Correctly Compiling Proofs About Programs Without Proving Compilers Correct

Audrey Seo*
University of Washington, USA

Chris Lam*
University of Illinois Urbana-Champaign, USA

Dan Grossman
University of Washington, USA

Talia Ringer
University of Illinois Urbana-Champaign, USA

Abstract

Guaranteeing correct compilation is nearly synonymous with compiler verification. However, the correctness guarantees for certified compilers and translation validation can be stronger than we need. While many compilers do have incorrect behavior, even when a compiler bug occurs it may not change the program’s behavior meaningfully with respect to its specification. Many real-world specifications are necessarily partial in that they do not completely specify all of a program’s behavior. While compiler verification and formal methods have had great success for safety-critical systems, there are magnitudes more code, such as math libraries, compiled with incorrect compilers, that would benefit from a guarantee of its partial specification.

This paper explores a technique to get guarantees about compiled programs even in the presence of an unverified, or even incorrect, compiler. Our workflow compiles programs, specifications, and proof objects, from an embedded source language and logic to an embedded target language and logic. We implement two simple imperative languages, each with its own Hoare-style program logic, and a framework for instantiating proof compilers out of compilers between these two languages that fulfill certain equational conditions in Coq. We instantiate our framework on four compilers: one that is incomplete, two that are incorrect, and one that is correct but unverified. We use these instances to compile Hoare proofs for several programs, and we are able to leverage compiled proofs to assist in proofs of larger programs. Our proof compiler framework is formally proven sound in Coq. We demonstrate how our approach enables strong target program guarantees even in the presence of incorrect compilation, opening up new options for which proof burdens one might shoulder instead of, or in addition to, compiler correctness.

2012 ACM Subject Classification Theory of computation → Logic and verification; Theory of computation → Hoare logic; Software and its engineering → Compilers

Keywords and phrases proof transformations, compiler validation, program logics, proof engineering

Digital Object Identifier 10.4230/LIPIcs.CVIT.2016.23

Funding This research was developed with funding from the Defense Advanced Research Projects Agency. The views, opinions, and/or findings expressed are those of the author(s) and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government.

* Co-first authors
1 Introduction

Program logic frameworks help proof engineers do more advanced reasoning about program-specific properties. Iris [17, 23], VST [8], CHL [10], and SEPREF [26] are just a few examples of such program logics. Traditionally, strong guarantees for compiled programs required composing program logics with verified compilers [8]. However, because functional specifications are often partial, preserving them through compilation sometimes does not require a correct compiler pass, much less global compiler correctness.

To see an example of where correct compilation becomes too strict, consider a Hoare triple \( \{0 \leq a \land 0 \leq \epsilon\} y := 42; x := \text{source_sqrt}(a) \{\{|a - x^2| \leq \epsilon\}\} \), which says that after setting \( y \) to 42 and calling \( \text{source_sqrt} \) on \( a \), the variable \( x \) stores a square root approximation of \( a \) within \( \epsilon \). Suppose that \( \text{source_sqrt} \) is compiled to some program \( \text{target_sqrt} \) such that if \( 0 \leq a \land 0 \leq \epsilon \), then after \( \text{target_sqrt}(a) \) runs, we have \( |a - x^2| \leq \frac{\epsilon}{2} \). In the end, we still have \( |a - x^2| \leq \epsilon \) for \( \text{target_sqrt} \) since \( \frac{\epsilon}{2} \leq \epsilon \), which meets the specification. Moreover, the 42 on the right-hand side of the assignment to \( y \) could be (mis)compiled to anything, and the specification would still be preserved. However, this compilation would be rejected by both certified compilation and translation validation, illustrating that compiler correctness is significantly more restrictive than specification preservation.

In order to achieve guaranteed specification-preserving compiler passes, we present the proof compiler framework PotPie. PotPie takes an existing compiler and produces a proof compiler. A proof compiler takes a program, a specification, and a proof of the specification and compiles all three such that (1) the specification’s meaning is preserved, and (2) the compiled proof shows that the compiled program meets the compiled specification.

PotPie is formally verified in Coq, and allows for partial specification-preserving compilation, even of incorrectly compiled programs. To get a sense of how PotPie differs from similar techniques, imagine a proof engineer has already shown the Hoare triple \( \{0 \leq a \land 0 \leq \epsilon\} x := \text{source_sqrt}(a) \{|x^2 - a| \leq \epsilon\} \) and wants to prove an analogous Hoare triple about the compiled square root approximation. Suppose also that the proof engineer has a compiler \( T \) on hand, which happens to have a small bug that switches \( < \) to \( \leq \) in programs and specifications. The square root program uses a while loop to approximate square roots, and the while loop condition contains at least one \( < \). At this point, PotPie provides two options:

1. **Tree workflow**: use \( T \) to instantiate a proof tree compiler that produces a target proof tree. After compiling the square root Hoare tree, they invoke the Tree Coq plugin which will check the proof tree, and if possible, produce a certificate that is checkable in Coq. Tree has only one proof obligation to invoke the plugin, but may fail in certain cases.

2. **CC workflow**: use \( T \) to instantiate a correct-by-construction proof compiler by showing that it satisfies the equations in Figure 5. To call this proof compiler, the proof engineer must show that the square root program is well-formed. CC is complete in that if the translation preserves the specification, then it is possible to perform.

Both methods work, even though the compiler \( T \) has a bug that causes miscompilation in the square root program. Because of this miscompilation, we cannot use translation validation, the state of the art for ensuring correct compilation for an unverified compiler. But the miscompilation does not affect our specification, so with PotPie, we can get strong guarantees about our compiled code regardless of miscompilation.

We make the following contributions:

1. We present the PotPie framework for specification-preserving proof compilation.
2. We describe two workflows for the PotPie framework: CC and Tree.

\[ a ::= \mathbb{N} \mid x \mid \text{param } k \mid a + a \mid a - a \mid f(a, \ldots, a) \]

\[ a ::= \mathbb{N} \mid \#k \mid a + a \mid a - a \mid f(a, \ldots, a) \]

\[ b ::= T \mid F \mid \neg b \mid a \leq a \mid b \land b \mid b \lor b \]

\[ b ::= T \mid F \mid \neg b \mid a \leq a \mid b \land b \mid b \lor b \]

\[ i ::= \text{skip} \mid x ::= a \mid i ; i \]

\[ \lambda ::= (f, k, i, \text{return } x) \]

\[ \lambda ::= (f, k, i, \text{return } a \ n) \]

\[ p ::= \{(\lambda, \ldots, \lambda), i \} \]

\[ p ::= \{(\lambda, \ldots, \lambda), i \} \]

\[ i ::= \text{skip} \mid \text{push} \mid \text{pop} \mid \#k ::= a \mid i ; i \]

\[ \mid \text{if } b \text{ then } i \text{ else } i \mid \text{while } b \text{ do } i \]

\[ \lambda ::= (f, k, i, \text{return } a \ n) \]

3. We demonstrate PotPie on several case studies, using code compilers with varying

degrees of incorrectness to correctly compile proofs. Our case studies include various

mathematical functions, such as infinite series and square root approximation.

4. We prove the CC and Tree workflows sound in Coq.

Non-Goals and Limitations Our work aims to complement, not replace, certified compilation.

One potential motivation for alternative compiler correctness techniques is to ease the burden

of compiler verification. However, easing the burden of compiler verification is not our

goal, nor do we think that this is the case for our work at this time. Rather, our goal is

demonstrate a complementary approach of specification-preserving compilation for program-
specific specifications, even when the program itself is incorrectly compiled. Our work

currently focuses on simple and closely related languages, and the compilers are likewise

simple, though we do not believe that these choices are central to our approach. Currently,

our work imposes significant limitations the kinds of control flow optimizations that can be

performed. This simplifying decision made the problem initially tractable, but we do not

believe it is inherent to our approach; we discuss a potential way of handling it in Section 7.

2 Programs, Specifications, and Proofs

In this section, we briefly present our six languages and how to compile programs and

specifications, with Section 2.1 describing the programming languages and program compiler,

Section 2.2 describing the specification languages and compiler, and Section 2.3 describing

the proof languages (the proof compiler framework is described in Section 3). Here and

throughout the paper, we include links such as \( \text{(immutable) parameters and local scope. STACK’s functions can access the entire stack.} \)

Bridging the Abstraction Gap The difference in memory model must be taken into account

when compiling from IMP to STACK. We define an equivalence between variable environments

\( \text{Stack} \text{’s functions can access the entire stack.} \)
Correctly Compiling Proofs About Programs Without Proving Compilers Correct

\[
\begin{align*}
\text{comp}_\sigma^x(n) & \triangleq n \\
\text{comp}_\sigma^x(x) & \triangleq \# \varphi(x) \\
\text{comp}_\sigma^x(\text{param} \ k) & \triangleq \#(|V| + k + 1) \\
\text{comp}_\sigma^x(a_1 \text{ op } a_2) & \triangleq \text{comp}_\sigma^x(a_1) \text{ op } \text{comp}_\sigma^x(a_2) \\
\text{comp}_\sigma^x(f(a_1, \ldots, a_n)) & \triangleq f(\text{comp}_\sigma^x(a_1), \ldots, \text{comp}_\sigma^x(a_n)) \\
\text{comp}_\sigma^x(T) & \triangleq T \\
\text{comp}_\sigma^x(F) & \triangleq F \\
\text{comp}_\sigma^x(\text{b} \text{ op } \text{b}_2) & \triangleq \text{comp}_\sigma^x(b_2) \text{ op } \text{comp}_\sigma^x(\text{b}_2) \\
\text{comp}_\sigma^x(a_1 \leq a_2) & \triangleq \text{comp}_\sigma^x(a_1) \leq \text{comp}_\sigma^x(a_2)
\end{align*}
\]

**Figure 2** An arithmetic expression compiler \( \text{comp}_\sigma \) (left) and a boolean expression compiler \( \text{comp}_\sigma \) (right). \( \text{op} \) stands for the appropriate binary operators: + and −, and ∧ and ∨, respectively.

\[
M ::= T | F | p_n [e, \ldots, e] \\
| M \land M | M \lor M \\
\sigma \vdash \text{TRUE} \\
\sigma \vdash p_n [a_1, \ldots, a_n] \\
\text{map_eval}_\sigma[a_1]^n[v_i]^n [v_i]^n \\
\text{N-ARY}
\]

**Figure 3** Syntax (left) and semantics (right) for base assertions for both IMP and STACK. \text{map_eval}_\sigma is a relation from lists of expressions to lists of values. The semantic interpretation is parametric over the types of \( v, \sigma \), and \text{map_eval}_\sigma. Interpretations for \( \land \) and \( \lor \) are standard.

**Definition 1.** Let \( \varphi : V \to \{1, \ldots, |V|\} \) be bijective with inverse \( \varphi^{-1} \). Then for all variable stores \( \sigma \), parameter stores \( \Delta \), and stacks \( \sigma_s \), we say that \( \sigma \) and \( \Delta \) are \( \varphi \)-equivalent to \( \sigma_s \), written \( (\sigma, \Delta) \equiv_\varphi \sigma_s \), if (1) for \( 1 \leq i \leq |V| \), we have \( \sigma_s[i] = \sigma(\varphi^{-1}(i)) \), and (2) for \( |V| + 1 \leq i \leq |V| + |\Delta| \), we have \( \sigma_s[i] = \Delta[i - |V|] \).

This equivalence is entirely dependent on our choice of mapping between variables and stack slots. It has this form since parameters are always at the top of the stack at the beginning of a function call, and are then pushed down as space for local variables is allocated, so parameters appear “after” (i.e., appended to) the local variables. Note that this implies \( |V| + |\Delta| \leq |\sigma_s| \) while saying nothing about stack indices beyond \( |V| + |\Delta| \).

**Compiling Programs** Although the POTPIE framework allows for some choice of compiler between IMP and STACK, most of our compilers follow a common structure. We give a translation for IMP arithmetic and boolean expressions (which we will refer to in sum as \text{expressions} from now on) in Figure 2. This infrastructure is a straightforward extension of the variable mapping function \( \varphi \) from Definition 1. The program compilers we deal with in our case studies (Section 4) define variations on this common structure.

**2.2 Specifications** The specification languages both embed IMP or STACK expressions inside of them, respectively. Base assertions are modeled as n-ary predicates over the arithmetic and boolean expressions of the given language. The semantics for assigning a truth value to a formula (Figure 3, right) parameterize predicates over the value types. For example, if we have the assertion \( p_1 \ a \) where \( a \) is an IMP expression that evaluates to \( v \), then \( p_1 \ a \) is true if and only if calling the Coq definition of \( p_1 \) with \( v \) is a true \text{Prop}. We can define a program logic \( SM \) for the source language this way by using the atoms in Figure 3 to embed arithmetic and boolean expressions in Coq propositions. We add conjunction and disjunction connectives at the logic level. We can define \( TM \) for the target language similarly. We then use this to construct the following specification grammars:

\[
SM ::= SM \land SM | SM \lor SM | TM \land TM \land TM \lor TM \lor TM
\]

where \( SM_e \) and \( TM_e \) are instances of the logic described in Figure 3 using IMP and STACK arithmetic and boolean expressions respectively.
\[
\text{comp}_{\varphi,k}^\text{spec}(T) \triangleq (k, T) \\
\text{comp}_{\varphi,k}^\text{spec}(F) \triangleq (k, F) \\
\text{comp}_{\varphi,k}^\text{spec}(SM_1 \text{ op } SM_2) \triangleq \text{comp}_{\varphi,k}^\text{spec}(SM_1) \text{ op } \text{comp}_{\varphi,k}^\text{spec}(SM_2)
\]

Figure 4 The specification compiler \(\text{comp}_{\varphi,k}^\text{spec}(SM)\), which is parameterized over \(\text{comp}_{\varphi}^\text{expr}\) (which can be either \(\text{comp}_{\varphi}^a\) or \(\text{comp}_{\varphi}^b\), depending on the type of expressions \(e\)). \text{op} is either \& or \lor.

Because the minimum stack size required by the compilation might not be captured by language expressions contained within the formula itself, we also want to specify a minimum stack size in Stack specifications. This is represented by the following judgement:

\[
|\sigma| \geq n \quad \sigma \vdash TM_e \\
\sigma \models (n, TM_e) \quad \text{STACK Base}
\]

We made the decision to allow function calls within specifications. This is not essential to our approach—one could disallow effectful constructs from expressions as in CLight [6]. For the current framework, we find it more natural to reason about effectful expressions in Imp.

### Compiling Specifications

We can reuse \(\varphi : V \rightarrow \{1, \ldots, |V|\}\) and the expression compilers from Section 2.1 to define a specification compiler (see Figure 4): recurse over the source logic formula and compile the leaves, i.e., Imp expressions. If \(k\) is the number of function arguments, give each assertion a minimal stack size, \(|V| + k\), to ensure well-formedness of the resulting Stack expressions within the specification, which is given as the maximum value of \(\varphi\) plus \(k\), where \(k\) is the number of arguments. Note that this definition is parameterized over an expression compiler, which need not be fully correct. To guarantee correctness of a translated proof in the sense that the target proof "proves the same thing", users must show that the specification compiler must be sound with respect to the user’s source specification (see Definition 3 and Section 3.2.2). This ensures that the compiled proof proves an analogous property even when the program is compiled incorrectly.

### 2.3 Proofs

Our logics are based on standard Hoare logic and are proven sound in Coq. Automatically ensuring that the rule of consequence’s implications are preserved by compilation would usually require correctness of compilation. To remove this requirement, we modify the rule of consequence so that implications must be in an implication database \(I\), which is a list of pairs of specifications that satisfy the following definition:

\[\text{Definition 2. } I \text{ is valid if for each pair } (P, Q) \text{ in } I, \forall \sigma, \sigma \vdash P \Rightarrow \sigma \vdash Q.\]

This implication database, which is present for both Imp and Stack, serves to (1) identify which implications must be preserved through compilation, and (2) make it easy to identify which source implication corresponds to which target implication across compilation. For the Stack logic, as a simplifying assumption, we further require all expressions in assignments, if conditions, or while conditions to be side effect-free, i.e., preserve the stack.

### 3 Compiling Proofs

PotPie’s two workflows share the same goal: to produce a term at the target representing a proof tree for the desired Stack-level property. To achieve this, both workflows have their own soundness theorems (Section 3.1), which need certain properties to be true of compiled programs and specifications. The workflows obtain these in different ways. Before
Table 1: Proof obligations and their relationship to the requirements for instantiating and invoking proof compilers (PC) for each of our workflows, and what properties may be guaranteed for Tree by these proof obligations. P means a user proof is required, A means that the plugin will attempt an automated check, × means the condition is not required, and - means the condition is not applicable to that column. "Trees WF" means the compiled code and assertions within the Stack Hoare tree have the right syntactic shape for Hoare rule application. "Valid Tree" means that the tree is a valid Stack Hoare proof (which is implied by a typechecked certificate). "CGC" indicates what is needed to ensure that once a certificate is generated and typechecks, that it is correct, i.e., preserves the meaning of the pre and postcondition. Since CC is correct-by-construction, all of the proof obligations are required.

<table>
<thead>
<tr>
<th>Comp, Comm.</th>
<th>Create PC</th>
<th>Invoke PC</th>
<th>Guaranteeing Properties</th>
<th>Create PC</th>
<th>Invoke PC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Spec DB</td>
<td>×</td>
<td>-</td>
<td>A</td>
<td>A</td>
<td>-</td>
</tr>
<tr>
<td>Pre/Post</td>
<td>- ×</td>
<td>× P</td>
<td>× P</td>
<td>- P</td>
<td>- P</td>
</tr>
<tr>
<td>IMP WF</td>
<td>- × ×</td>
<td>- × ×</td>
<td>A</td>
<td>A</td>
<td>-</td>
</tr>
<tr>
<td>preservesStack</td>
<td>- × ×</td>
<td>- × ×</td>
<td>A</td>
<td>A</td>
<td>-</td>
</tr>
</tbody>
</table>

being called, CC requires the user to prove certain equational properties about the compiler (Section 3.2.1) and well-formedness properties of the source program and proof (Section 3.2.3), and combines these to acquire the required syntactic and stack-preserving conditions for applying Stack Hoare rules. Tree simply compiles the Hoare proof tree, and its plugin performs an automated check (that can possibly fail) of whether the compiled tree is a valid Stack Hoare proof. Additionally, both workflows require the user to manually translate the implication databases (Section 3.2.2) to retrieve Stack-level rule-of-consequence applications. A breakdown of which proof obligations are required for which workflow and the guarantees they provide can be found in Table 1. None of these proof obligations require full semantic preservation; they allow for some miscompilation of programs as long as compilation does not break the (possibly partial) specification.

3.1 Soundness Theorems and Overview

Consider the IMP Hoare triple \( \{5 < 10\} x := 5\{x < 10\} \), which can be derived via a simple application of the IMP-level assignment rule. If we map \( x \) to stack slot #1, the “natural” translation of this IMP triple is the Stack triple \( \{5 < 10\} #1 := 5\{#1 < 10\} \), which can be derived via Stack’s assignment Hoare rule. This translation seems “natural” for two reasons: it is derived using the “same” rules, and it is proving the “same” thing. We use the former to compile the proofs, and we use the latter to define a notion of soundness for specification translation (30 (31), which each workflow can guarantee in a different way:

**Definition 3.** For a given \( P \), a specification compilation function \( \text{comp}_{\text{spec}}^{\text{imp}} \) is sound with respect to \( P \) if for all \( \sigma, \Delta, \sigma_s \) such that \( (\sigma, \Delta) \approx \sigma_s \), we have \( \sigma, \Delta \models P \leftrightarrow \sigma_s \models \text{comp}_{\text{spec}}^{\text{imp}}(P) \).

We can also define an informal notion of soundness for a proof compiler:

**Definition 4.** Given an IMP Hoare proof \( pf \) that proves the triple \( \{P\}c\{Q\} \), a proof compiler \( PC \) is sound with regards to it if \( PC(pf) = pf' \) and \( pf' \) proves the triple \( \{\text{comp } P\}(\text{comp } c)\{\text{comp } Q\} \).

Combining both notions of soundness lets us arrive at our definition of soundness for a proof compiler: if a specification and proof compiler are sound with regards to a specification and proof in the sense of Definitions 3 and 4, then the compiled version of that proof is both
A valid proof at the target and proves the same thing that the source proof proved. The Tree workflow can achieve these guarantees in piecewise progression when certain proof obligations are met, and CC always guarantees both when it is called. The form Definition 4 takes in our implementation is a method of constructing a term of type \texttt{hl_stk} (the STACK correct-by-construction Hoare proof type) from a term of type \texttt{hl_Imp_Lang}.

**Tree Proof Compiler** The Tree workflow utilizes a proof compiler that separates proof and compilation, and has two components: a compiler that produces a proof tree \((\text{Tree})\) and a Coq plugin, implemented in OCaml \((\text{CC})\), that checks the proof tree’s validity. The compiler is parameterized over the code and specification compilers from IMP to STACK. The proof tree compiler component is sound in the sense that if the proof obligations for the CC proof compiler are satisfied, then it will always produce a sound tree \((\text{Tree})\). The plugin can be used on any STACK proof tree and can optionally produce a certificate, which can be used to produce a STACK Hoare logic proof via this theorem:\(\text{CC}\)

\[
\text{Theorem valid_tree_can_construct_hl_stk} \quad \\
\begin{align*}
(P Q &: \text{AbsState}) (i &: \text{imp_stack}) (\text{facts}' &: \text{implication_env_stack}) \\
(fenv' &: \text{fun_env_stack}) (T &: \text{stk_hoare_tree}) & : \\
\forall (V &: \text{stk_valid_tree P i Q facts'} fenv') (\text{certificate type}) \\
\text{hl_stk P i Q facts'} fenv'.
\end{align*}
\]

An instance of Definition 4 can be retrieved by an appropriate substitution of variables.

We note that Tree is not complete: the requisite target-level properties could be true, and yet Tree will still fail. This can occur in the case of mutually recursive functions, along with some edge cases that we talk more about in Section 5.1. While Tree requires fewer proof obligations, it also provides fewer guarantees. One such guarantee it lacks is preservation of the pre and postcondition, i.e., specification-preserving compilation. This and other guarantees can be gained by showing the proof obligations indicated in Table 1.

**CC Proof Compiler** This workflow is correct by construction. Given an IMP Hoare proof \((\text{hl_Imp_Lang})\) along with the CC proof obligations (described in Section 3.2), CC produces a STACK Hoare proof \((\text{hl_stk})\) of the same property \((\text{CC})\) (some detail is omitted for brevity):

\[
\text{Definition proof_compiler} :
\begin{align*}
\forall (P Q &: \text{AbsEnv}) (i &: \text{imp_Imp_Lang}) (fenv &: \text{fun_env}) (\text{facts}: \text{implication_env}) \\
(var_to_stack_map &: \text{list string}) (\text{num_args} &: \text{nat}) \\
\text{(proof: hl_Imp_Lang P i Q facts} fenv') (\text{translate_facts: valid_imp_trans_def}) & : \\
\text{(well-formedness conditions and specification translation soundness \(\ast\))} & \rightarrow \\
\text{hl_stk (comp P) (comp i) (comp Q) (comp facts) (comp fenv)}.
\end{align*}
\]

Since the CC proof compiler is correct-by-construction, the type signature in the above Coq code guarantees the validity of the produced target Hoare proof. However, as compared to Tree, CC requires far more proof obligations before a CC proof compiler can even be instantiated, with invocation requiring several on top of the instantiation burden.

### 3.2 Proof Obligations

PotPie’s workflows both require some proof obligations in order to get target-level correctness guarantees. Table 1 breaks down these requirements for both workflows.

#### 3.2.1 Commutativity Equations – CC Only

These code and specification compiler proof obligations relate the compiled programs and specifications. CC requires that proof-compileable IMP programs and specifications satisfy the
23:8 Correctly Compiling Proofs About Programs Without Proving Compilers Correct

\[
\begin{align*}
\text{comp}_{\text{spec}}^{\varphi,k}(P[x \rightarrow a]) &= (\text{comp}_{\text{spec}}^{\varphi,k}(P))[\varphi(x) \rightarrow \text{comp}_{\text{code}}^k(a)] \\
\text{comp}_{\text{spec}}^{\varphi,k}((p_1 [b]) \land P) &= (k + \{V\}, \{p_1 \text{ comp}_{\text{spec}}^k(b)\} \land \text{comp}_{\text{spec}}^{\varphi,k}(P)) \\
\text{comp}_{\text{code}}^{\varphi,k}(x := a) &= \#\varphi(x) := \text{comp}_{\text{code}}^k(a) \\
\text{comp}_{\text{code}}^{\varphi,k}(\text{skip}) &= \text{skip} \\
\text{comp}_{\text{code}}^{\varphi,k}(i_1; i_2) &= \text{comp}_{\text{code}}^{\varphi,k}(i_1); \text{comp}_{\text{code}}^{\varphi,k}(i_2) \\
\text{comp}_{\text{code}}^{\varphi,k}(\text{if } b \text{ then } i_1 \text{ else } i_2) &= \text{if } \text{comp}_{\text{code}}^k(b) \text{ then } \text{comp}_{\text{code}}^{\varphi,k}(i_1) \text{ else } \text{comp}_{\text{code}}^{\varphi,k}(i_2) \\
\text{comp}_{\text{code}}^{\varphi,k}(\text{while } b \text{ do } i) &= \text{while } \text{comp}_{\text{code}}^k(b) \text{ do } \text{comp}_{\text{code}}^{\varphi,k}(i)
\end{align*}
\]

Figure 5 Equations compilers must satisfy to be used to instantiate a proof compiler.

These equations in Figure 5—TREE has no such requirement (Table 1) and will simply fail if these equations don’t hold. For example, consider the substitution performed by the assignment rule. Given some \( P \), in order to compile an application of the assignment rule, we want (2) to hold. If we have this equality, we have the following, where \( P' = \text{comp}_{\text{spec}}^{\varphi,k}(P) \):

\[
\text{comp}_{\text{pf}}^{\varphi,k}(\{P[x \rightarrow a]\} x := a \{P\}) = \{P'[\varphi(x) \rightarrow \text{comp}_{\text{code}}^k(a)]\} \varphi(x) := a \{P'\}
\]

This compiler proof obligation lets a CC proof compiler mechanically apply the Hoare rules.

In practice, as long as the program compilers are executable, these conditions are provable using reflexivity. These equations are the reason for the control-flow restrictions mentioned in the introduction and in Section 7. These equations also ensure that the specification compiler is “aware” of the way that expressions are compiled. For example, consider a code compiler that adds 1 to assignment statements’ right hand sides. This breaks the compilation of the assignment rule, as the specification compiler is “unaware” of a transformation that affects a Hoare rule application. Equations 2-4 and 7-8 in Figure 5 are to prevent such cases.

3.2.2 Specification Translation Conditions – Tree & CC

As we described in Section 2.3, the rule of consequence is the only Hoare rule that depends on the semantics of the program, and thus would require a completely correct compiler pass to completely automate. Our solution is to have the user specify which implications they are using in their Hoare proof in an implication database. Then the user proves that these implications are compiled soundly (2) (this is the “Spec DB” proof obligation in Table 1):

Definition 5. Given \( \varphi, k \), and a function environment, an IMP implication \( P \Rightarrow Q \) has a valid translation if for all \( \sigma, \Delta, \sigma_s, \) if \( (\sigma, \Delta) \vartriangleright_\varphi \sigma_s, \) then \( \sigma_s \models \text{comp}_{\text{spec}}^{\varphi,k}(P) \Rightarrow \text{comp}_{\text{spec}}^{\varphi,k}(Q) \).

While it lets us construct a proof in the target about the compiled program, it does not necessarily construct a proof of the same property, as the meaning of the precondition and postcondition could be destroyed by, for instance, compiling them both to \( \bot \).

To prevent this, another proof obligation is to prove the pre/postcondition of the IMP Hoare proof sound with regards to the specification compiler (Definition 3). This guarantees that while program behavior can change, the specification remains the same. This is in Table 1 as the “Pre/Post” row. While it is required by CC, it is optional for TREE but is needed to guarantee correctness of a certificate, hence the P in the CGC column of Table 1.

These conditions only need for compilation to preserve Definitions 3 and 5 and require no proofs of language-wide properties, nor of full compiler correctness. Rather, they require specific correctness properties for a finite set of assertions. In practice, we have found these proofs to be repetitive, and have built some tactics to solve these goals (28) (29). We have not built proof automation to generate a given proof’s implication database as a verification condition but we suspect this could be done via a weakest precondition calculation.
Table 2 The lines of code, number of theorems, and the time it took for the *Tree* plugin to generate and check our case studies in Section 4.1. “Core” refers to proving the source Hoare triple. “Tree” refers to how much work it took to get to the point where one could call the *Tree* plugin (which is different from calling the tree compiler, which is simply a one-liner), and “TreeC” the additional effort needed to ensure correctness. “CC” gives how much more work it would take to be able to use the CC workflow after ensuring tree compilation correctness.

<table>
<thead>
<tr>
<th>Multiplication</th>
<th>Exponentiation</th>
<th>Series</th>
<th>Square Root</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOC</td>
<td>Core Tree</td>
<td>TreeC</td>
<td>Core Tree</td>
</tr>
<tr>
<td>Theorems</td>
<td>209</td>
<td>104</td>
<td>56</td>
</tr>
<tr>
<td>TREE CG (s)</td>
<td>3.172</td>
<td>0.154</td>
<td>2.781</td>
</tr>
</tbody>
</table>

3.2.3 Well-formedness Conditions – CC Only

The last set of user proof obligations is specific to our choice of languages and logics. Specifically, while the syntax of *Imp* prevents most type errors, there are other ways a program can be malformed, e.g., calling a function with an incorrect number of arguments. These obligations show that all components of the source proof be well-formed. Additionally, any compiled functions should preserve the stack, so as to meet the preservesStack condition of the STACK logic. We have largely automated these proof burdens in our case studies.

4 Case Studies

We have two sets of case studies that highlight the tradeoffs of the *PotPie* framework:

1. Partial Correctness with Incorrect Compilation (Section 4.1): We prove meaningful partial correctness properties of arithmetic approximation functions that are slightly incorrectly compiled. This set of case studies highlights two benefits of *PotPie*:
   a. Specification-Preserving Compilation: We invoke *PotPie* with a slightly buggy program compiler to produce proofs that meaningfully preserve the correctness specifications down to the target level. Importantly, we obtain these meaningful target-level correctness proofs of our specification even though the program compiler does not preserve the full semantic behavior of the arithmetic approximation functions.
   b. Compositional Proof Compilation. We use *PotPie* to separately compile the correctness proofs of helper functions common to both approximation functions. Composition of those helper proofs within the target-level proof of the arithmetic function comes essentially “for free,” modulo termination conditions.

2. *PotPie* Three Ways (Section 4.2): We instantiate *PotPie* with three different variants of a program compiler (incomplete, incorrect, and correct but unverified), and briefly explore the tradeoffs of each of these instantiations.

4.1 Partial Correctness with Incorrect Compilation

We have written and proven correct two mathematics approximation programs in *Imp*. Both approximation programs use common helper functions, which we also prove correct (Section 4.1.1). We then build on and compose the helper proofs to prove our approximation programs correct up to specification even in the face of incorrect compilation (Sections 4.1.2 and 4.1.3). Our incorrect compiler has the following bug, miscompiling $\prec$ to $\leq$:

$$\text{comp}_{b_{badb}}(a_1 \prec a_2) \triangleq \text{comp}_{a}(a_1) \leq \text{comp}_{a}(a_2)$$

$\text{comp}_{b_{badb}}$ is a buggy boolean expression compiler that turns our less-than macro into a less-than-or-equal-to expression. While we do not have a less-than operator in the *Imp*
language, we have a less than macro defined as \( a_1 \leq a_2 \land \neg(a_1 \leq a_2 \land a_2 \leq a_1) \). For simplicity, we will use \(<\) in this paper. The resulting program compiler is correct for programs that do not contain \(<\), and we use it throughout this subsection. We give a short summary of the proof effort that it took to prove these case studies in Table 2.

### 4.1.1 Helper Functions

We describe how we compile proofs about two helper functions: multiplication and exponentiation. For clarity, we omit environments in the lemmas we state here.

**Multiplication** The first helper function is a multiplication function, which behaves as expected (code in green is actually wrapped Coq terms, whereas code in black is an expression in our language substituted into a Coq term as per the semantics of our logic in Figure 3):

```plaintext
{ ⊤ } x := param 0; y := 0; while (1 \leq x) do
  y := y + param 1;
  x := x - 1;
{ y = (param 0) \cdot (param 1) }
```

The proof of this IMP Hoare triple is straightforward since the body of the function does not encounter the incorrect behavior of the compiler. By combining this triple with a termination proof, we are able to generate a helper lemma that relates applications of the IMP multiplication function to Coq’s `Nat.mul`:

```plaintext
Lemma mult_aexp_wrapper a1 a2 n1 n2 : a1 \downarrow n1 \rightarrow a2 \downarrow n2 \rightarrow mult(a1, a2) \downarrow (n1 \times n2) \%nat.
```

This lemma lets us reason more directly about nats. We use this lemma in the subsequent case studies, demonstrating how PotPie enables us to reuse the source Hoare proof of this triple to get the target-level version of this lemma *almost* for free—we still have to reprove termination at the target level, something we hope to address in future work.

**Exponentiation** Exponentiation is similarly straightforward, except we use multiplication as defined above as a function in its body and thus must use the multiplication function wrapper to prove the loop invariant, and we obtain the following wrapper:

```plaintext
Lemma exp_aexp_wrapper : forall a1 a2 n1 n2, a1 \downarrow n1 \rightarrow a2 \downarrow n2 \rightarrow exp(a1, a2) \downarrow n2^n.
```

### 4.1.2 Geometric Series

One example use case for partial correctness specifications is floating point estimation of mathematical functions, like \( \sin(x) \) and \( e^x \), by way of computing infinite series with well-behaved error terms. Since floating point numbers are unable to represent all of the reals, we must approximate these functions within some error bound. As a simple version of this use case, we consider a program for calculating the geometric series \( \sum_{i=0}^{\infty} \frac{1}{x^i} \) within an error bound of \( \epsilon = \frac{\delta n}{\delta} \). We require \( x \geq 2 \) so that the series converges, which simplifies some of our assertions for this example. While this is a toy example that would be easier to compute in its closed form—the series \( \sum_{i=0}^{\infty} a \cdot r^i \) is known to converge to \( \frac{a}{1-r} \) for \( |r| < 1 \), it suffices as a simple example of using PotPie with an interesting partial specification. We cover a more realistic example in Section 4.1.3. The program we use to compute this series is as follows:
For brevity, we omit assertions outside of the pre/postcondition, loop invariant, and loop postcondition. We show wrapped Coq Props and arithmetic terms in green, i.e. \( \delta_n \cdot (x - 1) \).

Terms in black are Imp expressions. Note that we encounter the bug in our program compiler, which miscompiles the less-than macro: \( \langle a/b + c/d = (ad + cb)/(bd) \rangle \).

For (1), we will need to look at the underlying representation of our assertions. As shown in Figure 3, our precondition and postcondition actually have the following form:

\[
(\text{fun } x', \text{rn'} \rightarrow \text{rd'} \rightarrow \langle x' \wedge x' = x \wedge \delta_n \neq 0 \wedge \delta_d \neq 0 \wedge \text{rn'} = 1 \wedge \text{rd'} = x \wedge i' = 2 \rangle) \quad \forall x \ 1 \ 2
\]

Everything after the anonymous function is actually an expression in the Imp language.

For (2), we have to show that every implication in the Imp implication database is compiled to a valid implication in STACK. The implication most relevant to the successful compilation of the proof is the last one, which implies the program’s postcondition. Since the Imp loop condition \( \langle x \rangle \) gets compiled to \( \langle \rangle \) in STACK, our negated loop condition becomes

\[
\neg (\text{mult}(\#2, \delta_d \cdot (x - 1)) \cdot \text{mult}(\#5, \delta_n \cdot (x - 1)) \leq \text{mult}(\#5, \delta_d))
\]

This is equivalent to the below inequality, which still implies the compiled postcondition.

This is easily proved with Coq’s Psatz lia tactic:

\[
\text{mult}(\#5, \delta_d) < \text{mult}(\#2, \delta_d \cdot (x - 1)) \cdot \text{mult}(\#5, \delta_n \cdot (x - 1)) \equiv \frac{1}{x - 1} - \frac{\#2}{\#5} < \frac{\delta_n}{\delta_d}
\]

### 4.1.3 Square Root

The second approximation program we consider interacts with the same miscompilation and still meaningfully preserves the source specification. Given numbers \( a, b, \epsilon_a, \epsilon_d \), we consider a square root approximation program that calculates some \( x, y \) such that \( \left| x^2 - \frac{a^2}{y^2} \right| \leq \frac{\epsilon_a}{\epsilon_d^2} \). We can project the postcondition entirely into Coq terms, multiplying through both sides by the denominator so we can express it in our language. After writing the program, we come up with the following loop condition, which represents \( \frac{\epsilon_a}{\epsilon_d} < \left| \frac{x^2}{y^2} - \frac{a^2}{y^2} \right| \) (is syntactic sugar for \text{mult}, and \( \langle \rangle \) is actually the Imp less-than macro):
Correctly Compiling Proofs About Programs Without Proving Compilers Correct

Our Imp square root program and specification is given by the following.

```plaintext
loop_cond ≜ (y·y·b·ε_n < y·y·a·ε_n - x·x·a·ε_d) \lor (y·y·b·ε_n < x·x·b·ε_d - y·y·a·ε_d)
```

Most of the rules of consequence are straightforward. The only nontrivial implication involved is the final rule of consequence for the postcondition. The loop’s postcondition is

```
\neg \left( \frac{a}{c^2} < \frac{b}{d^2} \right) \equiv \frac{a}{c^2} \leq \frac{b}{d^2}
```

which directly gets us the program postcondition.

During compilation, the loop condition is miscompiled: the program compiler changes < to \leq. This results in the following target loop condition, where again, mult is represented by \cdot. Note this is not green since it represents an expression in Stack, not a Coq one.

```
stk_loop_cond ≜ #1·#1·b·ε_n \leq #1·#1·a·ε_n - #4·#4·b·ε_d
\lor #1·#1·b·ε_n \leq #4·#4·b·ε_d - #1·#1·a·ε_d
```

Compared to the target program and proof, the main difference is in the final application of the rule of consequence, where the incorrect behavior of the compiler appears and changes the semantics of the loop condition. The programs have meaningfully different semantics, and those meaningfully different semantics do manifest in the application of the while rule.

```
((T, T))
```

While the loop condition is indeed miscompiled, the postcondition uses Coq’s \leq, so the postcondition is not. Even though the unsound behavior of the compiler changes the semantics of the loop invariant, it is not enough to break the implication between the loop condition and the Coq-wrapped loop condition. Further, because of the way that the postcondition projects into Coq, the final implication is almost completely provable via applications of helper lemmas from Section 4.1.1 and the tactics inversion and Peatz.lia.

### 4.2 PotPie Three Ways

PotPie makes it easy to swap out control-flow-preserving program compilers and still reuse the same infrastructure. We instantiate PotPie with three variants of a program compiler, and use these on three small programs: shift (left-shift) \textbf{14}, max \textbf{15} (\textbf{16}), and min \textbf{17}:
1. An incomplete program compiler [18] that is missing entire cases of the source language grammar. Only shift can be compiled using the incomplete proof compiler.

2. An incorrect program compiler [19] that contains a mistake and an unsafe optimization, in a similar vein to the previous examples. We can compile max using it, but not min.

3. An unverified correct program compiler [20] that always preserves program and specification behavior. This can be used to proof compile all of the programs.

These examples show we are able to instantiate the PotPie framework for several different compilers, and PotPie is compatible with correct compilers as well. We are able to invoke the CC and Tree compilers with all of these case studies as well.

5 Implementation

While much of our proof development for PotPie is implemented in Coq, the Tree plugin is implemented in OCaml (Section 5.1). We prove that PotPie is sound for both workflows (Section 5.2) and keep PotPie’s trusted computing base small (Section 5.3).

5.1 The Tree Plugin

The Tree plugin is implemented in OCaml, and consists of about 2.2k LOC. While this is not a trivial amount of engineering, much of it consists of code that wraps Coq’s OCaml API. Additionally, such a plugin only has to be created once per target language-logic pair, and is completely independent from compilation. Indeed, the plugin can be called on any Stack Hoare tree—the tree need not be the result of compilation. While Table 1 indicates that the plugin automates a check for the commutativity equations from Section 3.2.1, this is because the properties checked by the plugin imply the commutativity equations for the included Tree proof compiler in our code [2]—it never actually checks the commutativity equations themselves. This makes Tree more flexible than the CC approach.

The plugin is called on a Stack tree, function environment, implication database (with proof of its validity), and list of functions. Here we call it on our multiplication example:

```
Certify (MultTargetTree.tree) (MultTargetTree.fenv) (ProdTargetTree.facts)
2 (MultValidFacts.valid_facts) (MultTargetTree.funcs) as mult.
```

mult contains the answer returned by the plugin. If the plugin is set to generate certificates and it is successful, mult has type stk_valid_tree. Otherwise, mult is a Coq bool.

The plugin recurses over the input tree and attempts to construct the certificate [21]. This may fail if the tree is malformed or there are mutually recursive functions. As we saw in Section 2.2, the Stack logic requires that all expressions preserve the stack, which is represented by the relation exp_stack_pure_rel [23]. However, due to the semantics of Stack functions, we need to know that all function calls preserve the stack, and showing that exp_stack_pure_rel is true in the presence of mutually recursive functions would lead to an infinite loop. If certificate generation fails, the plugin tries to provide a boolean answer as a fallback mechanism. It does this by checking each function for stack-preserving behavior modulo the behavior of other functions [23], then checking the proof tree recursively [24].

As we saw in Table 2, the certificate generator and tree checking algorithms are fairly performant. This is due to several caching and reduction algorithm optimizations we made. Before applying optimizations, the series and square root examples took >10 minutes to generate certificates, and now take <5 seconds. The main bottleneck was Coq’s \(\delta\)-reductions, which unfold constants. Our plugin provides an option to treat certain functions as “opaque”
Table 3 The proof engineering effort that went into stating and formalizing PotPie, including the infrastructure to support the code and spec languages, logics, the compilers, the case studies, and automation. Here, “specs” means the number of Definitions, Fixpoints, and Inductive. WF stands for well-formed, Insts. for instantiations of CC compilers, and Auto for automation. “Base Props” refers to code related to the base assertions seen in Figure 3.

<table>
<thead>
<tr>
<th>Category</th>
<th>Imp</th>
<th>Stack</th>
<th>Base</th>
<th>Compiler</th>
<th>Insts</th>
<th>Case Studies</th>
<th>Auto</th>
<th>Other</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOC</td>
<td>808</td>
<td>1948</td>
<td>3605</td>
<td>2593</td>
<td>1102</td>
<td>150</td>
<td>780</td>
<td>3045</td>
<td>6971</td>
</tr>
<tr>
<td>Theorems</td>
<td>15</td>
<td>67</td>
<td>103</td>
<td>91</td>
<td>44</td>
<td>2</td>
<td>17</td>
<td>93</td>
<td>288</td>
</tr>
<tr>
<td>Specs</td>
<td>43</td>
<td>32</td>
<td>51</td>
<td>29</td>
<td>25</td>
<td>14</td>
<td>13</td>
<td>40</td>
<td>100</td>
</tr>
</tbody>
</table>

inside the plugin (27), leaving their constants folded and speeding up normalization. This does not change the user’s Coq environment. The plugin also uses unification (for example, to match with constructors of option types (32)) to avoid all but one call to normalization, which we found to significantly improve performance.

5.2 Formal Proof

Our Coq formalization includes two proof of soundness, one for each of the workflows, as well as all of the case studies from Section 4. The CC soundness proof takes the form of a correct-by-construction function that takes a source Hoare proof, the well-formedness conditions, and the implication translation, and produces a verified Hoare proof in the target, as described in Section 3.1. For Tree, we prove that if all of the obligations for CC are satisfied, then the compiled tree is valid. As we mentioned in Section 3.1, we additionally show that when the OCaml plugin generates a certificate that typechecks, the certificate can be used to obtain an hl_stk proof.

We loosely based our code on Xavier Leroy’s course on mechanized semantics [29]. The lines of code (LOC) numbers for our proof development in Table 3 may be surprisingly large when compared to the size of Leroy’s course materials, but there are several key differences. First, our languages include functions, making our semantics more difficult to reason about than the course’s semantics. However, functions also give us the opportunity to reason about the composition of programs and their proofs (Section 4.1), so we stand by this decision. Second, our target language is far less well-behaved than either of the languages in the course. Third, PotPie supports two different workflows, two separate proof compilers that work to get guarantees even for incorrect compilation.

5.3 Trusted Computing Base (TCB)

PotPie’s two workflows for proof compilation have different TCBs and provide different levels of guarantees. The CC proof compiler’s TCB consisting of the Coq kernel, the mechanized semantics, the definition of the Hoare triple, and two localized Uniqueness of Identity Proofs (UIP) axioms for reasoning about the equalities between dependent types. UIP, which is consistent with Coq, states that any two equality proofs are equal for all types—we instead assume that equality proofs are equal to each other for two particular types, AbsEnvs (25) (the implementation of SM from Section 2.2) and function environments (26). This does not imply universal UIP but is similarly convenient for proof engineering. Whenever all of its proof obligations can be satisfied, the correct-by-construction proof compiler is guaranteed to produce a correct proof. However, the resulting proof object may not be independent from the source semantics, due to various opaque proof terms that cannot be further reduced.

The Tree plugin can either generate a certificate or run a check on a proof tree, returning its validity as a boolean. The certificate generator has a strictly smaller TCB than CC since
it does not assume any form of UIP. The certificate generator works by generating a term of type \texttt{stk_valid_tree} \cite{22}. Since this term must still be type-checked in Coq for it to be considered valid, this does not add to the TCB. The \texttt{Tree boolean proof tree checker} has its own “kernel,” also implemented in OCaml, for checking proof trees, which adds to its TCB. While it does not imply formal correctness, it can boost confidence in compiled proofs.

6 Related Work and Discussion

Early work on compiling proofs positioned itself as an extension of \textit{proof-carrying code} \cite{34}. A 2005 paper \cite{4} stated a theorem relating source and target program logics. Early work \cite{32} transformed Hoare-style proofs about Java-like programs to proofs about bytecode implemented in XML. Later work \cite{36} implemented \textit{proof-transforming compilation}, transforming proof objects from Eiffel to bytecode, and formalizing the specification compiler in Isabelle/HOL, with a hand-written proof of correctness of the proof compiler. Subsequent work \cite{15} showed how to embed the compiled bytecode proofs into Isabelle/HOL. Our work is the first we know of to formally verify the correctness of the proof compiler, and to use it to support specification-preserving compilation in the face of incorrect program compilation. Existing work on \textit{certificate translation} \cite{3, 25}, which is similar but focuses on compiler optimizations, may help us relax control-flow restrictions.

There is a lens through which our work is related to \textit{type-preserving compilation}: compiling programs in a way that preserves their types. There is work on this defined on a subset of Coq for CPS \cite{7} and ANF \cite{20} translations. As the source and target languages both have dependent types, this can likewise be used to compile proofs while preserving specifications. Our work focuses on compiling program logic proofs instead.

Our work implements a certified \textit{proof transformation} in Coq for an embedded program logic. Proof transformations were introduced in 1987 to bridge automation and usability \cite{38}, and have since been used for proof generalization \cite{14, 19, 16}, reuse \cite{30}, and repair \cite{40}.

The golden standard for correct compilation is \textit{certified compilation}: formally proving compilers correct. The CompCert verified C compiler \cite{28, 27} lacks bugs present in other compilers \cite{44}. The CakeML \cite{24} verified implementation of ML includes a verified compiler. Oeuf \cite{31} and CertiCoq \cite{2} are certified compilers for Coq’s term language Gallina. Certified compilation is desirable when possible, but real compilers may be unverified, incomplete, or incorrect. Our work complements certified compilation by exploring an underexplored part of the design space of compiler correctness: compilation that is \textit{specification-preserving} for a given source program and (possibly partial) specification, even when the compilation may not be fully \textit{meaning-preserving} for that program. The original CompCert paper \cite{27} brought up the possibility of specification-preserving compilation as part of a design space that is \textit{complementary} to, not in competition with, certified compilation. We agree; it expands the space of guarantees one can get for compiled programs—even when those programs are incorrectly compiled. It also expands the means by which one may get said guarantees.

Our work implements a kind of \textit{certifying compilation}: producing compiled code and a proof that its compilation is correct. For example, COGENT’s certifying compiler proves that, for a given program compiled from COGENT to C, target code correctly implements a high-level semantics embedded in Isabelle/HOL \cite{1, 41}. Certifying compilation shares the benefit that the compiler may be incorrect or incomplete, yet still produce proofs about the compiled program. Most prior work on certifying compilation that we are aware of targets general properties (like type safety) rather than program-specific ones. One exception is \textit{Rupicola} \cite{39}, a framework for correct but incomplete compilation from Gallina to low-level
code using proof search, which focuses on preservation of program-specific specifications
proven at the source level like we do. But it does not appear to address the case when the
program itself is incorrectly compiled, nor the case where there already exists an unverified
complete program compiler. Our work adds to the space of certifying compilation by
preserving program-specific partial specifications proven at the source level even when the
program itself is compiled incorrectly, with the added benefit of compositionality.

One immensely practical method for showing that programs compiled with unverified
compilers preserve behavior is translation validation. In translation validation, the
compiler produces a proof of the correctness of a particular program’s compilation, which
then needs to be checked [35]. Our work is in a similar spirit, but distinguishes itself in that
our method does not rely on functional equivalence for the particular compiled program.
Our method makes it possible to show that a compiler preserves a partial specification when
the program is miscompiled in ways that are not relevant to the specification.

Section 4.1.1 shows in a limited context our method’s potential for compositionality. Similar motivation is behind (much more mature) work in compositional certified compila-
tion [43, 13, 18]. DimSum [42] defines an elegant and powerful language-and-logic-agnostic
framework for language interoperability, though to get guarantees, it leans heavily on data
refinement arguments that show a simulation property stronger than what our framework
requires. We hope that in the future, we will make our compositional workflow more sys-
tematic and fill the gap of compositional multi-language reasoning in a relaxed correctness
setting—by linking compiled proofs directly in a common target logic. Similar motivations
are behind linking types [37], which are extensions to type systems for reasoning about
correct linking in a multilanguage setting. We expect tradeoffs similar to those between our
work and type-preserving compilation to arise in this setting.

Frameworks based on embedded program logics (e.g., Iris [17, 23], VST-Floyd [8],
Bedrock [11, 12], YNot [33], CHL [10], SepRef [26], and CFML [9]) help proof engineers
write proofs in a proof assistant about code with features that the proof assistant lacks. C
programs verified in the VST program logic are, by composition with CompCert, guaranteed
to preserve their specifications even after compilation to assembly code [5]. Our work aims to
create an alternative toolchain for preserving guarantees across compilation that allows the
program compiler to be unverified or even incorrect, even for the program being compiled.
Relative to practical frameworks like Iris and VST, the program logics we use for this are
much less mature. We hope to extend our work to more practical logics and lower-level
target languages in the future, so that users of toolchains like VST can get guarantees about
compiled programs even in the face of incorrect compilation.

7 Conclusion

We showed how compiling proofs across program logics can empower proof engineers to
reason directly about source programs yet still obtain proofs about compiled programs—even
when they are incorrectly compiled. Our implementation PotPie and its two workflows, CC
and Tree, are formally verified in Coq, providing guarantees that compiled proofs not only
prove their respective specifications, but also are correctly related to the source proofs. Our
hope is to provide an alternative to relying on verified program compilers without sacrificing
important correctness guarantees of program specifications.

Future Work In this work, we have not tackled the problem of control flow optimizations.
We believe the challenges of bridging abstraction levels and verifying control flow-modifying
optimizations are mostly orthogonal, and that the latter is out of our scope. In future work,
we would like to investigate ways our work could be composed with control flow optimizations. For example, we may be able to leverage Kleene algebras with tests (KAT) [21] to reason about control flow optimizations. An optimization pass could extract a proof subtree and return the optimized subprogram, while preserving semantic equality via KAT. This approach may even be able to leverage a Hoare triple’s preconditions to apply optimizations that would be otherwise unsound [22]. For an example of KATs applied to existing compiler optimizations, see existing work [21]. Beyond relaxing control flow restrictions, other next steps include supporting more source languages and logics, supporting additional linking of target-level proofs, implementing optimizing compilers, and bringing the benefits of proof compilation to more practical frameworks.

References


23:18  Correctly Compiling Proofs About Programs Without Proving Compilers Correct


