security (CCS'17)*. ACM, 2007–2020. https://doi.org/10.1145/3133956.3133974