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

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.

* Co-first authors

To see an example of where correct compilation becomes too strict, consider a Hoare triple $\{0 \leq a \wedge 0 \leq \epsilon\}\ y := 42; x := \text{source\_sqrt}(a)\ \{|a - x^2| \leq \epsilon\}$, which says that after setting y to 42 and calling source_sqrt on a, the variable x stores a square root approximation of a within ϵ. Suppose that source_sqrt is compiled to some program target_sqrt such that if $0 \leq a \wedge 0 \leq \epsilon$, then after 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 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 $\{0 \leq a \wedge 0 \leq \epsilon\} x := \text{source\_sqrt}(a) \{|x^2 - a| \leq \epsilon\}$ 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.
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*

$$
\begin{array}{rcl}
a & ::= & \mathbb{N} \mid x \mid \text{param } k \mid a + a \mid a - a \mid f(a, \ldots, a) \\
b & ::= & T \mid F \mid \neg b \mid a \leq a \mid b \wedge b \mid b \vee b \\
i & ::= & \text{skip} \mid x := a \mid i; i \\
 & \mid & \text{if } b \text{ then } i \text{ else } i \mid \text{while } b \text{ do } i \\
\lambda & ::= & (f, k, i, \text{return } x) \\
p & ::= & (\{\lambda, \ldots, \lambda\}, i)
\end{array}
\qquad
\begin{array}{rcl}
a & ::= & \mathbb{N} \mid \#k \mid a + a \mid a - a \mid f(a, \ldots, a) \\
b & ::= & T \mid F \mid \neg b \mid a \leq a \mid b \wedge b \mid b \vee b \\
i & ::= & \text{skip} \mid \text{push} \mid \text{pop} \mid \#k := a \mid i; i \\
 & \mid & \text{if } b \text{ then } i \text{ else } i \mid \text{while } b \text{ do } i \\
\lambda & ::= & (f, k, i, \text{return } a\ n) \\
p & ::= & (\{\lambda, \ldots, \lambda\}, i)
\end{array}
$$

Figure 1 Imp (left) and Stack (right) syntax, where a describes arithmetic expressions, b boolean expressions, i imperative statements, λ function definitions, and p whole programs, which consist of a set of functions and a "main" body. The evaluation of the main body yields the result of program. For Imp functions, (f,k,i,return x) is a function named f with k parameters that returns the value of the variable x after executing the function body, i. For Stack functions ($f, k, i,$ return a n), we return the result of evaluating a after executing the body i, and then pop n indices from the stack.

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 (42) to relevant locations in our code.

2.1 Programs

Our languages Imp and Stack are both simple imperative languages that are similar in syntax (Figure 1) yet have differing memory models. Imp has an abstract environment with two components: a mapping of identifiers to their `nat` values, and function parameters, which are accessed param k construct, whereas Stack has a single function call stack, where new variables are pushed to the low indices and stack indices are accessed with the $\#k$ construct. Function calls in Imp are always mutation-free since functions are limited to their (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 and stacks (4) so that "sound translation" is a well-defined concept.

▶ **Definition 1.** *Let V be a finite set of variable names, and let $\varphi : V \to \{1, \ldots, |V|\}$ be bijective with inverse φ^{-1}. Then for all variable stores σ, parameter stores Δ, and stacks σ_s, we say that σ and Δ are φ-equivalent to σ_s, written $(\sigma, \Delta) \approx_\varphi \sigma_s$, 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

$$\begin{array}{rlrl}
\text{comp}_\text{a}^\varphi(n) &\triangleq n & \text{comp}_\text{b}^\varphi(T) &\triangleq T \quad \text{comp}_\text{b}^\varphi(F) \triangleq F \\
\text{comp}_\text{a}^\varphi(\text{param } k) &\triangleq \#(|V| + k + 1) & \text{comp}_\text{b}^\varphi(\neg b) &\triangleq \neg\text{comp}_\text{b}^\varphi(b) \\
\text{comp}_\text{a}^\varphi(a_1 \text{ op } a_2) &\triangleq \text{comp}_\text{a}^\varphi(a_1) \text{ op } \text{comp}_\text{a}^\varphi(a_2) & \text{comp}_\text{b}^\varphi(b_1 \text{ op } b_2) &\triangleq \text{comp}_\text{b}^\varphi(b_2) \text{ op } \text{comp}_\text{b}^\varphi(b_2) \\
\text{comp}_\text{a}^\varphi(f(a_1, \ldots, a_n)) &\triangleq f(\text{comp}_\text{a}^\varphi(a_1), \ldots, \text{comp}_\text{a}^\varphi(a_n)) & \text{comp}_\text{b}^\varphi(a_1 \leq a_2) &\triangleq \text{comp}_\text{a}^\varphi(a_1) \leq \text{comp}_\text{a}^\varphi(a_2)
\end{array}$$

Figure 2 An arithmetic expression compiler comp_a (left) and a boolean expression compiler comp_b (right). op stands for the appropriate binary operators: $+$ and $-$, and $\wedge$ and $\vee$, respectively

$$\begin{array}{l}
M ::= T \mid F \mid p_n\ [e, \ldots, e] \\
\qquad \mid\ M \wedge M \mid M \vee M
\end{array}
\qquad
\frac{}{\sigma \vDash T}\ \textsc{True}
\qquad
\frac{map_eval_\sigma\ [a_i]_1^n\ [v_i]_1^n \qquad p_n\ v_\text{list}}{\sigma \vDash p_n\ [a_1, \ldots, a_n]}\ \text{N-ARY}$$

Figure 3 Syntax (left) and semantics (right) for base assertions for both Imp and Stack. map_eval_σ is a relation from lists of expressions to lists of values. The semantic interpretation is parametric over the types of v, σ, and map_eval_σ. Interpretations for $\wedge$ and $\vee$ are standard.

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 *expressions* from now on) in Figure 2. This infrastructure is a straightforward extension of the variable mapping function φ 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 `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:

$$SM ::= SM_e \mid SM \wedge SM \mid SM \vee SM \qquad TM ::= (n, TM_e) \mid TM \wedge TM \mid TM \vee TM \quad (1)$$

where SM_e and TM_e are instances of the logic described in Figure 3 using Imp and Stack arithmetic and boolean expressions respectively.

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:

$$\frac{|\sigma| \geq n \qquad \sigma \vDash TM_e}{\sigma \vDash (n, TM_e)}\ \textsc{Stack Base}$$

We made the decision to allow function calls within specifications. This is not essential to

$$\mathrm{comp}^{\varphi,k}_{\mathrm{spec}}(T) \triangleq (k, T) \qquad \mathrm{comp}^{\varphi,k}_{\mathrm{spec}}(p_n\ (e_1, \ldots, e_n)) \triangleq (k, p_n\ (\mathrm{comp}^{\varphi}_{\mathrm{expr}}(e_1), \ldots, \mathrm{comp}^{\varphi}_{\mathrm{expr}}(e_n)))$$
$$\mathrm{comp}^{\varphi,k}_{\mathrm{spec}}(F) \triangleq (k, F) \qquad \mathrm{comp}^{\varphi,k}_{\mathrm{spec}}(SM_1 \mathrm{\ op\ } SM_2) \triangleq \mathrm{comp}^{\varphi,k}_{\mathrm{spec}}(SM_1) \mathrm{\ op\ } \mathrm{comp}^{\varphi,k}_{\mathrm{spec}}(SM_2)$$

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

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 $\varphi : V \to \{1, \ldots, |V|\}$ 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 φ 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:

▶ **Definition 2.** *I is valid if for each pair (P, Q) 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 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 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.

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.

		Tree						CC	
		Create PC	Invoke		Guaranteeing Properties			Create PC	Invoke PC
			PC	Plugin	Trees WF	Valid Tree	CGC		
Comp.	Comm.	×	-	-	A	A	-	P	-
User	Spec DB	-	×	P	×	P	-	-	P
	Pre/Post	-	×	×	×	×	P	-	P
	Imp WF	-	×	×	-	-	-	-	P
	preservesStack	-	×	×	A	A	-	-	P

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 $\mathrm{comp}^{\varphi,k}_{\mathrm{spec}}$ is sound with respect to P if for all σ, Δ, σ_s such that $(\sigma, \Delta) \approx_\varphi \sigma_s$, we have $\sigma, \Delta \models P \Leftrightarrow \sigma_s \models \mathrm{comp}^{\varphi,k}_{\mathrm{spec}}(P)$.*

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 $\{comp\ P\}(comp\ c)\{comp\ Q\}$.*

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 `hl_stk` (the Stack correct-by-construction Hoare proof type) from a term of type `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 (2) and a

Coq plugin, implemented in OCaml ⑤, that checks the proof tree's validity ⑥. 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 ⑫. 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 ⑬:

```
Theorem valid_tree_can_construct_hl_stk
    (P Q: AbsState) (i: imp_stack) (facts': implication_env_stk)
    (fenv': fun_env_stk) (T: stk_hoare_tree):
        ∀ (V: stk_valid_tree P i Q facts' fenv' T), (* certificate type*)
            hl_stk P i Q facts' fenv'.
```

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 (`hl_Imp_Lang`) along with the CC proof obligations (described in Section 3.2), CC produces a Stack Hoare proof (`hl_stk`) of the same property ① (some detail is omitted for brevity):

```
Definition proof_compiler :
    ∀ (P Q: AbsEnv) (i: imp_Imp_Lang) (fenv: fun_env) (facts: implication_env)
    (var_to_stack_map: list string) (num_args: nat)
    (proof: hl_Imp_Lang P i Q facts fenv) (translate_facts: valid_imp_trans_def),
        (* well-formedness conditions and specification translation soundness *) →
        hl_stk (comp P) (comp i) (comp Q) (comp facts) (comp fenv).
```

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-compilable Imp programs and specifications satisfy the 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 $P' = \mathrm{comp}^{\varphi,k}_{\mathrm{spec}}(P)$:

$$\mathrm{comp}^{\varphi,k}_{\mathrm{pf}}\left(\{P[\mathtt{x} \to \mathtt{a}]\}\ \mathtt{x}\ \mathtt{:=}\ \mathtt{a}\ \{P\}\right) = \left\{P'[\varphi(x) \to \mathrm{comp}^{\varphi,k}_{\mathrm{code}}(a)]\right\}\ \varphi(\mathtt{x})\mathtt{:=}\ \mathtt{a}\ \left\{P'\right\}$$

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

$$\text{comp}^{\varphi,k}_{\text{spec}}(P[x \to a]) = (\text{comp}^{\varphi,k}_{\text{spec}}(P))[\varphi(x) \to \text{comp}^{\varphi}_{\text{a}}(a)] \tag{2}$$

$$\text{comp}^{\varphi,k}_{\text{spec}}((p_1\ [b]) \wedge P) = \big(k + |V|, (p_1\ [\text{comp}^{\varphi}_{\text{b}}(b)]) \wedge \text{comp}^{\varphi,k}_{\text{spec}}(P)\big) \tag{3}$$

$$\text{comp}^{\varphi,k}_{\text{code}}(x := a) = \#\varphi(x) := \text{comp}^{\varphi}_{\text{a}}(a) \tag{4}$$

$$\text{comp}^{\varphi,k}_{\text{code}}(\text{skip}) = \text{skip} \tag{5}$$

$$\text{comp}^{\varphi,k}_{\text{code}}(i_1; i_2) = \text{comp}^{\varphi,k}_{\text{code}}(i_1); \text{comp}^{\varphi,k}_{\text{code}}(i_2) \tag{6}$$

$$\text{comp}^{\varphi,k}_{\text{code}}(\text{if } b \text{ then } i_1 \text{ else } i_2) = \text{if } \text{comp}^{\varphi}_{\text{b}}(b) \text{ then } \text{comp}^{\varphi,k}_{\text{code}}(i_1) \text{ else } \text{comp}^{\varphi,k}_{\text{code}}(i_2) \tag{7}$$

$$\text{comp}^{\varphi,k}_{\text{code}}(\text{while } b \text{ do } i) = \text{while } \text{comp}^{\varphi}_{\text{b}}(b) \text{ do } \text{comp}^{\varphi,k}_{\text{code}}(i) \tag{8}$$

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

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 ⑦ (this is the "Spec DB" proof obligation in Table 1):

▶ **Definition 5.** *Given φ, k, and a function environment, an* Imp *implication $P \Rightarrow Q$ has a valid translation if for all σ, Δ, σ_s, if $(\sigma, \Delta) \approx_\varphi \sigma_s$, then $\sigma_s \models \text{comp}^{\varphi,k}_{\text{spec}}(P) \Rightarrow \text{comp}^{\varphi,k}_{\text{spec}}(Q)$.*

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.

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.

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.

	Multiplication				Exponentiation				Series				Square Root			
	Core	Tree	TreeC	CC	Core	Tree	TreeC	CC	Core	Tree	TreeC	CC	Core	Tree	TreeC	CC
LOC	209	104	56	508	478	107	54	362	679	174	45	630	406	154	43	286
Theorems	3	1	2	28	9	1	2	26	14	1	2	48	6	1	2	29
Tree CG (s)	0.172				0.154				2.781				4.279			
Tree Check (s)	0.131				0.098				0.534				1.946			

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 $<$ to $\leq$:

$$\mathrm{comp}^{\varphi}_{\mathrm{badb}}(a_1 < a_2) \triangleq \mathrm{comp}^{\varphi}_{\mathrm{a}}(a_1) \leq \mathrm{comp}^{\varphi}_{\mathrm{a}}(a_2)$$

$\mathrm{comp}_{\mathrm{badb}}$ 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 $a_1 \leq a_2 \wedge \neg(a_1 \leq a_2 \wedge a_2 \leq a_1)$. For simplicity, we will use $<$ in this paper. The resulting program compiler ⑧ 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):

```
{ ⊤ }
x := param 0; y := 0;
while (1 ≤ x) do
  y := y + param 1;
  x := x - 1;
{ y = (param 0) · (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 ⑨ that relates applications of the Imp multiplication function to Coq's `Nat.mul`:

```
Lemma mult_aexp_wrapper a1 a2 n1 n2: a1 ⇓ n1 → a2 ⇓ n2 → mult(a1, a2) ⇓ (n1 * 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 ⑩:

```
Lemma exp_aexp_wrapper : forall a1 a2 n1 n2, a1 ⇓ n1 → a2 ⇓ n2 → exp(a1, a2) ⇓ n2^n1.
```

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 $\sum_{i=1}^{\infty} \frac{1}{x^i}$ within an error bound of $\epsilon = \frac{\delta_n}{\delta_d}$. 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:

```
{ 2 ≤ x ∧ x = x ∧ δn ≠ 0 ∧ δd ≠ 0 ∧ 1 = 1; ∧x = x ∧ 2 = 2 }
  x  := x; // the series denominator
  rn := 1; // the result numerator
  rd := x; // the result denominator (for i = 1)
  i  := 2; // the next exponent
  { rn · x^i − rn · x^(i−1) = rd · x^(i−1) − rd ∧ x = x ∧ 2 ≤ x ∧ 2 ≤ i } // loop invariant
  // the loop condition is equivalent to ε < 1/(x−1) − rn/rd, and 1/(x−1) = Σ_{i=1}^{∞} 1/x
  while (mult(rn, δd · (x − 1)) + mult(rd, δn · (x − 1)) < mult(rd, δd)) do
    d  := exp(x, i);
    rn := frac_add_numerator(rn, rd, 1, d); // a/b + c/d = (ad + cb)/(bd)
    rd := frac_add_denominator(rd, d); // fraction addition denominator
    i  := i + 1;
{ ¬ (mult(rn, δd · (x − 1)) + mult(rd, δn · (x − 1)) < mult(rd, δd))
  ∧ (rn · x^i − rn · x^(i−1) = rd · x^(i−1) − rd ∧ x = x ∧ 2 ≤ x ∧ 2 ≤ i) } // loop postcondition
{ δd · rd ≤ δn · (x − 1) · rd + δd · (x − 1) · rn } // program postcondition: 1/(x−1) − rn/rd ≤ δn/δd
```

For brevity, we omit assertions outside of the pre/postcondition, loop invariant, and loop postcondition. We show wrapped Coq `Prop`s 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 `<` in the while loop conditional. However, we are still able to compile this program and its proof to Stack because (1) the pre/postconditions' meaning is preserved by compilation, and (2) the implication database is still valid, i.e., every compiled Imp implication is still an implication in Stack.

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

`(fun x' rn' rd' i' =>` $2 \leq$ `x'` $\wedge$ `x'` $= x \wedge \delta_n \neq 0 \wedge \delta_d \neq 0 \wedge$ `rn'` $= 1 \wedge$ `rd'` $= x \wedge$ `i'` $= 2$`)` $x\ 1\ x\ 2$

`(fun rn' rd' =>` $\delta_d \cdot$ `rd'` $\leq \delta_n \cdot (x-1) \cdot$ `rd'` $+ \delta_d \cdot (x-1) \cdot$ `rn'`$)$ `rn rd`

Everything after the anonymous function is actually an expression in the Imp language. These are the only parts of the assertions that are compiled by the specification compiler. For instance, x is a constant arithmetic expression in Imp, which wraps Coq's `nat` type. The arithmetic compiler, comp_a, from Figure 2 compiles these to `nat` constants in the Stack language. For the variables `rn` and `rd`, $\text{comp}_\text{a}^{\varphi,k}(\texttt{rn}) = \#\varphi(\texttt{rn})$. After compiling, we get the postcondition $\delta_d \cdot$ `#5` $\leq \delta_n \cdot (x-1) \cdot$ `#5` $+ \delta_d \cdot (x-1) \cdot$ `#2`, or symbolically: $\frac{1}{x-1} - \frac{\#2}{\#5} \leq \frac{\delta_n}{\delta_d}$.

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 `<` gets compiled to `<=` in Stack, our negated loop condition becomes

$$\neg\ \texttt{(mult(\#2,}\ \delta_d \cdot (x-1)\texttt{) + mult(\#5,}\ \delta_n \cdot (x-1)\texttt{)} \leq \texttt{mult(\#5,}\ \delta_d\texttt{))}$$

This is equivalent to the below inequality, which still implies the compiled postcondition. This is easily proved with Coq's `Psatz.lia` tactic.

$$\texttt{mult(\#5,}\ \delta_d\texttt{)} < \texttt{mult(\#2,}\ \delta_d \cdot (x-1)\texttt{) + mult(\#5,}\ \delta_n \cdot (x-1)\texttt{)} \quad \equiv \frac{1}{x-1} - \frac{\#2}{\#5} < \frac{\delta_n}{\delta_d}$$

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 $a, b, \epsilon_n, \epsilon_d$, we consider a square root approximation program that calculates some x, y such that $|\frac{x^2}{y^2} - \frac{a}{b}| \leq \frac{\epsilon_n}{\epsilon_d}$. 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 $\frac{\epsilon_n}{\epsilon_d} < \left|\frac{x^2}{y^2} - \frac{a}{b}\right|$ ($\cdot$ is syntactic sugar for `mult`, and $<$ is actually the Imp less-than macro):

$$\texttt{loop\_cond} \triangleq (y \cdot y \cdot b \cdot \epsilon_n < y \cdot y \cdot a \cdot \epsilon_d - x \cdot x \cdot a \cdot \epsilon_d) \vee (y \cdot y \cdot b \cdot \epsilon_n < x \cdot x \cdot b \cdot \epsilon_d - y \cdot y \cdot a \cdot \epsilon_d)$$

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

```
{⊤}
  x     := a;  y     := mult(2, b);
  inc_n := a;  inc_d := mult(2, b);
  while (loop_cond) do
    inc_d := mult(2, inc_d);
    if (mult(mult(y, y), mult(a, ϵd)) ≤ mult(mult(x, x), mult(b, ϵd)))
    then x := frac_sub_numerator(x, y, inc_n, inc_d);
    else x := frac_add_numerator(x, y, inc_n, inc_d);
    y := frac_add_denominator(y, inc_d);
{ ¬loop_condition ∧ ⊤ } ⟹
{ ((x · x · b · ϵd) − (y · y · a · ϵd) ≤ y · y · b · ϵn) ∧ ((y · y · a · ϵd) − (x · x · b · ϵd) ≤ y · y · b · ϵn) }
```

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 $\neg\left(\frac{\epsilon_n}{\epsilon_d} < \left|\frac{x^2}{y^2} - \frac{a}{b}\right|\right) \equiv \left|\frac{x^2}{y^2} - \frac{a}{b}\right| \leq \frac{\epsilon_n}{\epsilon_d}$, 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.

$$\begin{aligned}\texttt{stk\_loop\_cond} \triangleq\ & \#1 \cdot \#1 \cdot b \cdot \epsilon_n \leq \#1 \cdot \#1 \cdot a \cdot \epsilon_d - \#4 \cdot \#4 \cdot b \cdot \epsilon_d \\ \vee\ & \#1 \cdot \#1 \cdot b \cdot \epsilon_n \leq \#4 \cdot \#4 \cdot b \cdot \epsilon_d - \#1 \cdot \#1 \cdot a \cdot \epsilon_d\end{aligned}$$

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.

```
{(⊤, ⊤)}
  push; push; push; push;
  #4 := a; #1 := mult(2, b);
  #3 := a; #2 := mult(2, b);
  {4, ⊤}
  while (stk_loop_cond) do
    #2 := mult(2, #2);
    if (mult(mult(#1, #1), mult(a, ϵd)) ≤ mult(mult(#4, #4), mult(a, ϵd)))
    then #4 := frac_sub_numerator(#4, #1, #3, #2);
    else #4 := frac_add_numerator(#4, #1, #3, #2);
    #1 := frac_add_denominator(#1, #2)
{(4, ¬target_loop_condition)) /\ (4, ⊤)} ⟹
{4, (#4·#4·b·ϵd) − (#1·#1·a·ϵd) ≤ (#1·#1·b·ϵn) ∧ ((#1·#1·a·ϵd) − (#4·#4·b·ϵd) ≤ #1·#1·b·ϵn)}
```

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 `Psatz.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) (14), `max` (15) (16), and `min` (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. While we are able to invoke CC compilers with all of these case studies, the Tree certificate generator fails for `max` due to a plugin bug, though the plugin's tree validity checker does work and returns `true`.

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:

```
1 Certify (MultTargetTree.tree) (MultTargetTree.fenv) (ProdTargetTree.facts)
2         (MultValidFacts.valid_facts) (MultTargetTree.funcs) as mult.
3 Check 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` ③. 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 δ-reductions, which unfold constants. Our plugin provides an option to treat certain functions as "opaque" 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 ① takes the form of a correct-by-construction function that takes a source Hoare proof, the well-formedness

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 `Definition`s, `Fixpoint`s, and `Inductive`s. 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.

Category	Imp			Stack			Base	Compiler				Insts.	Case	Auto	Other	Total
	Lang	Logic	WF	Lang	Logic	WF	Props	Code	Spec	Tree	CC		Studies			
LOC	808	1948	3605	2593	1077	5635	941	1102	159	780	3045	2133	6971	2914	3225	36936
Theorems	15	67	103	91	17	204	37	44	2	17	93	52	288	31	105	1166
Specs	43	32	51	29	51	63	31	25	14	13	40	100	238	50	107	887

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 (12). As we mentioned in Section 3.1, we additionally show that when the OCaml plugin (5) 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*, `AbsEnv`s (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 `stk_valid_tree` (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 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 **proof-carrying code** [34]. A 2005 paper [4] stated a theorem relating source and target program logics. Early work [32]

transformed Hoare-style proofs about Java-like programs to proofs about bytecode implemented in XML. Later work [36] implemented **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 [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 **certificate translation** [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 **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 [7] and ANF [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 **proof transformation** in Coq for an embedded program logic. Proof transformations were introduced in 1987 to bridge automation and usability [38], and have since been used for proof generalization [14, 19, 16], reuse [30], and repair [40].

The golden standard for correct compilation is **certified compilation**: formally proving compilers correct. The CompCert verified C compiler [28, 27] lacks bugs present in other compilers [44]. The CakeML [24] verified implementation of ML includes a verified compiler. Oeuf [31] and CertiCoq [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 **specification-preserving** for a given source program and (possibly partial) specification, even when the compilation may not be fully **meaning-preserving** for that program. The original CompCert paper [27] brought up the possibility of specification-preserving compilation as part of a design space that is *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 **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 [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 *Rupicola* [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 compilation [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 systematic 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

1. Sidney Amani, Alex Hixon, Zilin Chen, Christine Rizkallah, Peter Chubb, Liam O'Connor, Joel Beeren, Yutaka Nagashima, Japheth Lim, Thomas Sewell, Joseph Tuong, Gabriele Keller, Toby Murray, Gerwin Klein, and Gernot Heiser. Cogent: Verifying high-assurance file system implementations. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 175–188, Atlanta, GA, USA, April 2016. doi:10.1145/2872362.2872404.
2. Abhishek Anand, Andrew W. Appel, Greg Morrisett, Matthew Weaver, Matthieu Sozeau, Olivier Savary Belanger, Randy Pollack, and Zoe Paraskevopoulou. CertiCoq: A verified compiler for Coq. In *CoqPL*, 2017. URL: http://www.cs.princeton.edu/~appel/papers/certicoq-coqpl.pdf.
3. Gilles Barthe, Benjamin Grégoire, César Kunz, and Tamara Rezk. Certificate translation for optimizing compilers. *ACM Trans. Program. Lang. Syst.*, 31(5), jul 2009. doi:10.1145/1538917.1538919.
4. Gilles Barthe, Tamara Rezk, and Ando Saabas. Proof obligations preserving compilation. In *Formal Aspects in Security and Trust: Thrid International Workshop, FAST 2005, Newcastle upon Tyne, UK, July 18-19, 2005, Revised Selected Papers*, pages 112–126. Springer, 2005.
5. Lennart Beringer and Andrew W. Appel. Abstraction and subsumption in modular verification of C programs. *Formal Methods in System Design*, 58(1):322–345, October 2021. doi:10.1007/s10703-020-00353-1.
6. Sandrine Blazy and Xavier Leroy. Mechanized Semantics for the Clight Subset of the C Language. *Journal of Automated Reasoning*, 43(3):263–288, October 2009. doi:10.1007/s10817-009-9148-3.
7. William J. Bowman, Youyou Cong, Nick Rioux, and Amal Ahmed. Type-preserving cps translation of and types is not not possible. *Proc. ACM Program. Lang.*, 2(POPL), dec 2017. doi:10.1145/3158110.
8. Qinxiang Cao, Lennart Beringer, Samuel Gruetter, Josiah Dodds, and Andrew W. Appel. VST-Floyd: A Separation Logic Tool to Verify Correctness of C Programs. *Journal of Automated Reasoning*, 61(1):367–422, June 2018. doi:10.1007/s10817-018-9457-5.
9. Arthur Charguéraud. Characteristic formulae for the verification of imperative programs. *SIGPLAN Not.*, 46(9):418–430, sep 2011. doi:10.1145/2034574.2034828.
10. Haogang Chen, Daniel Ziegler, Tej Chajed, Adam Chlipala, M. Frans Kaashoek, and Nickolai Zeldovich. Using crash hoare logic for certifying the fscq file system. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, page 18–37, New York, NY, USA, 2015. Association for Computing Machinery. doi:10.1145/2815400.2815402.
11. Adam Chlipala. Mostly-automated verification of low-level programs in computational separation logic. *SIGPLAN Not.*, 46(6):234–245, jun 2011. doi:10.1145/1993316.1993526.
12. Adam Chlipala. The bedrock structured programming system: Combining generative metaprogramming and hoare logic in an extensible program verifier. *SIGPLAN Not.*, 48(9):391–402, sep 2013. doi:10.1145/2544174.2500592.
13. Ronghui Gu, Jérémie Koenig, Tahina Ramananandro, Zhong Shao, Xiongnan (Newman) Wu, Shu-Chun Weng, Haozhong Zhang, and Yu Guo. Deep specifications and certified abstraction layers. *SIGPLAN Not.*, 50(1):595–608, jan 2015. doi:10.1145/2775051.2676975.
14. Robert W Hasker and Uday S Reddy. Generalization at higher types. In *Proceedings of the Workshop on the λProlog Programming Language*, pages 257–271, 1992.
15. Bruno Hauser. Embedding proof-carrying components into Isabelle. Master's thesis, ETH, Swiss Federal Institute of Technology Zurich, Institute of Theoretical . . . , 2009.
16. Einar Broch Johnsen and Christoph Lüth. Theorem reuse by proof term transformation. In *Theorem Proving in Higher Order Logics: 17th International Conference, TPHOLs 2004, Park City, Utah, USA, September 14-17, 2004. Proceedings*, pages 152–167. Springer, Berlin, Heidelberg, 2004. doi:10.1007/978-3-540-30142-4_12.

17 Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. Iris: Monoids and invariants as an orthogonal basis for concurrent reasoning. *SIGPLAN Not.*, 50(1):637–650, jan 2015. doi:10.1145/2775051.2676980.

18 Jérémie Koenig and Zhong Shao. Compcerto: Compiling certified open c components. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2021, page 1095–1109, New York, NY, USA, 2021. Association for Computing Machinery. doi:10.1145/3453483.3454097.

19 Thomas Kolbe and Christoph Walther. *Proof Analysis, Generalization and Reuse*, pages 189–219. Springer, Dordrecht, 1998. doi:10.1007/978-94-017-0435-9_8.

20 Paulette Koronkevitch, Ramon Rakow, Amal Ahmed, and William J. Bowman. Anf preserves dependent types up to extensional equality. *Journal of Functional Programming*, 32:e12, 2022. doi:10.1017/S0956796822000090.

21 Dexter Kozen and Maria-Cristina Patron. Certification of compiler optimizations using kleene algebra with tests. In John Lloyd, Veronica Dahl, Ulrich Furbach, Manfred Kerber, Kung-Kiu Lau, Catuscia Palamidessi, Luís Moniz Pereira, Yehoshua Sagiv, and Peter J. Stuckey, editors, *Computational Logic — CL 2000*, pages 568–582, Berlin, Heidelberg, 2000. Springer Berlin Heidelberg. doi:10.1007/3-540-44957-4_38.

22 Dexter Kozen and Jerzy Tiuryn. On the completeness of propositional hoare logic. *Information Sciences*, 139(3):187–195, 2001. Relational Methods in Computer Sciece. URL: https://www.sciencedirect.com/science/article/pii/S0020025501001645, doi:10.1016/S0020-0255(01)00164-5.

23 Robbert Krebbers, Amin Timany, and Lars Birkedal. Interactive proofs in higher-order concurrent separation logic. *SIGPLAN Not.*, 52(1):205–217, jan 2017. doi:10.1145/3093333.3009855.

24 Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. CakeML: A verified implementation of ML. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '14, pages 179–191, New York, NY, USA, 2014. ACM. doi:10.1145/2535838.2535841.

25 César Kunz. *Certificate Translation Alongside Program Transformations*. PhD thesis, ParisTech, Paris, France, 2009.

26 Peter Lammich. Refinement to imperative HOL. *J. Autom. Reason.*, 62(4):481–503, apr 2019. doi:10.1007/s10817-017-9437-1.

27 Xavier Leroy. Formal certification of a compiler back-end, or: programming a compiler with a proof assistant. In *33rd ACM symposium on Principles of Programming Languages*, pages 42–54. ACM Press, 2006. URL: http://xavierleroy.org/publi/compiler-certif.pdf.

28 Xavier Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, July 2009. doi:10.1145/1538788.1538814.

29 Xavier Leroy. Coq development for the course "Mechanized semantics", 2019-2021. URL: https://github.com/xavierleroy/cdf-mech-sem.

30 Nicolas Magaud. Changing data representation within the Coq system. In *International Conference on Theorem Proving in Higher Order Logics*, pages 87–102. Springer, 2003.

31 Eric Mullen, Stuart Pernsteiner, James R. Wilcox, Zachary Tatlock, and Dan Grossman. Oeuf: Minimizing the Coq extraction TCB. In *CPP*, pages 172–185, New York, NY, USA, 2018. ACM. doi:10.1145/3167089.

32 Peter Müller and Martin Nordio. Proof-transforming compilation of programs with abrupt termination. In *SAVCBS*, pages 39–46, 01 2007. doi:10.1145/1292316.1292321.

33 Aleksandar Nanevski, Greg Morrisett, Avraham Shinnar, Paul Govereau, and Lars Birkedal. Ynot: Dependent types for imperative programs. *SIGPLAN Not.*, 43(9):229–240, sep 2008. doi:10.1145/1411203.1411237.

34 George C. Necula. Proof-carrying code. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '97, pages 106–119, New York, NY, USA, January 1997. Association for Computing Machinery. doi:10.1145/263699.263712.

35 George C. Necula. Translation validation for an optimizing compiler. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, PLDI '00, page 83–94, New York, NY, USA, 2000. Association for Computing Machinery. doi:10.1145/349299.349314.

36 Martin Nordio, Peter Müller, and Bertrand Meyer. Formalizing proof-transforming compilation of Eiffel programs. Technical report, ETH Zurich, 2008.

37 Daniel Patterson and Amal Ahmed. Linking Types for Multi-Language Software: Have Your Cake and Eat It Too. In Benjamin S. Lerner, Rastislav Bodík, and Shriram Krishnamurthi, editors, *2nd Summit on Advances in Programming Languages (SNAPL 2017)*, volume 71 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 12:1–12:15, Dagstuhl, Germany, 2017. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. URL: https://drops.dagstuhl.de/opus/volltexte/2017/7125, doi:10.4230/LIPIcs.SNAPL.2017.12.

38 Frank Pfenning. *Proof Transformations in Higher-Order Logic*. PhD thesis, Carnegie Mellon University, 1987.

39 Clément Pit-Claudel, Jade Philipoom, Dustin Jamner, Andres Erbsen, and Adam Chlipala. Relational compilation for performance-critical applications: Extensible proof-producing translation of functional models into low-level code. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2022, page 918–933, New York, NY, USA, 2022. Association for Computing Machinery. doi:10.1145/3519939.3523706.

40 Talia Ringer. *Proof Repair*. PhD thesis, University of Washington, 2021.

41 Christine Rizkallah, Japheth Lim, Yutaka Nagashima, Thomas Sewell, Zilin Chen, Liam O'Connor, Toby Murray, Gabriele Keller, and Gerwin Klein. A framework for the automatic formal verification of refinement from Cogent to C. In *Interactive Theorem Proving*, pages 323–340, Cham, 2016. Springer. doi:10.1007/978-3-319-43144-4_20.

42 Michael Sammler, Simon Spies, Youngju Song, Emanuele D'Osualdo, Robbert Krebbers, Deepak Garg, and Derek Dreyer. Dimsum: A decentralized approach to multi-language semantics and verification. *Proc. ACM Program. Lang.*, 7(POPL), jan 2023. doi:10.1145/3571220.

43 Yuting Wang, Pierre Wilke, and Zhong Shao. An abstract stack based approach to verified compositional compilation to machine code. *Proc. ACM Program. Lang.*, 3(POPL), jan 2019. doi:10.1145/3290375.

44 Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in c compilers. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, page 283–294, New York, NY, USA, 2011. Association for Computing Machinery. doi:10.1145/1993498.1993532.