# Proof-assistant-based verification of programs

Valentin Robert
University of California, San Diego
vrobert@cs.ucsd.edu

## Abstract

Proof assistants are now widely used in the field of programming languages research to reason formally about objects of increasing complexity, like advanced type systems, large programming languages, compilers, or mathematical constructs. This research examination report focuses on the use of proof assistants in reasoning about programs, and surveys the methods developed to tame the complexity of representing languages and their semantics, as well as the difficulties encountered in reasoning about programs in imperative and concurrent settings. We also examine concrete tools developed to perform this reasoning, and consider their trade-offs in expressivity and proof automation.

## 1. Introduction

Over the past decades, proof assistants have changed the social process of exchanging formal results about programming languages, and has opened the way, along with solver-based techniques, to tackling challenges much more realistic than what was usually done using pen and paper. Interactive theorem provers can be used to formalize existing bodies of mathematics, to develop new mathematical theories in a formal setting, to reason about type systems, or to implement verified programs. It is now common to see formal developments attached to publications in the field of programming languages, where the proofs used to be detailed informally, often in an informal combination of mathematical notations and natural language.

In this report, we focus on the use of proof assistants in the verification of programs, and highlight some of the main directions of research in this area, and for each of those, some of the most succesful or promising techniques.

Section 2 focuses on the use of proof assistants to implement and verify functional programs. In section 3, we describe the meta-theoretic tools developed by the community to reason about programming languages. In section 4, we give an overview of several verified compilers for different flavors of languages. Section 5 focuses on separation logic, a logic designed to reason about imperative programs. In section 6, we describe the most recent developments in verifying low-level code. Finally, in section 7, we address a cross-cutting concern in all of these directions, the challenge of automating the proof effort.

## 2. Functional programs

Modern proof assistants are often built on top of a higher-order dependent type theory. By leveraging the Curry-Howard isomorphism, a mathematical correspondence between programs and proofs, types and propositions, they allow their user to build higher-order functional programs and to prove theorems using the exact same mechanisms, essentially removing the gap between the programming and the logical fragments.

In a system with dependent types, the return type of a function can refer to the value of its argument, and thus express properties about this value. This allows the user of these tools to write functions and verify strong specifications, by following one of two approaches.

The *intrinsic* approach uses dependent types as a programming tool which lets one write certified computations. One can use inductive type families to describe correct-by-construction data types with strong invariants, or use dependent records to package simpler data types with proof terms that characterize such invariants. A classic example of the former is the vector inductive family, indexed by its size:

```
Inductive Vec (T : Type) : nat -> Type :=
| Nil : Vec 0
| Cons : ∀ {n}, T -> Vec n -> Vec (S n)
.
```

By construction, an element of the type `Vec T n` must contain exactly `n` elements. An example of the latter could be a type for a particular tree structure (balanced, red-black, etc.) represented as a dependent record containing a raw tree and a proof of the expected property:

```
Record Type BST (T : Type) := {
  t : BinaryTree T,
  _ : IsOrdered t,
}.
```

Consumers of this data type have access to both the underlying tree `t`, and a proof that `t` is a search tree. Using this proof, they can derive more facts about the ordering of the values in `t` if necessary, without requiring dynamic tests. Producers of this data type, on the other hand, have additional work, since they need not only build `t`, but also construct the proof term for the `IsOrdered` property. This construction can be done either in the form of programming, or, in some systems, it can be dispatched on the side, either fully-automatically with solvers, or manually with tactics. In pratice, this approach can become extremely verbose, because dependently-typed elimination can require complex user annotations for the return types and complex propagation patterns for the facts that are known to be true (like the *convoy pattern*). Systems like Agda mitigate this hurdle by using stronger axioms and performing some automated propagation of equalities when performing elimination on inductive type families.

On the other hand, the *extrinsic* approach separates the programming and reasoning phases. Extrinsic code is written in the higher-order polymorphic part of the calculus, and properties of the intended behavior are captured a

posteriori, in theorems that make up the specification of the code. The benefits here is that the code is much easier to read and write, and clearly separated from the proofs. The inconvenience is that the code does not benefit from the static guarantees that the intrinsic style captures, and therefore, needs to often deal with cases that cannot happen in actual executions.

Most proof assistants require intrinsic termination annotations even for extrinsic code, as non-terminating computations can make the logical fragment inconsistent. Casinghino et al. [7] propose a more flexible phase distinction that lets one write non-terminating functional programs in the programming fragment, and allows transportation to and from the logical fragment whenever it is safe to do so. To achieve this, they index their typing judgement by the fragment in which it holds. When one obtains a derivation in the logical fragment, they may use the term either as a proof or as a program (following the Curry-Howard isomorphism). On the other hand, a derivation in the programming fragment only allows running the program, not using it as a proof. As a result, they may allow more constructs in the programming fragment than the safe ones available in the logical one, but they let the languages overlap so that safe programs are mobile across fragments.

A problem somewhat related to that of termination analysis is that of dealing with infinite-size data and computation over it. A functional programmer may wish to describe infinite constructions, either by virtue of their circularity or by expliciting a generating function that could be expanded as much as needed. For instance, one may wish to describe circular graphs, or to define a possibly-infinite sequence of elements. These are meant either to be consumed only to a finite depth, using mechanisms to delay the evaluation, or to be processed ad infinitum by a recursive process. For the same consistency arguments, these patterns are not expressible safely within an inductive dependently-typed theory. Coinduction can be used to implement functions working over infinite data, as long as there is a guarantee that the function will produce an output within a finite number of execution steps. Furthermore, the system ensures this by an over-conservative, mostly syntactic check. This is good enough for most purposes, but some seemingly-reasonable programs cannot be written (for instance, a stream filter, since it could potentially never output anything).

## 3. Meta-theory

When one wishes to prove meta-theoretical results about some language within a proof assistant, they need to encode this language of discourse (we will refer to it as the *object language*) within the language of reasoning (the *meta-language*). The first challenge one encounters in doing so is the choice of representation for the binders of the object language. The POPLmark challenge [2] recognizes this as a central challenge of mechanized meta-theory. The problem is challenging because of the expectations we have for our abstract syntax. Harper et al. [19] define the desired property as *adequacy*, describing a representation that is *full* (does not contain entities out of the domain of discourse) and *faithful* (uniquely represents the entities we care for).

Early work in the field used first-order approaches. One such approach is a simple named representation, to be considered up to $\alpha$-conversion (renaming of free variables). It is convenient to read and write, but requires a lot of capture-avoiding substitutions in reasoning formally. Some proof assistants implement a nominal logic, within which these considerations are internalized (for instance, Twelf), but most don't. An opposite approach is that of de Bruijn indices, which encode binders as their distance to their abstraction location. This representation is immediately canonical, and avoids the need for substitutions altogether, but it cannot represent terms with free variables without further quotienting, and manipulation of such terms becomes cluttered with shifting binders.

An approach introduced by Coquand [13] and later reused by McKinna et al. [28][29] consists in separating syntactically bound and free occurences of variables. This makes substitutions for free variables simpler since names cannot be captured. This idea is further refined by Leroy [27] and Charguéraud [8] with the *locally nameless* approach. In this setting, bound variables are nameless, that is, they are represented by their de Bruijn index, but free variables are named. This has two advantages: $\alpha$-equivalence is syntactic equality, and free variables can have meaningful names.

In Engineering Formal Meta-theory [1], the authors address another hurdle of formal reasoning with names in the locally nameless approach. The issue is that the classic typing rule for abstraction:

$$\frac{x \notin FV(t) \qquad E, x : S \vdash t^x : T}{E \vdash \texttt{Lam}\, t : S \to T}\ \textsc{Exists-fresh}$$

gives rise to an arguably poor induction principle ($t^x$ stands for the body of the abstraction where de Bruijn variables referring to the opened `Lam` are replaced with `FreeVar` $x$). Upon introduction, this rule is easy to use as we need only come up with a given $x$ that is sufficiently fresh with respect to the body of the abstraction. The issue is that upon elimination, we obtain again a fixed variable name $x$, which is only guaranteed to be fresh with respect to the body of the abstraction. Often, we need to rather pick an $x$ which is fresh with respect to both the body of the abstraction, and some other set of variable names, say $S$. This should be achievable because we can arbitrarily rename variables according to the Barendregt convention (that is, in an $\alpha$-equivalent way), but this presentation forces the user to perform this substitution and prove its correctness. They propose this alternative rule:

$$\frac{\forall x \notin L.\, E, x : S \vdash t^x : T}{E \vdash \texttt{Lam}\, t : S \to T}\ \textsc{Co-finite}$$

This makes the introduction slightly more cumbersome, as one now needs to witness a set of bad names $L$ and prove that all other names are usable. However, the elimination is now much stronger, as we get to pick any name that is not in $L$, and in particular, we can further restrict $x$ to not be in the previously-mentioned set $S$. They show that this presentation is equivalent to the previous one, that it is definable in a proof-assistant, and that it facilitates some classic meta-theoretic proofs.

Yet another, old approach [12], which resurfaced more recently [37] is *higher-order abstract syntax* (HOAS). In essence, it leverages the binders of the meta-language as the binding mechanism for the object language. This gives $\alpha$-conversion and capture-avoiding substitution for free, or rather, for the cost of having implemented them for the meta-language. Unfortunately, the naïve encoding is not definable as an inductive type, as it contains a negative recursive occurence for the binder. The first step towards recovering a higher-order abstract syntax consists in replacing that negative occurence with a different type. Weak HOAS

| | $\lambda x.\lambda y.y\,x$ | $\lambda x.z\,x$ |
|---|---|---|
| Nominal | `Lam "x" (Lam "y" (App (Var "y") (Var "x")))` | `Lam "x" (App (Var "z") (Var "x"))` |
| de Bruijn | `Lam (Lam (App (Var 1) (Var 2)))` | Not representable without an environment |
| Locally nameless | `Lam (Lam (App (Var 1) (Var 2)))` | `Lam (App (FreeVar "z") (Var 1))` |
| HOAS | `Lam (λx. Lam (λy. App (Var x) (Var y)))` | `Lam (λx. App (FreeVar "z") (Var x))` |

**Figure 1.** Representation of a closed term and an open term in different binder approaches. In the HOAS line, $\lambda$ represents the binding construct of the meta-language.

replaces the negative occurence with an abstract data type. This prevents *exotic* terms, that is, terms that inspect or alter their binder, since the binder representation is abstracted over, but it also prevents writing some useful functions because abstractions cannot be crossed. A more satisfactory approach, *parametric higher-order abstract syntax* (PHOAS) [42][10], is to replace the negative occurence with a universally quantified type. This eliminates exotic terms by virtue of parametricity, but also allows arbitrary instantiation of the binder type, which allows to write computations that need to analyze terms under abstractions.

Figure 1 gives a flavor of how an open term and a closed term could be encoded in these different approaches. The POPLmark challenge recognized no clear winner among those ways to address binders, but some techniques like PHOAS were understood after the contest ended and seem promising.

Another interesting challenge in doing meta-theory in a proof assistant is that of reusability and compositionality. Most meta-theoretic efforts are done from scratch or patching together existing code and proofs. The Meta-Theory à la Carte [15] paper adresses exactly this issue. In order to achieve composability, the features of a language must be represented as data, namely, as functors, which can be composed algebraically. Algebras can then be defined as methods of a type class, and properly lifted to composite functors, thus providing composition. The paper presents the proper Mendler-style Church encodings necessary in order to avoid unsafe recursion (Church) and to not relax the order of evaluation (Mendler). They showcase the resulting framework by modularly implementing a mini-ML language, compounded from a language of arithmetic expressions, a language of boolean expressions, a lambda-calculus, a fixpoint-combinator, and an elimination for natural numbers.

## 4. Certified compilation

A spearhead of the formal verification movement has been its application to the mechanizing of program semantics in order to build verified meta-programs. Of particular interest, compilers and program optimizers are artefacts that involve dealing with the semantics of one or several languages, and often prove general statements about all executions of programs before and after transformation. Such features are notably hard to get correct in the first place, and possibly even more complex to maintain when the software evolves to support richer features of the source language or to perform more aggressive optimizations.

The concern for the presence of bugs, as well as the confidence in proof environments as a tool to tame this complexity, is not theoretical. Two studies have compared mainstream C compilers and a verified C compiler. In both cases, hundreds of bugs were found in the unverified compilers, and none in their verified counterpart. In [44], Yang et al. developed Csmith, a random test generation tool, and attempted to discover errors in open source C compilers. Despite spending six CPU-years on fuzz-testing CompCert, they did not find any bug in the verified compiler. In [25], Le et al. reach the same conclusion after applying their equivalence modulo inputs technique to the validation of optimizing compilers by differential testing.

There are two approaches to the development of verified compilation or optimization steps. One can attempt to implement a pass in full formal detail, within their proof assistant, and thus obtain a verified implementation of it. This requires some effort, especially in constructive proof assistants where termination arguments must be explicitly provided and cyclic data structures cannot be explicitly represented. When this proves too difficult, a simpler approach consists in writing the algorithm in an unverified language, and have it output a certificate that can be formally checked. This allows the analysis to be performed in a Turing-complete language, while preserving the soundness of the tool. The price to pay is that of completeness: there is no guarantee that the unproven tool will terminate or that the certificate produced will be accepted by the validator.

Spearheading the effort of developing verified compilers, the CompCert C compiler formalizes a large subset of the C language, and proves a transformation all the way from the parsing of the C syntax to the production of assembly language in the assembly language of a couple of different target architectures. The first version [26] of CompCert was a modest but fully-functional proof of concept. The correctness theorem stated that if the source program terminates, then the generated assembly program terminates and produces the same value. The high-level languages semantics were expressed using big-step semantics, and the lower-level ones using a mix of small-step instruction-level semantics and big-step function-call semantics. The proofs were done by forward simulation of the big-step evaluation.

In order to account for observable side-effects and tail-calls, the next iteration added traces to the high-level languages, and used labeled transition systems for the low-level passes. The enriched correctness theorem conserves the trace of observable effects for terminating programs, that is, it performs the same sequence of system calls and read-write operations to volatile global variables.

The next challenge addressed was non-termination. The authors switched to a coinductive interpretation of the big-step semantics, since the inductive interpretation fails to capture non-termination. Most recently, they added support for unstructured control and partially-unspecified evaluation orders, by moving away from big-step semantics, instead relying on small-step semantics with explicit continuation passing.

Most of the compiler architecture was implemented and proved formally. Two parts are still validated a posteriori: the LR(1) parser of the C syntax into the first intermediate

language CompCert C, and the register allocator which uses a graph coloring technique called iterated register coalescing.

In a parallel line of work, several groups have attempted to develop verified compilers and runtime systems for functional languages. In [9], the author presents a verified compiler from simply-typed lambda calculus to an idealized assembly language, with infinite registers and memory, and with special instructions that perform automatic memory management. This compiler does not perform optimizations, but it has the desired property of preserving the types throughout compilation, which means that the garbage collection algorithm can benefit from knowing type information statically rather than tracking it dynamically in memory.

MLCompCert [14] is a verified front-end compiling programs from a subset of ML into the source language of the CompCert C compiler. ML has a rank-1 prenex polymorphic type system, that is, it supports both the simply-typed lambda calculus functions and polymorphic higher-order functions where the universal quantifiers are in prenex position (they do not appear in the type of a function argument). MLCompCert also supports data types and pattern matching. Instead of implementing an entire translation down to machine code, they compile the code down to CompCert C, which allows them to rely on the verified compiler for the rest of the transformation and optimizations. They do perform higher-level optimizations expected from a functional language compiler: uncurryfication of multiple arguments function, tail-call optimization, and a translation to continuation-passing style (CPS).

CompCertTSO [40] is an extension of CompCert to concurrent programming. CompCert's memory model only covers a deterministic subset of the x86 semantics, and therefore does not allow reasoning about shared memory concurrency. CompCertTSO extends the memory model to expose the relaxed Total Store Ordering (TSO) semantics of modern x86 processors. They provide an alternative ClightTSO language as the entry point into the compiler. ClightTSO is a variation of Clight with threads and relaxed semantics for loads and stores. It also exposes a compare-and-swap instruction, some read-modify-write atomic instructions, and a memory barrier instruction. The relaxed memory semantics make for interesting changes in the compiler. For instance, common subexpression elimination (CSE) cannot remove memory reads, for it can lead to illegal behaviors in the presence of a concurrent thread modifying the read variable. They also implement some concurrency-specific optimizations, like redundant fence elimination.

There are also efforts to improve the safety of high-level compilers to JavaScript. These often arise in web development frameworks that provide a typed surface language that gets compiled down to pure JavaScript. In [18], the authors demonstrate a technique to preserve program equivalence through such a compilation scheme. In their setting, they face an additional challenge in proving their safety with respect to malicious JavaScript code. They use a classic forward simulation argument to prove that their translation preserves typing and heap invariants, and use a coinductive labeled bisimulation which they show coincides with contextual equivalence: they can therefore show that two high-level programs are equivalent in arbitrary contexts exactly when their JavaScript translation are equivalent in arbitrary contexts.

Further ongoing efforts are bridging the remaining challenges of verified compilation. On the backend side, some challenges are static analysis and complex optimizations, as well as modular verification and certified linking. On the frontend side, there remain challenges in formalizing more complex programming paradigms and languages, different consistency models for distributed programming, different weak memory models for shared-memory concurrency, new programming models suited to verification, with the possible use of domain-specific languages.

## 5. Separation logic

One technique that has been increasingly prevalent to reason about pointer-manipulating programs is *separation logic*. In 1972, Burstall adapts Floyd's proof technique for reasoning about control-flow graphs [17], and verifies programs that use mutable lists without sharing [6]. He notices the recurring need to state that lists contain distinct elements (they do not repeat) and have distinct elements from one another throughout computations. He defines a predicate to assert that a system of lists is distinct non-repeating. It is the same need of support for local reasoning that will give rise to separation logic, formalized by Ishtiaq, O'Hearn, Reynolds, and Yang [20][36][38].

When reasoning formally about mutable data structures, it is cumbersome to propagate antialiasing facts. On the other hand, when reasoning informally, we implicitly assume that commands whose specifications do not mention some memory locations or data structures will not affect them when executed. To bridge this gap, O'Hearn et al. extend a Floyd-Hoare logic with connectives to reason about the heap. In particular, the *separating conjuction* $P * Q$ captures the notion of separation: it asserts that the heap can be divided in two separate regions, satisfying respectively predicates $P$ and $Q$. This is a substructural logic: it does not admit contraction and weakening rules for the separating conjunction. Thus, facts about the heap cannot be duplicated or forgotten within the logic, enforcing the soundness of the heap.

The essence of separation logic is captured in the frame rule:

$$\frac{\{P\}\ c\ \{Q\}}{\{P * R\}\ c\ \{Q * R\}}\ \textsc{Frame}$$

Assuming $c$ does not modify free variables of $R$, the frame rule states that separation logic triples are preserved by separated extensions of the heap. A specification therefore only needs mention the part of the heaps it requires, but can be extended to work in any context, and the rule guarantees that the frame $R$ is preserved. We say a program can be specified with its "small footprint", which brings support for local reasoning. Figure 2 shows an example of a derivation in separation logic that makes use of the first-order frame rule to apply the specification of the `free` function while safely framing out the tail of the list being freed.

Over the following decade, this line of work has been pushed in a few directions.

For one, the logic in the initial paper is first-order and cannot capture information hiding patterns or higher-order functions and specifications. It also only deals with simple, non-recursive types. In [35], O'Hearn et al. introduce a new rule, the hypothetical frame rule (or second-order frame rule), which allows information hiding.

$$\frac{\{P_l\}\ \mathsf{library}\ \{Q_l\} \vdash \{P_c\}\ \mathsf{client}\ \{Q_c\}}{\{P_l * I\}\ \mathsf{library}\ \{Q_l * I\} \vdash \{P_c * I\}\ \mathsf{client}\ \{Q_c * I\}}\ \textsc{Hyp. Frame}$$

Assuming: $\vdash \{\texttt{x} \mapsto \_\}\ \texttt{free(x)}\ \{\texttt{emp}\}$

We show: $\vdash \{\texttt{list}\, l\, \texttt{l}\}\ \texttt{freeList(l)}\ \{\texttt{emp}\}$

```
  {list l l}
if (l is empty) {
  {emp}                                  (* by def. of list predicate *)
} else {
  { l.hd ↦ _ * list (tl l) l.tl}         (* by def. of list predicate *)
free(l.hd);
  { list (tl l) l.tl}
freeList(l.tl);
  {emp}
}
  {emp}
```

**Figure 2.** A sample derivation of a Hoare triple for a recursive function freeing a linked list. Program lines are interspersed with the facts known at the intermediate program point. The frame rule is used at the `free` step to frame out the `list` fact.

This rule allows modular composition by allowing the verification of client with regards to the specification $\{P\} - \{Q\}$ of library, while the verification of library is performed with the additional private invariant $I$. Furthermore, client is guaranteed to preserve this invariant. This rule is eventually generalized in [3] to a family of higher-order frame rules.

A related line of work has been marrying separation logic and dependtly-typed theory. Initially, Nanevski et al. [31] only embed a classical Hoare logic within a first-order classical logic, creating a Hoare Type Theory (HTT). They do so by encapsulating effectful computations within an indexed monad capturing Hoare-style pre- and post-conditions, effectively encoding a verification condition generation. They then proceed to embed separation logic constructs within a second-order variant of HTT [32], and eventually build Ynot [33], an axiomatic extension of the Coq proof assistant that supports a higher-order separation logic.

There have also been pushes on developing relational and parametric models of separation logic. Relational separation logic [43] allows reasoning about properties relating multiple programs. For instance, one can demonstrate the equivalence of two programs and verify compiler optimizations, or prove that different implementations of an abstraction have observationally equivalent behaviors. Parametricity allows proving that some code behaves independently of some choices of implementation or data structures. A model supporting parametric reasoning for a first-order logic with higher-order frame rules is presented in [4].

On the practical side, perhaps one of the major accomplishments performed in part with separation logic is the formal verification of seL4 [23], a formally-verified L4 microkernel. seL4 comes with a functional correctness proof which relates its C implementation to an abstract specification of the expected behavior. Furthermore, it comes with a binary correctness proof, which states that the binary code faithfully implements the C code. This is interesting because it bypasses the need for a verified assembler and linker, which are yet to be developed. Finally, they also demonstrate that the functional specification of their microkernel implies the desired properties of integrity (data may not be changed without permission) and confidentiality (data may not be accessed without permission).

To achieve this, they built an abstract separation logic framework for reasoning in the presence of virtual memory, using the proof assistant Isabelle/HOL. They instantiate the framework with a memory model for C and of the page tables of an ARM processor.

## 6. Verifying low-level code

In reasoning about low-level code, one must choose the granularity of abstraction at which they wish to operate. For instance, one can choose to assume the code is described by an abstract syntax tree that is neither inspectable nor modifiable by the execution of the program. More faithful formalizations choose to consider code according to its machine representation, which leads to more concerns when dealing with unaligned function pointers that can decode a different instruction sequence than the one intended. Even further, one can wish to allow and reason about self-modifying code.

The FLINT project and their Certified Assembly Programming (CAP) line of work incrementally addressed these issues. Their initial work [46][47] describes a low-level assembly language that supports a continuation-passing Hoare-style reasoning with an assertion language in the style of separation logic, which they use to build a verified library for dynamic storage allocation. It assumes an infinite memory pool, but the algorithm uses it efficiently, managing a free list and operations for splitting and coalescing memory regions. They generalize this framework to a non-preemptive concurrent CAP language named CCAP [45]. They reason compositionally about concurrency by following an *assume-guarantee* approach. Their system only supports a static number of threads, and does not support higher-order code pointers. In XCAP [34], they instead address the need for *embedded code pointers*, that is first-class function pointer manipulation, which require higher-order methods of reasoning and were a challenge to tackle in a separation logic setting. Their approach combines a modular verification of functions, which postpones a global verification of the assumptions necessary for the whole program to be safe with regards to its code pointers. The same year, they also present CAP0 [16], a generic framework for defining program logics dealing with stack-based control abstractions.

Jensen et al. [21] designed an even more realistic high-level separation logic with support for reasoning about low-level code. Their framework does not assume an infinite memory, and exposes code as data, allowing self-modification and misaligned decodings. Their logic supports first- and higher-order frame rules. They also expose a modality to encapsulate step-indexing, and a frame connective in the style of the invariant extension of [5], as well as a read-only version of the latter. This allows, for instance, recovering the usual frame rule for basic blocks as:

$$\{P\}\ c\ \{Q\} \triangleq \forall i, j.(\texttt{safe} \otimes (\texttt{EIP} \mapsto j * Q) \implies \texttt{safe} \otimes (\texttt{EIP} \mapsto i * P)) \oslash i..j \mapsto c$$

which can be read in the following way: for any two memory locations $i$ and $j$, assuming the bytes of code $c$ reside exactly between $i$ and $j$, and assuming they are not modified ($\oslash$ is the read-only frame connective), if it is safe to be in a

configuration where the instruction pointer is at $j$ and $Q$ holds, then it is safe to be in a configuration where the instruction pointer is at $i$ and $P$ holds. Here, safety refers to the fact that it is possible to keep executing for an arbitrary number of steps, following the usual CPS version of Hoare logic in low-level contexts. They apply this logic in [22] to verify realistic x86 code embedded in Coq, using Coq's features to turn it into a macro-assembler. They showcase the expressivity of the framework with examples like classic calling conventions, and a compiler for regular expressions which builds a DFA.

Myreen et al. propose an interesting approach to verifying machine code produced for different architectures. The traditional approach would be to perform verification in each architecture instruction set, using the particular semantics for that language. They propose decompilation into logic [30], an approach where one proves properties of an algorithm with regards to a pure implementation of it, and can verify that implementations of this algorithm on different architecture are correct. To do so, they implement a decompiler for each architecture, which safely transforms an assembly program into an equivalent functional program. From there, one only needs to prove functional equivalence with the algorithm to ensure that the assembly code was a faithful implementation. They have implemented such decompilers for three instruction sets (ARM, x86 and PowerPC) in HOL4 and show some examples of application of the technique in a subsequent paper. The technique is fully automated, but restricted to deterministic code with static function invocations.

## 7. Proof Automation

A cross-cutting concern in reasoning formally about programs is the mitigation of the proof burden for the user.

Proofs done in a proof assistant are often much longer and verbose than their pen-and-paper counterpart, because they must perform exhaustive analysis of all the cases, even the trivial and irrelevant ones. In particular, when dealing with programming languages, there are often dozens of instructions to be considered, if not hundreds when dealing with low-level languages. This can often result in an explosion of proof cases, some of which can be very redundant. Proof automation can help alleviate this burden by systematically dispatching irrelevant or redundant cases, leaving the user with only those cases that require expertise.

Another case for automation can be made with respect to robustness of a proof. Automation procedures are usually designed in a generic way so that they are resistant to small changes, like the appearance, disappearance, or reordering of hypotheses and subgoals.

The support for automation in proof assistants is very heterogeneous. A system like Agda lacks it almost entirely, and is therefore considered more as a programming language than a proof assistant. A system like Isabelle offers a push-button automation tactic, referred to as *sledgehammer*, which relies on first-order logic automated theorem provers (E, SPASS, Vampire) to attempt to find proofs, after having performed some transformations to try and transform the context to a first-order form. Most systems (Coq, HOL, Isabelle, Nuprl, PVS) have implemented a set of complete and incomplete decision procedures for diverse theories (arrays, fields and rings simplification, linear arithmetic, Peano arithmetic, Presburger arithmetic, primality, sequent calculus for intuitionistic propositional logic, Shostak's combination of theories...). Finally, some systems like Coq expose an extensible tactic language, that is, an untyped programming language to manipulate proof contexts and build custom decision procedures. In these systems, the users are free to build their own decision procedure libraries.

Using untyped tactic languages for writing tactics, as in Coq or Isabelle, can be unsatisfactory when one wants to ensure that tactics are correct, complete, or to compose them correctly. One would like to use the very type system available in proof assistants to check the tactics themselves. However, one cannot hope to reproduce the features of a language like Coq's Ltac within Coq, for Ltac performs operations that cannot be internalized in the logic, like syntactic pattern-matching on the open universe of propositions. A partial solution to this problem is known as computational reflection. This technique lets the user reify the current proof context as a data type, which allows writing decision or semi-decision procedures whose result can be reinjected as a proof.

The Bedrock framework [11] addresses the problem of automating the proof effort to reason about low-level programs. Bedrock is a Coq library initially instantiated with an idealized machine model (infinite memory) with higher-order function pointers, and a second-order assertion logic inspired by XCAP, where blocks of code are specified by the pre-condition which makes them safe to execute. The system provides certified macros over the assembly language that lets one use higher-level control structures to build low-level programs. The system provides tactics to generate verification conditions and discharge separation logic entailments.

The Reflex framework [39] aims at pushing the automation even further, essentially removing the need for expertise in the proof assistant. The key to this level of automation is in identifying a class of programs large enough that they can be described using a domain-specific language, but restricted enough that the assertions one may want to prove can be expressed in a simple logic. The framework instantiates this idea with the class of reactive systems that can be specified using a simple request-response model, with only a top-level loop. The user may describe the components and messages of the system, as well as how the kernel should react to component interactions. A small fragment of temporal logic allows one to state and automatically prove theorems about the proper ordering of sequences of messages, for instance to verify that a protocol is correctly implemented. An additional logical primitive allows for verification of a non-interference property between partitions of the system.

In the context of compiler verification, the XCERT [41] extension of CompCert addresses one of the issues of developing compiler optimizations: in the CompCert workflow, each optimization pass requires its own proof, which usually requires exhibiting a simulation relation and proving that it is preserved by the small-step semantics. This process is tedious, and most of the difficulty is in the crafting of the proper simulation relation. XCERT wants to allow users to write new optimizations without knowledge of Coq. To do so, it relies on the technique of Parameterized Equivalence Checking [24] (PEC) to automatically infer the simulation relation. XCERT then implements a verified optimization engine that, given the inferred simulation relation, implements and proves correct the optimization pass. While technically not done in a proof assistant (the PEC engine uses SMT solvers to discharge obligations in building the simulation relation whose correctness is then axiomatized for the verified rewrite engine), the framework is set up so that one

could get full verification by implementing a verified version of PEC.

All these frameworks provide different levels of automation and target users of different levels. Bedrock has low automation but great expressivity, and is therefore geared towards expert users who need support in reasoning with low-level code. Reflex and XCERT try to require no proof expertise: the former requires knowledge of compilers, the latter requires knowledge of reactive systems and temporal logic. However, they all seem to share a challenging issue: when the automation fails, what is to be presented to the user, and what alternatives do they have? In the systems we have seen, the user is generally left stuck in the proof that was not fully-automated. In the case of Bedrock, this is not unexpected, but for Reflex and XCERT, this contrasts clearly with the assumption that the user needs no knowledge in theorem proving.

## 8. Conclusion

As we have seen throughout this report, proof assistants span a large part of the spectrum of programming languages research, and along with automated theorem provers and software testing frameworks, they have become a tool of choice.

The somewhat rich ecosystem of proof assistants has allowed for different directions of research, from pure programming tools with dependent types to logical frameworks with high degrees of automation.

Almost all the layers of the software stack and all the different tools a programmer could need have been subject to a formalization effort. At the highest-level, run-time environments for high-level languages, parsers, garbage collection algorithms. At a middle ground, compilers from high-level languages to low-level ones, optimization passes and the algorithms they depend upon. At the lowest-level security mechanisms for software fault isolation, operating system kernels, linkers. These efforts are often carried in isolation, assuming that one can then compose all these guarantees together into a secure software stack. In practice, however, this composition is often left for future work, and the diversity of the projects makes it a challenge to accomplish. However, some projects, like the CompCert C compiler, have gained enough momentum that many developments relating to the C language build on top of it.

Generalizing from this, software architecture for verified software is a hard challenge. There is now a diversity of proof assistants, each with their own language and expressivity, and a plethora of mechanized semantics for diverse high- and low-level languages, and many specialized logics with different sets of axioms. Providing unifying frameworks, design principles for transporting these results to different settings than the one they were originally developed for, is a complex challenge that would benefit the entire community.

Proof automation is also an area with potential for improvement. There are ongoing efforts to have SMT solvers produce certificates so that they can be safely integrated into proof assistants. There are also attempts to improve tactic languages, by improving support for computational reflection and using monadic types to write certified tactics.

## Acknowledgments

I would like to thank the members of the committee, my advisor, and the students of the Programming Languages group for their stimulating discussions and their astute feedback.

## References

[1] B. Aydemir, A. Charguéraud, B. C. Pierce, R. Pollack, and S. Weirich. Engineering formal metatheory. In *ACM SIGPLAN Notices*, volume 43, pages 3–15. ACM, 2008.

[2] B. E. Aydemir, A. Bohannon, M. Fairbairn, J. N. Foster, B. C. Pierce, P. Sewell, D. Vytiniotis, G. Washburn, S. Weirich, and S. Zdancewic. Mechanized metatheory for the masses: The POPLmark challenge. In *Theorem Proving in Higher Order Logics*, pages 50–65. Springer, 2005.

[3] B. Biering, L. Birkedal, and N. Torp-Smith. BI hyperdoctrines and higher-order separation logic. In *Programming Languages and Systems*, pages 233–247. Springer, 2005.

[4] L. Birkedal and H. Yang. Relational parametricity and separation logic. In *Foundations of Software Science and Computational Structures*, pages 93–107. Springer, 2007.

[5] L. Birkedal, N. Torp-Smith, and H. Yang. Semantics of separation-logic typing and higher-order frame rules for Algol-like languages. *arXiv preprint cs/0610081*, 2006.

[6] R. M. Burstall. Some techniques for proving correctness of programs which alter data structures. *Machine intelligence*, 7(23-50):3, 1972.

[7] C. Casinghino, V. Sjöberg, and S. Weirich. Combining proofs and programs in a dependently typed language. *ACM SIGPLAN Notices*, 49(1):33–45, 2014.

[8] A. Charguéraud. The locally nameless representation. *Journal of Automated Reasoning*, 49(3):363–408, 2012.

[9] A. Chlipala. A certified type-preserving compiler from lambda calculus to assembly language. *ACM SIGPLAN Notices*, 42(6):54–65, 2007.

[10] A. Chlipala. Parametric higher-order abstract syntax for mechanized semantics. In *ACM Sigplan Notices*, volume 43, pages 143–156. ACM, 2008.

[11] A. Chlipala. Mostly-automated verification of low-level programs in computational separation logic. *ACM SIGPLAN Notices*, 46(6):234–245, 2011.

[12] A. Church. A formulation of the simple theory of types. *The journal of symbolic logic*, 5(02):56–68, 1940.

[13] T. Coquand. An algorithm for testing conversion in type theory. *Logical frameworks*, 1:255–279, 1991.

[14] Z. Dargaye. *Vérification formelle dâĂŹun compilateur optimisant pour langages fonctionnels*. PhD thesis, Université Paris 7, 2010.

[15] B. Delaware, B. C. d S Oliveira, and T. Schrijvers. Meta-theory à la carte. In *ACM SIGPLAN Notices*, volume 48, pages 207–218. ACM, 2013.

[16] X. Feng, Z. Shao, A. Vaynberg, S. Xiang, and Z. Ni. Modular verification of assembly code with stack-based control abstractions. In *ACM SIGPLAN Notices*, volume 41, pages 401–414. ACM, 2006.

[17] R. W. Floyd. Assigning meanings to programs. In *Proc. Symp. in Applied Mathematics*, volume 19, pages 19–32, 1967.

[18] C. Fournet, N. Swamy, J. Chen, P.-E. Dagand, P.-Y. Strub, and B. Livshits. Fully abstract compilation to javascript. In *ACM SIGPLAN Notices*, volume 48, pages 371–384. ACM, 2013.

[19] R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. *Journal of the ACM (JACM)*, 40(1):143–184, 1993.

[20] S. S. Ishtiaq and P. W. O'Hearn. BI as an assertion language for mutable data structures. In *ACM SIGPLAN Notices*, volume 36, pages 14–26. ACM, 2001.

[21] J. B. Jensen, N. Benton, and A. Kennedy. High-level separation logic for low-level code. In *ACM SIGPLAN Notices*, volume 48, pages 301–314. ACM, 2013.

[22] A. Kennedy, N. Benton, J. B. Jensen, and P.-E. Dagand. Coq: the world's best macro assembler? In *Proceedings of the 15th Symposium on Principles and Practice of Declarative Programming*, pages 13–24. ACM, 2013.

[23] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, et al. sel4: Formal verification of an os kernel. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 207–220. ACM, 2009.

[24] S. Kundu, Z. Tatlock, and S. Lerner. Proving optimizations correct using parameterized program equivalence. In *ACM Sigplan Notices*, volume 44, pages 327–337. ACM, 2009.

[25] V. Le, M. Afshari, and Z. Su. Compiler validation via equivalence modulo inputs. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, page 25. ACM, 2014.

[26] X. Leroy. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. *ACM SIGPLAN Notices*, 41(1):42–54, 2006.

[27] X. Leroy. A locally nameless solution to the POPLmark challenge. 2007.

[28] J. McKinna and R. Pollack. Pure type systems formalized. In *Typed Lambda Calculi and Applications*, pages 289–305. Springer, 1993.

[29] J. McKinna and R. Pollack. Some lambda calculus and type theory formalized. *Journal of Automated Reasoning*, 23(3): 373–409, 1999.

[30] M. O. Myreen, M. J. Gordon, and K. Slind. Decompilation into logicâĂŤimproved. In *Formal Methods in Computer-Aided Design (FMCAD), 2012*, pages 78–81. IEEE, 2012.

[31] A. Nanevski and G. Morrisett. Dependent type theory of stateful higher-order functions. Technical report, Technical Report TR-24-05, Harvard University, 2005.

[32] A. Nanevski, G. Morrisett, and L. Birkedal. Polymorphism and separation in Hoare type theory. In *ACM SIGPLAN Notices*, volume 41, pages 62–73. ACM, 2006.

[33] A. Nanevski, G. Morrisett, A. Shinnar, P. Govereau, and L. Birkedal. Ynot: dependent types for imperative programs. In *ACM Sigplan Notices*, volume 43, pages 229–240. ACM, 2008.

[34] Z. Ni and Z. Shao. Certified assembly programming with embedded code pointers. *ACM SIGPLAN Notices*, 41(1): 320–333, 2006.

[35] P. W. O'Hearn, H. Yang, and J. C. Reynolds. Separation and information hiding. In *ACM SIGPLAN Notices*, volume 39, pages 268–280. ACM, 2004.

[36] P. OâĂŹHearn, J. Reynolds, and H. Yang. Local reasoning about programs that alter data structures. In *Computer science logic*, pages 1–19. Springer, 2001.

[37] F. Pfenning and C. Elliot. Higher-order abstract syntax. In *ACM SIGPLAN Notices*, volume 23, pages 199–208. ACM, 1988.

[38] J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Logic in Computer Science, 2002. Proceedings. 17th Annual IEEE Symposium on*, pages 55–74. IEEE, 2002.

[39] D. Ricketts, V. Robert, D. Jang, Z. Tatlock, and S. Lerner. Automating formal proofs for reactive systems. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, page 47. ACM, 2014.

[40] J. Ševčík, V. Vafeiadis, F. Zappa Nardelli, S. Jagannathan, and P. Sewell. Compcerttso: A verified compiler for relaxed-memory concurrency. *Journal of the ACM (JACM)*, 60(3): 22, 2013.

[41] Z. Tatlock and S. Lerner. Bringing extensibility to verified compilers. In *ACM Sigplan Notices*, volume 45, pages 111–121. ACM, 2010.

[42] G. Washburn and S. Weirich. Boxes go bananas: Encoding higher-order abstract syntax with parametric polymorphism. In *ACM SIGPLAN Notices*, volume 38, pages 249–262. ACM, 2003.

[43] H. Yang. Relational separation logic. *Theoretical Computer Science*, 375(1):308–334, 2007.

[44] X. Yang, Y. Chen, E. Eide, and J. Regehr. Finding and understanding bugs in c compilers. In *ACM SIGPLAN Notices*, volume 46, pages 283–294. ACM, 2011.

[45] D. Yu and Z. Shao. *Verification of safety properties for concurrent assembly code*, volume 39. ACM, 2004.

[46] D. Yu, N. A. Hamid, and Z. Shao. *Building certified libraries for PCC: Dynamic storage allocation*. Springer, 2003.

[47] D. Yu, N. A. Hamid, and Z. Shao. Building certified libraries for PCC: Dynamic storage allocation. *Science of Computer Programming*, 50(1):101–127, 2004.