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

Just as one can embed domain-specific languages into general-purpose host languages, one can embed programming languages and program logics into the host language of an interactive proof assistant like Coq. Such an embedding can comprise a language definition, a program logic for program specifications, and frameworks for effective reasoning about programs. We can then use the host language to prove metatheoretic properties about the embedded language like the soundness of the program logic or the correctness of a compiler to a different target language. We can also use the host language to reason about individual programs by having the embedded program logic use features of the host language as appropriate. In such a workflow, it is typical to combine a formal proof of compiler correctness, a formal proof of program-logic soundness, and a proof using the program logic that a program in the embedded language meets a specification. Together, this ensures, under the usual adequacy assumptions, that the compiled program meets the same specification.

This paper explores a variation of this workflow that allows the program compiler to be *unverified* or even sometimes *incorrect*, but still provides strong guarantees about compiled programs. Our workflow involves compiling embedded programs, specifications, and proof objects all at once, from an embedded source language and logic to an embedded target language and logic. We implement such a proof compiler in Coq, moving from an imperative language with variables and functions, to a language where the only memory is a single call stack. We show how this proof compiler produces proofs about target-level programs for four variants of a program compiler: one that is incomplete, one that is incorrect, one that is correct but unverified, and one that is correct and verified. Our proof compiler is formally proven sound in Coq—it is constructive and correct by construction, with its type specifying the relation between the source and target proofs. We demonstrate how our approach enables strong target-program guarantees without a compiler-correctness proof, opening up new options for which proof burdens one might shoulder instead of, or in addition to, compiler correctness.

Additional Key Words and Phrases: program logics, formal verification, compiler correctness

ACM Reference Format:
Audrey Seo*, Chris Lam*, Dan Grossman, and Talia Ringer. 2018. Correctly Compiling Proofs About Programs Without Proving Compilers Correct. In *Woodstock '18: ACM Symposium on Neural Gaze Detection, June 03–05, 2018, Woodstock, NY*. ACM, New York, NY, USA, 28 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

Proof assistants like Coq and Isabelle/HOL make it possible to prove programs correct directly. But proof engineers who wish to reason about programs written in langauges with features that the proof assistant lacks, like effects or concurrency, may turn to frameworks that embed program

* Co-first authors

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
Conference acronym 'XX, June 03–05, 2018, Woodstock, NY
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/XXXXXXX.XXXXXXX

logics within the proof assistant, like Iris [Jung et al. 2015; Krebbers et al. 2017], VST [Cao et al. 2018], CHL [Chen et al. 2015], or Sepref [Lammich 2019]. By chaining these program logics with a verified compiler, they can get strong guarantees about compiled programs [Cao et al. 2018].

This workflow crucially depends on a verified compiler—and a verified compiler must actually be correct. But real compilers remain very difficult to prove correct. In fact, real compilers are often *not* correct, whether it is due to bugs, known incomplete functionality, or "optimizations" that may be known to be incorrect in some rare or unimportant scenarios (like gcc's `-ffast-math` [Byrne 2023]). Without a proof of compiler correctness, proofs about programs in the source language provide *no guarantees* about programs in the target language.

In this paper, we explore a different approach to getting strong guarantees about compiled programs that allows the program compiler to be unverified or even sometimes incorrect. Our approach is to compile embedded programs, specifications, and proofs all at once—in the style of certificate translation [Kunz 2009] and proof-transforming compilation [Hauser 2009; Müller and Nordio 2007; Nordio et al. 2008]. This approach transforms proofs about source programs directly to proofs about their compiled counterparts. Furthermore, it does so in a way that preserves the original specification, up to changes in the abstraction level. With this approach, we can prove that compilation of specific programs preserves those programs' specifications even without a proof of compiler correctness. This is in the spirit of translation validation [Necula 2000], but with a focus on preserving the source program's specification and producing a target-language proof that the specification is still met, even in cases when the program may be incorrectly compiled.

To that end, this paper presents a formally verified proof compiler in Coq, which we call PotPie (Proof Object Transformation Preserving Imp Embeddings). PotPie compiles embedded programs, specifications, and proofs all at once, moving from an imperative language with variables and functions, to a language where the only memory is a single call stack. PotPie is constructive and correct by construction, guaranteeing that whenever the proof compiler can be invoked, the compiled proofs preserve the original specification up to the change in abstraction level—all without relying on a proof of program compiler correctness. Rather, if the program compiler is incorrect in a way that is relevant to preserving a particular specification, then PotPie cannot be invoked.

Our contributions are threefold:

(1) We present PotPie, a proof compiler (Section 2) that compiles embedded programs, specifications, and proofs (Sections 3 and 4) in a way that preserves specifications without relying on full program compiler correctness.
(2) We demonstrate PotPie on case studies to compile proofs about programs that have been compiled using four different variants of a program compiler: an incomplete program compiler, an incorrect program compiler, an unverified correct program compiler, and a verified correct program compiler (Section 5).
(3) We prove PotPie sound, both informally (Section 6) and formally in Coq (Section 7). PotPie is, to the best of our knowledge, the first formally verified proof compiler across program logics. (Section 8 discusses other projects, including those that did proof compilation but with only an informal proof of proof compiler correctness.)

The PotPie Coq formalization will be available in the associated artifact.[1]

2 OVERVIEW

PotPie compiles source programs, specifications, and proofs to target programs, specifications, and proofs. It does so soundly, meaning that whenever the proof compiler can be invoked, the compiled proof proves the compiled specification about the compiled program, and the compiled specification

[1]Coming soon.

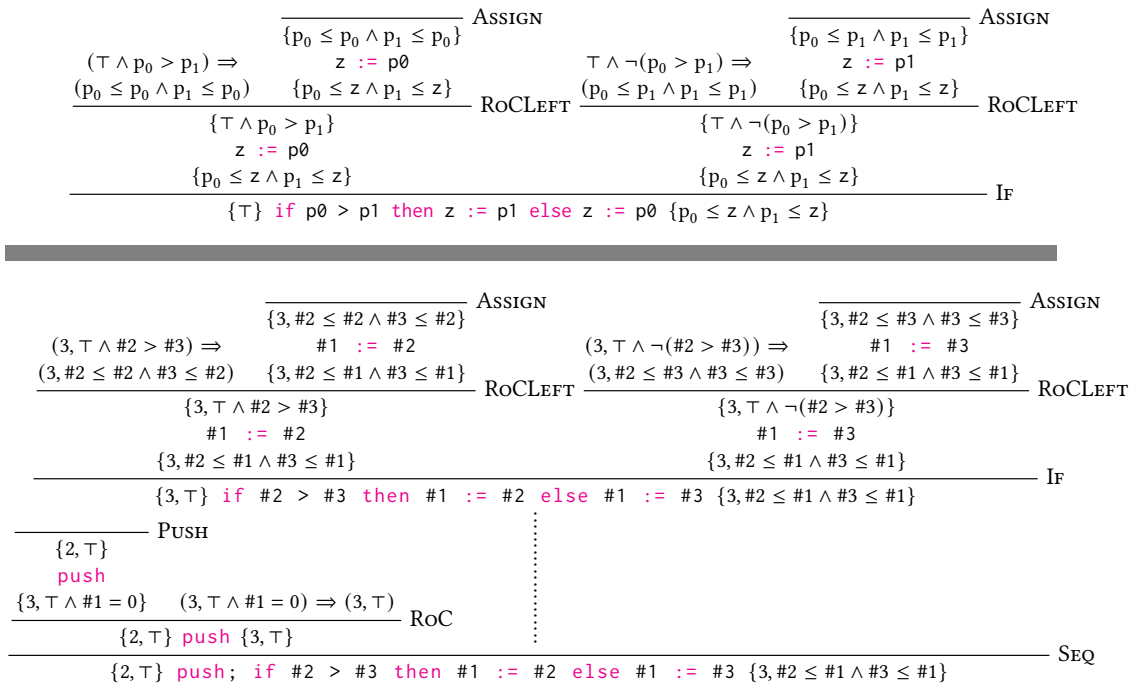

Fig. 1. The Hoare tree for the program `max` (above) and its compiled version (below). We use p_i and `pi` as shorthand for the i^{th} parameter in the source specification and code, respectively. In the compiled proof, assertions have the form (n, P) where n is the minimum stack size required. In the target language, we write #n to to mean the n^{th} stack index.

is the same as the source specification up to the change in abstraction from the source language to the target language. For simplicity, in this section, we demonstrate this with a *correct* program compiler. For examples of correctly compiled proofs with an *incorrect* compiler, see Section 5.

Our source and target languages are called Imp and Stack. While both are imperative, Imp uses variables (which are function-scoped) and a distinct construct for accessing function parameters. Stack, on the other hand, maintains only a single stack for storing values, so the compiler must map variables and parameters to stack positions and manage the stack size (via push and pop commands) to compile function calls. Due to this change in abstraction, their specification languages (which encode Hoare triple pre and post conditions) are also different. For more details, see Section 3.

To demonstrate end-to-end proof compilation, we consider a program `max` that computes the maximum of two numbers, its specification, and a proof that the program meets the specification:

```
1  {⊤}
2  if (param 0 > param 1) then
3    z := param 0
4  else
5    z := param 1
6  {param 0 ≤ z ∧ param 1 ≤ z}
```

This program takes in two parameters (`param 0` and `param 1`), and sets the variable `z` to the maximum of its arguments. The program's precondition, ⊤ (True), indicates that there are no requirements on its input. The program's postcondition, `param 0` ≤ `z` ∧ `param 1` ≤ `z`, indicates `z` is greater than or equal to both of the inputs. This is a partial specification for this function.

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

Fig. 2. Imp (left) and Stack (right) language 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. In the syntax for Imp functions, (f,k,i,return x) is a function named f taking k parameters that returns the value stored in the variable x after executing the function body, i. For Stack functions ($f, k, i,$ return $a\ n$), we instead return the result of evaluating a after executing the body i, and then pop n positions from the stack.

$$
\boxed{\langle\, \sigma, \Lambda, \Delta, a \,\rangle \Downarrow_a c}
$$

$$
\frac{\begin{array}{c}\Lambda(f) = (f, k, i, y) \\ \forall j \leq k, \langle \sigma, \Lambda, \Delta, a_j \rangle \Downarrow_a c_j \\ \langle \varnothing, \Lambda, [c_1, \ldots, c_k], i \rangle \Downarrow_i \sigma' \\ \sigma'(y) = c_y\end{array}}{\langle \sigma, \Lambda, \Delta, f(a_1, a_2, a_3, \ldots, a_k) \rangle \Downarrow_a c_y}\ \textsc{Fun}
\qquad
\frac{\sigma(x) = c}{\langle \sigma, \Lambda, \Delta, x \rangle \Downarrow_a c}\ \textsc{Var}
\qquad
\frac{\begin{array}{c}|\Delta| \geq i \\ \Delta[i] = c\end{array}}{\langle \sigma, \Lambda, \Delta, \text{param } i \rangle \Downarrow_a c}\ \textsc{Param}
$$

Fig. 3. Big step semantics for the relevant cases of the Imp language.

We represent the proof of this specification explicitly as a proof object that corresponds to a Hoare tree (Figure 1, top). Our goal is to compile this program, specification, and proof object from the source language and logic to the target language and logic. The code compiler $\text{comp}_{\text{code}}$ is just a conventional compiler that makes memory-layout decisions (i.e., where parameters and variables go on the stack). The specification compiler $\text{comp}_{\text{spec}}$ needs to know the behavior of the code compiler so that the target specifications use the correct stack positions and thus the right Hoare triples can be produced. The proof compiler comp_{pf} is then just a recursive traversal over the source proof, using the code and specification compilers in the natural way. This results in Figure 1, bottom, which proves the analogous specification about the compiled program.

3 PROGRAMS, SPECIFICATIONS, AND PROOFS

This section presents our six languages, with Section 3.1 describing programming languages, Section 3.2 describing specification languages, and Section 3.3 describing proof languages.

3.1 Programs

We designed Imp and Stack to be similar enough to facilitate building a formally verified proof compiler in a single pass, yet still have interesting properties and changes in abstraction that are reminiscent of real-world source and target languages. Imp and Stack have similar syntax (Figure 2), except for variables vs. stack slots (denoted with the syntax $\#k$ for index k, where the newest stack variables have the least index) and the addition of push and pop in Stack. Furthermore, parameters (param i) in Imp cannot be assigned to.

The basic infrastructure for both our Imp and Stack language semantics is based on the materials for Xavier Leroy's course on mechanized semantics [Leroy 2021]. Taking inspiration from his approach, we define a big step semantics for our languages. Our semantics judgments for Imp are

$$\boxed{\langle\, \sigma, \Lambda, a \,\rangle \Downarrow_a (\, \sigma', c \,)}$$

$$\frac{1 \le k \le |\sigma| \qquad \sigma[k] = c}{\langle \sigma, \Lambda, \#k \rangle \Downarrow_a (\sigma, c)}\ \textsc{StkVar} \qquad \frac{\langle \sigma, \Lambda, a_1 \rangle \Downarrow_a (\sigma', c_1) \qquad \langle \sigma', \Lambda, a_2 \rangle \Downarrow_a (\sigma'', c_2)}{\langle \sigma, \Lambda, a_1 + a_2 \rangle \Downarrow_a (\sigma'', c_1 + c_2)}\ \textsc{StkAdd}$$

$$\frac{\begin{array}{l}\Lambda(f) = (f, k, i, \text{return } a_{\texttt{ret}}\ n) \quad \forall j \le k, \langle \sigma_{j-1}, \Lambda, a_j \rangle \Downarrow_a (\sigma_j, c_j) \\ \sigma' = [c_1, \ldots, c_k] : \sigma_k \quad \langle \sigma', \Lambda, i \rangle \Downarrow_i \sigma'' \quad \langle \sigma'', \Lambda, a_{\texttt{ret}} \rangle \Downarrow_a (\sigma''', c_{\texttt{ret}}) \quad \sigma_f = pop(\sigma''', k + n)\end{array}}{\langle \sigma_0, \Lambda, f(a_1, a_2, a_3, \ldots, a_k) \rangle \Downarrow_a (\sigma_f, c_{\texttt{ret}})}\ \textsc{StkFun}$$

$$\boxed{\langle\, \sigma, \Lambda, i \,\rangle \Downarrow_i \sigma'}$$

$$\frac{}{\langle \sigma, \Lambda, \text{push} \rangle \Downarrow_i [0] : \sigma}\ \textsc{StkPush} \qquad \frac{}{\langle [v] : \sigma, \Lambda, \text{pop} \rangle \Downarrow_i \sigma}\ \textsc{StkPop}$$

Fig. 4. Big step semantics for the relevant cases of the Stack language. Here, σ is a stack and Λ is the function environment. We denote concatenation by $\sigma_1 : \sigma_2$, and stack length by $|\sigma|$.

of the form $\langle \sigma, \Lambda, \Delta, e \rangle \Downarrow r$, where σ is the variable environment, Λ is the function environment, and Δ contains the values of parameters. The result, r, depends on the type of e: it is another variable environment if e is a statement, a boolean value if e is a boolean expression, and a natural number if e is an arithmetic expression. Figure 3 contains the inference rules for Imp that are either non-standard or are different from Stack's semantics. Note we allow for function calls inside of arithmetic expressions. Functions are call-by-value, and the function body (a command statement) is then evaluated on an empty variable environment with the evaluated arguments as its parameter environment. Lastly, when a call completes, it returns the value stored in the function's return variable in the local environment. These semantics are formalized in Coq.

For Stack, our semantics judgments have the form $\langle \sigma, \Lambda, e \rangle \Downarrow r$, where Λ is again the function environment, but σ is a stack. As in Imp, the type of r depends on the type of e. If e is a statement, r is a stack. But if e is an arithmetic expression, r is a *pair* of a stack and a natural number, since arithmetic expressions in Stack can have side effects (similarly for booleans). Figure 4 contains the inference rules for Stack that are either non-standard, or that are different from Imp's semantics. Stack adds several complexities—instead of having an abstract environment where any variable can be queried and used, variables are stored on a stack that can be directly manipulated via pushing, popping, and assigning. Further, instead of each function evaluating in its own private environment, function calls involve pushing variables and arguments onto the stack and later popping them. The Stack semantics are also formalized in Coq. Even though in this paper, we usually deal with programs that do not modify memory outside of their own stack frame, there is no semantic restriction that prevents this.

Our program logics follow the same basic syntactic structure for assertions over their respective languages (Figure 5, left). Here, p_n is an n-ary predicate over expressions, which we denote e. We offer support for predicates of any length by allowing predicates that take lists of expressions. This specific structure enables two important aspects of our system:

(1) It makes the logic compilation very easy. In order to compile these assertions, all that has to be done is to compile the expressions.
(2) It permits the user of PotPie to be as expressive as they wish, without adding new relations or operators to the languages. For example, we do not provide a multiply operator, but a user could write a specification using multiplication by placing the multiplication application within the Coq proposition of an n-ary predicate.

$$M ::= T \mid F \mid p_n\ [e, \ldots, e] \mid M \wedge M \mid M \vee M$$

$$\boxed{\sigma \vDash M}$$

$$\frac{}{\sigma \vDash T}\ \textsc{True} \qquad \frac{map\_eval_\sigma\ a_{list}\ v_{list} \quad p_{\text{list}}\ v_{\text{list}}}{\sigma \vDash p_{\text{list}}\ a_{\text{list}}}\ \textsc{N-ary}$$

Fig. 5. 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.

The semantics for assigning a truth value to a formula (Figure 5, right) parametrize predicates over the value types. For example, if we have the assertion $p_1\ a$ where a is a language 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`.

3.2 Specifications

Using this approach, we can define a program logic SM for the source language by assuming predicates for all the arithmetic (e.g., +) and boolean operations (e.g., <) in the source language in Figure 5, and then adding conjunction and disjunction at the logic level. We can define TM for the target language similarly. We then use this to construct the following specification grammars:

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

where SM_e and TM_e are instantiations of the logic described in Figure 5 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:

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

3.3 Proofs

Both Imp and Stack use a Hoare logic as their program logic, and we have proven them sound in Coq.[2] Our Imp logic is largely standard (Figure 6; see Hoare [1969] for more), besides the fact that we split the rule of consequence into two separate rules (one for weakening and one for strengthening), as well as the fact that all applications of these two rules must draw from an *implication database* which is a list of pairs of specifications satisfying the following definition:

Definition 3.1. An implication database I is valid if, for each pair of implications (P, Q) in I, $\forall \sigma, \sigma \vDash P \Rightarrow \sigma \vDash Q$.

This implication database serves three purposes: (1) to identify which implications must be preserved through compilation, (2) to ensure we only need to compile each implication once in case it is used multiple times in a proof, and (3) to make it easy to identify which source implication corresponds to which target implication across compilation.

The Stack proof rules (Figure 7) are similar to the Imp ones, except for obvious language differences (stack index accesses instead of variables, the addition of Push and Pop). Further, the

[2]In our Coq implementation, these are defined as inductive types that project the logics into `Type`, because we need to access, match over, and transform the contents of the proof in order to compile it.

$$\frac{}{\{P\}\ \text{skip}\ \{P\}}\ \textsc{ImpSkip} \qquad \frac{}{\{P[x \to a]\}\ x := a\ \{P\}}\ \textsc{ImpAssign} \qquad \frac{\{P\}\ i_1\ \{R\} \quad \{R\}\ i_2\ \{Q\}}{\{P\}\ i_1; i_2\ \{Q\}}\ \textsc{ImpSeq} \qquad \boxed{\{P\}\ c\ \{Q\}}$$

$$\frac{\{P \wedge b\}\ i_1\ \{Q\} \quad \{P \wedge \neg b\}\ i_2\ \{Q\}}{\{P\}\ \text{if } b \text{ then } i_1 \text{ else } i_2\ \{Q\}}\ \textsc{ImpIf} \qquad \frac{\{P \wedge b\}\ i\ \{P\}}{\{P\}\ \text{while } b \text{ do } i\ \{P \wedge \neg b\}}\ \textsc{ImpWhile}$$

$$\frac{(P_1, P_2) \in I \quad P_1 \Rightarrow P_2 \quad \{P_2\}\ i\ \{Q\}}{\{P_1\}\ i\ \{Q\}}\ \textsc{ImpRoCLeft} \qquad \frac{(Q_1, Q_2) \in I \quad \{P\}\ i\ \{Q_1\} \quad Q_1 \Rightarrow Q_2}{\{P\}\ i\ \{Q_2\}}\ \textsc{ImpRoCRight}$$

Fig. 6. Hoare deduction rules for the Imp language, where RoC stands for "rule of consequence." I here is the implication database satisfying Definition 3.1.

$$\frac{}{\{A\}\ \text{skip}\ \{A\}}\ \textsc{StkSkip} \qquad \frac{\text{preservesStack}(b, n) \quad \{n, P \wedge b\}\ i\ \{n, P\}}{\{n, P\}\ \text{while } b \text{ do } i\ \{n, P \wedge \neg b\}}\ \textsc{StkWhile} \qquad \boxed{\{n, P\}\ c\ \{m, Q\}}$$

$$\frac{}{\{n, P\}\ \text{push}\ \{n+1, \text{inc}(P) \wedge \#1 = 0\}}\ \textsc{StkPush} \qquad \frac{}{\{(n+1, \text{inc}(P) \wedge Q)\}\ \text{pop}\ \{(n, P)\}}\ \textsc{StkPop}$$

$$\frac{\text{preservesStack}(a, n)}{\{n, P[\#k \to a]\}\ \#k := a\ \{n, P\}}\ \textsc{StkAssign} \qquad \frac{\{n_1, P\}\ i_1\ \{n_2, Q\} \quad \{n_2, Q\}\ i_2\ \{n_3, R\}}{\{n_1, P\}\ i_1; i_2\ \{n_3, R\}}\ \textsc{StkSeq}$$

$$\frac{\text{preservesStack}(b, n_1) \quad \{n_1, P \wedge b\}\ i_1\ \{n_2, Q\} \quad \{n_1, P \wedge \neg b\}\ i_2\ \{n_2, Q\}}{\{n_1, P\}\ \text{if } b \text{ then } i_1 \text{ else } i_2\ \{n_2, Q\}}\ \textsc{StkIf}$$

$$\frac{P' \Rightarrow P \quad \{P\}\ i\ \{C\} \quad Q \Rightarrow Q'}{\{P'\}\ i\ \{Q'\}}\ \textsc{StkRoC}$$

Fig. 7. Hoare rules for the Stack language. The left and right rules of consequence are omitted, as they are identical to the Imp rules. Stack has access to the full rule of consequence, as, being the target language, it itself is not being compiled, and therefore the rule of consequence does not present a problem.

validity of the implication database is defined the same as in Definition 3.1. There is one other major way that the Stack rules differ from their Imp counterparts: the preservesStack premise in the Assign, If, and While cases, which specifies that output stack after the expression is evaluated must be the same as the input stack. This is a simplifying assumption that we add to avoid reasoning about how arbitrary functions can change the stack, since in the Stack semantics for function calls (Figure 4), there are no guarantees about effects. Function calls could affect the entire stack, so even the addition rule is not pure. This can present a welter of issues for how to track changes introduced by expressions, as well as how to alter the Hoare logic rules to keep them sound.

For any expression e, preservesStack e says that all function call subexpressions of e do not modify the rest of the stack frame. Because preserving stacks is dependent on the function environment,

the preservesStack predicate is implicitly parametric on a function environment. Since function calls are the only source of effects in expressions, we have the following lemma (proven in Coq):

Lemma 3.2. *If for* Stack *expression* e *we have* preservesStack(e)*, then for all stacks* σ, σ'*, if* $\langle \sigma, \Lambda, e \rangle \Downarrow (\sigma', v)$ *for some value* v*, we must have* $\sigma = \sigma'$*.*

While this assumption is fine for programs with function calls that do not change the stack, if we want a more general program logic to reason about the target language we need a more robust system to reason about both memory usage (à la separation logic) and behavior within assertions that may change the state. As our focus is on the approach of compiling proofs to preserve guarantees in the face of unverified or incorrect compilation, we defer this to later work.

4 COMPILATION

The meat of the PotPie approach is compiling programs, specifications, and proofs all at once to get strong guarantees at the target level, even when the program compiler is sometimes incorrect. This section describes how we bridge the abstraction gap from Imp to Stack (Section 4.1) to compile programs (Section 4.2), specifications (Section 4.3), and proofs (Section 4.4). We will define soundness and prove this approach sound in Section 6.

4.1 Bridging the Abstraction Gap

All of our compilers have one thing in common: they must bridge the abstraction gap from Imp to Stack. To do this, one must define an equivalence between stacks and variable environments in order for "sound translation" to be a well defined concept. All of our compilers begin by defining such an equivalence. Note that this equivalence is entirely dependent on our choice of a mapping between variables and stack slots. Intuitively, 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, the parameters should be "after" (i.e., appended to the end of) the local variables. In other words:

Definition 4.1. 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

- for $1 \le i \le |V|$, we have $\sigma_s[i] = \sigma(\varphi^{-1}(i))$, and
- for $|V| + 1 \le i \le |V| + |\Delta|$, we have $\sigma_s[i] = \Delta[i - |V|]$.

Note that this implies $|V| + |\Delta| \le |\sigma_s|$ while saying nothing about stack indices beyond $|V| + |\Delta|$. This will be important for translating properties about functions to the target level of abstraction.

4.2 Compiling Programs

Since the PotPie approach parameterizes over the program compiler, we define several variants of a program compiler in Section 5, capturing both incorrect and correct program compilation. All build on common infrastructure to translate Imp arithmetic and boolean expressions (Figure 8). This infrastructure is a straightforward extension of the variable mapping function φ from Definition 4.1.

4.3 Compiling Specifications

Once we have a program compiler and $\varphi : V \to \{1, \ldots, |V|\}$, defining a specification compiler is relatively simple: 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 Imp expressions within the specification, which is given as the maximum value of φ plus k, where k is the number of arguments.

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

Fig. 8. The basic infrastructure needed to define a program compiler from Imp to Stack: an arithmetic expression compiler comp_{a} (left) and a boolean expression compiler comp_{b} (right).

$$\begin{aligned}
\text{comp}_{\text{spec}}^{\varphi,k}(T) &\triangleq (k, T) \\
\text{comp}_{\text{spec}}^{\varphi,k}(F) &\triangleq (k, F) \\
\text{comp}_{\text{spec}}^{\varphi,k}(p_n\ (e_1, \ldots, e_n) &\triangleq (k, p_n\ (\text{comp}_{\text{expr}}^{\varphi}(e_1), \ldots, \text{comp}_{\text{expr}}^{\varphi}(e_n))) \\
\text{comp}_{\text{spec}}^{\varphi,k}(SM_1 \wedge SM_2) &\triangleq \text{comp}_{\text{spec}}^{\varphi,k}(SM_1) \wedge \text{comp}_{\text{spec}}^{\varphi,k}(SM_2) \\
\text{comp}_{\text{spec}}^{\varphi,k}(SM_1 \vee SM_2) &\triangleq \text{comp}_{\text{spec}}^{\varphi,k}(SM_1) \vee \text{comp}_{\text{spec}}^{\varphi,k}(SM_2)
\end{aligned}$$

Fig. 9. The specification compiler $\text{comp}_{\text{spec}}^{\varphi,k}(SM)$, which is parameterized over an expression compiler $\text{comp}_{\text{expr}}^{\varphi}$ (a subset of a program compiler).

We define the specification compiler in Figure 9. Note that this definition is parameterized over an expression compiler—a subset of a program compiler. This expression compiler need not be fully correct. But when it is *not* fully correct, for the proof compiler to be sound, we must ensure that the compiled specifications are still reasonable. That is, for any given φ, k, and an assertion P, we want to be able tell when $\text{comp}_{\text{spec}}^{\varphi,k}$ compiles P in a sound manner. In this context, "sound" means that if a variable and parameter environment satisfy the formula P, then we want the translation of those environments to also satisfy $\text{comp}_{\text{spec}}^{\varphi,k}(P)$ and vice versa.[3]

In order to invoke the proof compiler at all, we require that the specification compiler is sound with regards to the precondition and the postcondition of the source proof (we define this formally in Section 6, as part of our proof of correctness). This ensures that the produced proof actually proves the analogous property in the target language, even when the program compiler is incorrect.

4.4 Compiling Proofs

The proof compiler is defined recursively over the Hoare proof data type, parameterizing over the program and specification compilers (Figure 10). For the proof compiler to work, the program compilers must commute with the syntactic specification transformations of the Hoare rule. For example, consider the substitution performed by the assignment rule. Given P, in order to compile an application of the assignment rule, we want the following equality to hold:

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

[3]The reason for the if and only if here is to avoid situations like those where all specifications are compiled to true, which technically leads to a valid Hoare proof that proves nothing.

$$\mathrm{comp}_{\mathrm{pf}}^{\varphi,k}\left(\frac{}{\{P\}\ \mathrm{skip}\ \{P\}}\right) = \frac{}{\left\{\mathrm{comp}_{\mathrm{spec}}^{\varphi,k}(P)\right\}\ \mathrm{skip}\ \left\{\mathrm{comp}_{\mathrm{spec}}^{\varphi,k}(P)\right\}}$$

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

$$\mathrm{comp}_{\mathrm{pf}}^{\varphi,k}\left(\frac{\{P\}\ i_1\ \{P'\} \quad \{P'\}\ i_2\ \{P''\}}{\{P\}\ i_1;i_2\ \{P''\}}\right) = \frac{\mathrm{comp}_{\mathrm{pf}}^{\varphi,k}(\{P\}\ i_1\ \{P'\}) \quad \mathrm{comp}_{\mathrm{pf}}^{\varphi,k}(\{P'\}\ i_2\ \{P''\})}{\left\{\mathrm{comp}_{\mathrm{spec}}^{\varphi,k}(P)\right\}\ \mathrm{comp}_{\mathrm{code}}^{\varphi,k}(i_1;i_2)\ \left\{\mathrm{comp}_{\mathrm{spec}}^{\varphi,k}(P'')\right\}}$$

$$\mathrm{comp}_{\mathrm{pf}}^{\varphi,k}\left(\frac{\{P\wedge b\}\ i_1\ \{Q\} \quad \{P\wedge\neg b\}\ i_2\ \{Q\}}{\{P\}\ \mathrm{if}\ b\ \mathrm{then}\ i_1\ \mathrm{else}\ i_2\ \{Q\}}\right) = \frac{\mathrm{comp}_{\mathrm{pf}}^{\varphi,k}(\{P\wedge b\}\ i_1\ \{Q\}) \quad \mathrm{comp}_{\mathrm{pf}}^{\varphi,k}(\{P\wedge\neg b\}\ i_2\ \{Q\})}{\left\{\mathrm{comp}_{\mathrm{spec}}^{\varphi,k}(P)\right\}\ \mathrm{if}\ \mathrm{comp}_{\mathrm{b}}^{\varphi}(b)\ \mathrm{then}\ \mathrm{comp}_{\mathrm{code}}^{\varphi,k}(i_1)\ \mathrm{else}\ \mathrm{comp}_{\mathrm{code}}^{\varphi,k}(i_2)\ \left\{\mathrm{comp}_{\mathrm{spec}}^{\varphi,k}(Q)\right\}}$$

$$\mathrm{comp}_{\mathrm{pf}}^{\varphi,k}\left(\frac{\{P\wedge b\}\ i\ \{P\}}{\{P\}\ \mathrm{while}\ b\ \mathrm{do}\ i\ \{P\wedge\neg b\}}\right) = \frac{\mathrm{comp}_{\mathrm{pf}}^{\varphi,k}(\{P\wedge b\}\ i\ \{P\})}{\left\{\mathrm{comp}_{\mathrm{spec}}^{\varphi,k}(P)\right\}\ \mathrm{while}\ \mathrm{comp}_{\mathrm{b}}^{\varphi}(b)\ \mathrm{do}\ \mathrm{comp}_{\mathrm{code}}^{\varphi,k}(i))\ \left\{\mathrm{comp}_{\mathrm{spec}}^{\varphi,k}(P)\wedge\neg\mathrm{comp}_{\mathrm{b}}^{\varphi}(b)\right\}}$$

$$\mathrm{comp}_{\mathrm{pf}}^{\varphi,k}\left(\frac{(P,P')\in I \quad P\Longrightarrow P' \quad \{P'\}\ i\ \{Q\}}{\{P\}\ i\ \{Q\}}\right) = \frac{\mathrm{comp}_{\mathrm{spec}}^{\varphi,k}(P)\Rightarrow\mathrm{comp}_{\mathrm{spec}}^{\varphi,k}(P') \quad \mathrm{comp}_{\mathrm{pf}}^{\varphi,k}(\{P'\}\ i\ \{Q\})}{\left\{\mathrm{comp}_{\mathrm{spec}}^{\varphi,k}(P)\right\}\ \mathrm{comp}_{\mathrm{code}}^{\varphi,k}(i)\ \left\{\mathrm{comp}_{\mathrm{spec}}^{\varphi,k}(Q)\right\}}$$

Fig. 10. The proof compiler, given by $\mathrm{comp}_{\mathrm{pf}}$, is defined recursively on the structure of an Imp Hoare proof tree (see Figure 6). We omit the compilation of ImpRoCRight, since it is similar to ImpRoCLeft.

$$\mathrm{comp}_{\mathrm{spec}}^{\varphi,k}(P[x\to a]) = (\mathrm{comp}_{\mathrm{spec}}^{\varphi,k}(P))[\varphi(x)\to\mathrm{comp}_{\mathrm{a}}^{\varphi}(a)] \tag{1}$$

$$\mathrm{comp}_{\mathrm{spec}}^{\varphi,k}((p_1\ [b])\wedge P) = \left(k+|V|, (p_1\ [\mathrm{comp}_{\mathrm{b}}^{\varphi}(b)])\wedge\mathrm{comp}_{\mathrm{spec}}^{\varphi,k}(P)\right) \tag{2}$$

$$\mathrm{comp}_{\mathrm{code}}^{\varphi,k}(x := a) = \#\varphi(x) := \mathrm{comp}_{\mathrm{a}}^{\varphi}(a) \tag{3}$$

$$\mathrm{comp}_{\mathrm{code}}^{\varphi,k}(\mathrm{if}\ b\ \mathrm{then}\ i_1\ \mathrm{else}\ i_2) = \mathrm{if}\ \mathrm{comp}_{\mathrm{b}}^{\varphi}(b)\ \mathrm{then}\ \mathrm{comp}_{\mathrm{code}}^{\varphi,k}(i_1)\ \mathrm{else}\ \mathrm{comp}_{\mathrm{code}}^{\varphi,k}(i_2) \tag{4}$$

$$\mathrm{comp}_{\mathrm{code}}^{\varphi,k}(\mathrm{while}\ b\ \mathrm{do}\ i) = \mathrm{while}\ \mathrm{comp}_{\mathrm{b}}^{\varphi}(b)\ \mathrm{do}\ \mathrm{comp}_{\mathrm{code}}^{\varphi,k}(i) \tag{5}$$

Fig. 11. Equations the specification and program compilers need to satisfy on a concrete application to a proof in addition to preserving control flow in order to apply a proof compiler using this framework. (Equations 3-5 ensure that the code compiler actually invokes the arithmetic and boolean compilers.)

If we have this equality, then we can convert an instance of a source constructor to an instance of a target constructor. More formally, we can exactly define the following case, where $P' = \mathrm{comp}_{\mathrm{spec}}^{\varphi,k}(P)$:

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

More generally, as long as the code compiler commutes with these Hoare-rule induced syntactic transformations and complies with the purity conditions of the target program logic, the proof compiler can mechanically invoke the target Hoare rules and produce a target-level Hoare proof through structural induction. The full set of equations required to instantiate this framework on a specific instance of a proof can be found in Figure 11, where these equations need only be proven for each specific instance of the specifications contained within the concrete proof as opposed

Table 1. The four different compiler variants, and, for each variant, which of three guarantees about three different source programs we are able to preserve down to the target level using PotPie, and where the user proof burden is centered. Note that verifying the correct compiler has no impact on which guarantees we can preserve—rather, it simply shifts the user's proof burden so that more of it is upfront.

	Shift (Complete Spec.)	**Max (Partial Spec.)**	**Min (Partial Spec.)**	**Proof Burden**
Incomplete	✓			Per-Proof
Incorrect		✓		Per-Proof
Unverified Correct	✓	✓	✓	Per-Proof
Verified Correct	✓	✓	✓	More Upfront

to a general compiler property. As such, so long as the program compilers are executable, these conditions should be dispatchable with simple reflexivity tactics after computation, as equality should be decidable and all compiler computation on Imp terms should reduce to Stack terms.

While most of the proof compiler is fairly straightforward, the primary difficulty for soundness in the face of unsound program compilers is introduced by the rule of consequence. We discuss the difficulties this introduces and how we handle them in Section 6.

5 CASE STUDIES: POTPIE FOUR WAYS

This section builds on the translation from Section 4 to define four different variants of a program compiler, each corresponding to a different use case for proof compilation:

(1) First, we consider an **incomplete program compiler** that is missing entire cases of the source language grammar. Despite this compiler being incomplete, we can still invoke the proof compiler to get guarantees at the target level, so long as the programs we consider do not include those cases (Section 5.1).
(2) We then consider an **incorrect program compiler** that contains a mistake and an unsafe optimization. We can sometimes invoke this proof compiler to get guarantees at the target level even when the program compiles incorrectly, so long as the (possibly incomplete) specification is preserved (Section 5.2).
(3) We fix the bugs to get an **unverified correct program compiler** that always preserves program and specification behavior. We can always invoke this to get guarantees at the target level for any source program and proof (Section 5.3).
(4) Finally, we prove the compiler correct to get a **verified correct program compiler**. This shifts much of the proof burden for invoking the proof compiler from a per-proof basis to an upfront proof obligation (Section 5.4).

Table 1 outlines these case studies and the particular programs we consider. These case studies are available in our Coq formalization as well.

5.1 Incomplete Program Compiler

We start with an incomplete program compiler that handles only a subset of our source language grammar. This use case is inspired by a recent study of proof engineers [Ringer et al. 2020], in which one proof engineer developed a language incrementally, writing proofs about the language as they introduced new features, and revisiting those proofs after adding support for new cases. With this workflow, we can still invoke the PotPie proof compiler to get strong guarantees about

$$\text{incompleteComp}_{\text{c}}^{\varphi,k}(x := a) \triangleq \#(\varphi(x)) := \text{comp}_{\text{a}}^{\varphi}(a)$$
$$\text{incompleteComp}_{\text{c}}^{\varphi,k}(\_) \triangleq \text{skip}$$
$$\text{incompleteComp}_{\text{b}}^{\varphi}(\_) \triangleq \top$$

Fig. 12. An incomplete compiler from Imp to Stack. Here $\text{incompleteComp}_{\text{c}}$ is the code compiler, and $\text{incompleteComp}_{\text{b}}$ is the boolean expression compiler. The arithmetic expression compiler comp_{a} however is complete, and is defined as in Figure 8.

target programs as we go, so long as those programs stay in the supported incomplete fragment of the source language.

All of our program compilers build on infrastructure from Section 4.1, including the equivalence φ from Definition 4.1, which relates the environments of Imp and the stacks of Stack. With that, we can start to define our first program compiler, $\text{incompleteComp}_{\text{c}}$, which we show in Figure 12. Given Imp program $(\{f_1, \ldots, f_n\}, i_{\text{main}})$, where V contains the free variables of i_{main}, we can construct a bijection $\varphi : V \to \{1, \ldots, |V|\}$. Once we have φ, syntactic compilation of code is simple: replace a variable x with $\#(\varphi(x))$ and a parameter, param k, with $\#(|V| + k)$, producing c', which is Stack code. Finally, add $|V|$ pushes to the start of c' to make enough room for local variables.

Note that while we have the full arithmetic expression compiler, we omit nearly every grammar production for Imp statements in our definition of $\text{incompleteComp}_{\text{c}}$, because we have yet to implement them in the compiler. This means that $\text{incompleteComp}_{\text{c}}$ is, by classic correctness definitions such as functional equivalence (i.e., the behavior of the source and target programs are the same), incorrect. In spite of this, we can already use it to examine how program logics cross abstraction gaps, and to get strong guarantees about some target programs while we continue to develop the compiler. Consider, for example, compiling the `left_shift` function, that takes one parameter, `param 0`, stores the value of shifting `param 0` by 1 in `r`, and returns the value stored in `r`:[4]

```
{⊤}
r := param 0 + param 0
{r = 2 × param 0}
```

If we run the incomplete compiler on `left_shift`, we get the following Stack `left_shift` function

```
{(1, ⊤)}
push;
{(2, ⊤)}
#1 := #2 + #2
{(2, #1 = 2 × #2)}
```

as well as a proof that it matches the spec.

$$\dfrac{\dfrac{\{1, \top\}\ \texttt{push}\ \{2, \top \wedge \#1 = 0\} \quad (2, \wedge \#1 = 0) \Rightarrow (2, \top)}{\{1, \top\}\ \texttt{push}\ \{2, \top\}} \quad \dfrac{(2, \top) \Rightarrow (2, \#2 + \#2 = 2 \times \#2) \quad \{2, \#2 + \#2 = 2 \times \#2\}\ \texttt{\#1 := \#2 + \#2}\ \{2, \#1 = 2 \times \#2\}}{\{2, \top\}\ \texttt{\#1 := \#2 + \#2}\ \{2, \#1 = 2 \times \#2\}}}{\{1, \top\}\ \texttt{push; \#1 := \#2 + \#2}\ \{2, \#1 = 2 \times \#2\}}$$

[4]Note that despite multiplication being missing from our language, we can still use it in the postcondition, because we allow assertions to contain any Coq `Prop`s.

Despite that incompleteComp$_c$ is only partially implemented, we are still able to compile the `left_shift` function code. Furthermore, we can also compile proofs about programs that use the `left_shift` function, such as the following program.

```
{⊤}
x := left_shift(y)
{x = 2 × y}
```

This `left_shift` program is compiled to the following snippet of Stack code.

```
push;
push;
{2, ⊤}
#1 := left_shift(#2);
{2, #1 = 2 × #2)}
pop;
pop
```

We are able to prove that for all σ, Δ, σ_s such that $(\sigma, \Delta) \approx_\varphi \sigma_s$, we have $\sigma, \Delta \models x = 2 \times y$ if and only if $\sigma_s \models (2, \#1 = 2 \times \#2)$. We can also show that due to the behavior of `left_shift`, we have $\top \Rightarrow \text{shift}(y) = 2 \times y$, an implication that is compiled soundly to $(2, \top) \Rightarrow (2, \text{shift}(\#2) = 2 \times 2)$. This satisfies all of the requirements of the proof compilers, so we can produce the following proof.

$$\frac{(2, \top) \Rightarrow (2, \text{shift}(\#2) = 2 \times \#2) \qquad \{2, \text{shift}(\#2) = 2 \times \#2\}\ \texttt{\#1 := shift(\#2)}\ \{\#1 = 2 \times \#2\}}{\{2, \top\}\ \texttt{\#1 := shift(\#2)}\ \{2, \#1 = 2 \times \#2\}}$$

Our proof compiler however will not allow us to prove anything that compiles in a way that breaks the specification. If we compile the `max` function for example using incompleteComp$_c$, we get this target program:

```
{(2, ⊤)}
push;
skip
{(3, ⊤ ∧ ⊤)}
```

The entire `if` statement is turned into `skip`, and because the comparisons were actually Imp booleans, which are all compiled to $\top$, the incomplete specification compiler compiles the postcondition to mean something entirely different than it did in the source, preventing the proof compiler from being called. This notion of unsound specification translation is developed more formally in Section 6.1.

In other words, with our incomplete program compiler, we can still use the proof compiler to get strong guarantees as long as we stay in the supported fragment of the language. We can then continue to develop our program compiler, growing the guarantees we can preserve and the proofs we can compile as we grow our program compiler itself.

5.2 Incorrect Program Compiler

Suppose we continue to grow our program compiler to support the other cases—but we also make some mistakes. These mistakes change the behavior of functions we care about, but not always in ways that break the specifications we hope to preserve. In such a case, we can still compile proofs when we do not break those specifications.

To demonstrate this, we have implemented a complete but slightly buggy program compiler and used it to compile a function such that the behavior of that function changes. Figure 13 shows this compiler, which includes the following two silly bugs:

(1) We add 1 to function return statements.

$$\mathrm{comp}^{\varphi}_{\mathrm{badb}}(a_1 \le a_2 \wedge (\neg(a_3 \le a_4 \wedge a_5 \le a_6))) \triangleq \begin{cases} \neg(\mathrm{comp}^{\varphi}_{\mathrm{a}}(a_2) \le \mathrm{comp}^{\varphi}_{\mathrm{a}}(a_1)) & \text{if } a_1 = a_3 = a_6 \\ & \text{and } a_2 = a_4 = a_5 \\ T & \text{otherwise} \end{cases}$$

$$\mathrm{comp}^{\varphi}_{\mathrm{badb}}(b) \triangleq \mathrm{comp}^{\varphi}_{\mathrm{b}}(b)$$

$$\mathrm{comp}^{\varphi,k}_{\mathrm{badc}}(\mathrm{skip}) \triangleq \mathrm{skip}$$

$$\mathrm{comp}^{\varphi,k}_{\mathrm{badc}}(x := a) \triangleq \#(\varphi(x)) := \mathrm{comp}^{\varphi}_{\mathrm{a}}(a)$$

$$\mathrm{comp}^{\varphi,k}_{\mathrm{badc}}(c_1; c_2) \triangleq \mathrm{comp}^{\varphi}_{\mathrm{badc}}(c_1); \mathrm{comp}^{\varphi,k}_{\mathrm{badc}}(c_2)$$

$$\mathrm{comp}^{\varphi,k}_{\mathrm{badc}}(\text{if } b \text{ then } c_1 \text{ else } c_2) \triangleq \text{if } \mathrm{comp}^{\varphi}_{\mathrm{badb}}(b) \text{ then } \mathrm{comp}^{\varphi,k}_{\mathrm{badc}}(c_1) \text{ else } \mathrm{comp}^{\varphi,k}_{\mathrm{badc}}(c_2)$$

$$\mathrm{comp}^{\varphi,k}_{\mathrm{badc}}(\text{while } b \text{ do } c) \triangleq \text{while } \mathrm{comp}^{\varphi}_{\mathrm{badb}}(b) \text{ do } \mathrm{comp}^{\varphi,k}_{\mathrm{badc}}(c)$$

$$\mathrm{comp}_{\mathrm{bad}\lambda}(f, n_{\mathrm{args}}, \text{return } x_{\mathrm{ret}}, i_{\mathrm{body}}) \triangleq (f, n_{\mathrm{args}}, \mathrm{comp}^{\varphi_{i_{\mathrm{body}}}, n_{\mathrm{args}}}_{\mathrm{badc}}(i_{\mathrm{body}}), \text{return } (\varphi_{i_{\mathrm{body}}}(x_{\mathrm{ret}}) + 1)\ |V|)$$

Fig. 13. An incorrect compiler. $\mathrm{comp}_{\mathrm{badb}}$ is a buggy boolean expression compiler that contains a partially implemented optimization. $\mathrm{comp}_{\mathrm{badc}}$ is a code compiler, that is implemented using the buggy boolean expression compiler, and a correct arithmetic expression compiler ($\mathrm{comp}_{\mathrm{a}}$). $\mathrm{comp}_{\mathrm{bad}\lambda}$ is a function compiler.

(2) We partially implement a small optimization on boolean expressions in the generated code. For the optimization, in the Imp language, the only comparator is ≤. All other comparators, such as =, >, etc., are desugared into the native boolean operators. For example, the expression `a>b` is desugared into `(b≤a)∧¬(b≤a∧a≤b)`. However, this is quite unwieldy, and has many execution steps, and multiple common subexpressions. One optimization would be to instead compile it to `¬(a≤b)`. This can be evaluated much more quickly. This also happens to be a correct optimization, but there is no longer a one-to-one correspondence between the original code and the optimized version. Further, the optimization is only partially implemented—if the conditional matches the desugared AST but the expressions within them do not match up to `a` and `b`, the buggy boolean expression compiler will simply output `true`.

Despite the bugs in this compiler, this proof of the max function is still compilable because the unsound compiler optimization behavior is not encountered, i.e., the if statement's condition is compiled correctly.

```
{⊤}
push
if (¬ (#3 ≤ #2)) then
  #1 := #3
else
  #1 := #2
{#2 ≤ #1 ∧ #3 ≤ #1}
```

More saliently, we are able to compile the following triple in spite of the fact that the `max` function behavior will be meaningfully changed in the target.

```
{⊤}
x := max(y, z)
```

```
{y ≤ x ∧ z ≤ x}
```

Despite the fact that the return statement for the compiled `max` function has a 1 added to it, this does not affect the given weakened specification of max, and thus this triple can still be compiled.

```
{3, ⊤}
#1 := max(#2, #3)
{#2 ≤ #1 ∧ #3 ≤ #1}
```

However, if we tried to prove an analogous specification for a call of this `min` function:

```
{⊤}
y := param 0
if (param 0 ≤ param 1 ∧ ¬ (y ≤ param 1 ∧ param 1 ≤ param 0)) then
  z := param 0
else
  z := param 1
{z ≤ param 0 ∧ z ≤ param 1}
```

the proof would not go through, as the broken compiler actually breaks even the weaker specification. This is because this `min` function, which uses a local variable to store the value of one of the parameters, has the boolean expression `param 0 ≤ param 1 ∧ ¬ (y ≤ param 1 ∧ param 1 ≤ param 0)` in its `if` statement. When we try to compile that condition, $\text{comp}_{\text{badb}}$ tries to determine if

- `param 0 = y = param 0`, and
- `param 1 = param 1 = param 1`.

Since `param 0 ≠ y`, the if statement's condition is instead compiled to `⊤`, resulting in the following:

```
{(2, ⊤)}
push
push
#1 := #3
if (T)
then
  #2 := #3
else
  #2 := #4
{(4, #2 ≤ #3 ∧ #2 ≤ #4)}
```

Because only the `then` branch will be executed ever, it is possible that the result, stored in #2, will not be less than or equal to the second argument of `min`, which is stored in #4.

More generally, this approach to translating proofs encourages finding the most relaxed implication capable of proving a desired property in order to dodge potential compiler bugs. If we want to preserve all implications, then we need to define a correct compiler.

5.3 Unverified Correct Program Compiler

Imagine that we fix our buggy program compiler to get a correct program compiler. In this case, we can always call the proof compiler—even if we have not yet verified our program compiler.

Figure 14 defines the correct compiler. We can use this compiler in combination with our PotPie proof compiler to get strong guarantees about any of the `shift`, `max`, and `min` functions at the target level. For example, the `min` program[5] would be compiled correctly:

[5]We do not show a proof of the `min` function body, since it would be large and is similar to the `max` example in Figure 1.

$$
\begin{aligned}
\mathrm{comp}^{\varphi,k}_{\mathrm{code}}(skip) &\triangleq \mathrm{skip} \\
\mathrm{comp}^{\varphi,k}_{\mathrm{code}}(x := a) &\triangleq \#(\varphi(x)) := \mathrm{comp}^{\varphi}_{\mathrm{a}}(a) \\
\mathrm{comp}^{\varphi,k}_{\mathrm{code}}(c_1; c_2) &\triangleq \mathrm{comp}^{\varphi,k}_{\mathrm{code}}(c_1); \mathrm{comp}^{\varphi,k}_{\mathrm{code}}(c_2) \\
\mathrm{comp}^{\varphi,k}_{\mathrm{code}}(\text{if } b \text{ then } c_1 \text{ else } c_2) &\triangleq \text{if } \mathrm{comp}^{\varphi}_{\mathrm{b}}(b) \text{ then } \mathrm{comp}^{\varphi,k}_{\mathrm{code}}(c_1) \text{ else } \mathrm{comp}^{\varphi,k}_{\mathrm{code}}(c_2) \\
\mathrm{comp}^{\varphi,k}_{\mathrm{code}}(\text{while } b \text{ do } c) &\triangleq \text{while } \mathrm{comp}^{\varphi}_{\mathrm{b}}(b) \text{ do } \mathrm{comp}^{\varphi,k}_{\mathrm{code}}(c) \\
\mathrm{comp}_{\lambda}(f, k_\lambda, i, \text{return } x) &\triangleq (f, k_\lambda, \mathrm{comp}^{\varphi_\lambda,k_\lambda}_{\mathrm{code}}(i), \text{return } \#\varphi_\lambda(x)\ |V_i|)
\end{aligned}
$$

Fig. 14. A correct compiler from Imp to Stack. φ is a map from variables to stack indices, and k is the number of available function parameters. $\mathrm{comp}_{\mathrm{code}}$ is a code compiler, $\mathrm{comp}_{\mathrm{a}}$ and $\mathrm{comp}_{\mathrm{b}}$ are the arithmetic and boolean expression compilers, respectively, defined in Figure 8, and comp_{λ} is the function compiler. The correct code compiler $\mathrm{comp}_{\mathrm{code}}$ is defined recursively.

```
{(3, ⊤)}
#1 := min(#2, #3)
{(3, #1 ≤ #2 ∧ #1 ≤ #3)}
```

and the proof of this Hoare triple would also be compiled correctly:

$$
\frac{(3, \top) \Rightarrow (3, \min(\#2, \#3) \le \#2 \wedge \min(\#2, \#3) \le \#3) \quad \{3, \min(\#2, \#3) \le \#2 \wedge \min(\#2, \#3) \le \#3\}\ \texttt{\#1 := min(\#2, \#3)}\ \{\#1 \le \#2 \wedge \#1 \le \#3\}}{\{3, \top\}\ \texttt{\#1 := min(\#2, \#3)}\ \{3, \#1 \le \#2 \wedge \#1 \le \#3\}}
$$

Since this program compiler is correct, we can always satisfy the proof obligations to invoke the proof compiler, even if the compiler is not yet verified.

5.4 Verified Correct Compiler

Finally, suppose that we verify our correct compiler. While this is not necessary to invoke the proof compiler, it does buy us the additional benefit of dispatching many of the user proof obligations for invoking the proof compiler.

For our verified correct compiler, we prove the following in Coq (for clarity we have omitted assumptions about the well-formedness of compiler inputs—e.g., functions are called with the right number of arguments—please see our formalization):

Theorem 5.1. *For Imp arithmetic expressions a, boolean expressions b, and commands i, given a function environment Λ, variable environment σ, compiled function environment Λ', and variable map $\varphi : V \to \{1, \ldots, |V|\}$ we have*

- $\langle\ \sigma, \Lambda, \Delta, a\ \rangle \Downarrow_a c \wedge (\sigma, \Delta) \approx_\varphi \sigma_s \implies \langle\ \sigma_s, \Lambda', \mathrm{comp}^{\varphi}_{\mathrm{a}}(a)\ \rangle \Downarrow_a (\sigma_s, c)$
- $\langle\ \sigma, \Lambda, \Delta, b\ \rangle \Downarrow_b v \wedge (\sigma, \Delta) \approx_\varphi \sigma_s \implies \langle\ \sigma_s, \Lambda', \mathrm{comp}^{\varphi}_{\mathrm{b}}(b)\ \rangle \Downarrow_b (\sigma_s, v)$
- $\langle\ \sigma, \Lambda, \Delta, i\ \rangle \Downarrow_i \sigma' \wedge (\sigma, \Delta) \approx_\varphi \sigma_s \wedge (\sigma', \Delta) \approx_\varphi \sigma'_s \implies \langle\ \sigma_s, \Lambda', \mathrm{comp}^{\varphi,|\Delta|}_{\mathrm{code}}(i)\ \rangle \Downarrow_i \sigma'_s$.

Because the relation $\approx_\varphi$ ignores the stack beyond index $|V| + |\Delta|$ (by Definition 4.1), we can be assured that function execution is correct as well since our compiler correctness theorems are robust to the rest of the call stack.

Our proof of correctness differs slightly from more traditional compiler correctness theorems, as our semantics is big-step as opposed to small-step. Because of this, we do not use the traditional simulation diagram to model the usual observable behavior mimicry.

While this proof of program compiler correctness is strictly a bonus, and is not needed for invoking the proof compiler, verifying the program compiler does come with some additional benefits. Specifically, it makes the proof compilation process vastly more automatic—instead of having to prove that the specification compiler is sound with regards to each of the pairs of specifications in the implication database, the caller can simply prove that all of the Imp expressions in those specifications terminate. The termination condition in addition to the forwards compiler soundness allows for automatic translation of arbitrary implications. In fact, this is how a previous version of our implementation worked.

If we also had a backwards compiler correctness theorem, the caller would not even need to prove termination, as implications would be able to be translated completely automatically.

6 SOUNDNESS

The PotPie approach is parameterized not only over the particular choice of program compiler, but also the specification compiler. The proof compiler is then derived from a combination of the program and specification compilers (see Section 7.4 for some important notes on how this currently manifests in theory versus in practice). The assumptions of the proof compiler guarantee the correctness of the compiled proof, in the case that a proof *is* produced. That is, when a proof is produced, the proof will (1) be correct, and (2) have the same meaning as the source proof.

More formally, fix the program compiler and the specification compiler. To invoke the proof compiler, we require that the specification compiler be sound with regards to the precondition and postcondition of the source proof—this is a proof obligation for the caller.[6] We define this soundness condition as follows:

Definition 6.1. A specification compilation function $\text{comp}_{\text{spec}}^{\varphi,k}$ is sound with respect to P if for all σ, Δ, σ_s such that $(\sigma, \Delta) \approx_\varphi \sigma_s$, we have $\sigma, \Delta \models P \Leftrightarrow \sigma_s \models \text{comp}_{\text{spec}}^{\varphi,k}(P)$.

With this notion of specification soundness as a bidirectional implication, we can now informally state the soundness theorem for the proof compiler (we will state this more precisely in Section 6.2):

Theorem 6.2. *If the user is able to invoke the proof compiler on a source-level Hoare proof* $H = \{P\}\ c\ \{Q\}$*, our system guarantees that*

- $\text{comp}_{\text{pf}}^{\varphi,k}(H) = \{P'\}\ c'\ \{Q'\}$ *is a valid Hoare-deduction proof at the target level.*
- *The specification compilation function used in* comp_{pf} *is sound with respect to* P *and* Q*.*

The PotPie proof compiler in Coq satisfies this theorem *by construction*, meaning that it is simultaneously defined and proven correct (see Section 7 for implementation details). The remainder of this section describes at a high level the key insights behind the soundness of the proof compiler (Section 6.1), and provides a proof sketch of this soundness theorem (Section 6.2).

6.1 Key Insights

Because of the correct-by-construction nature of the proof compiler, merely being able to invoke it on a particular source program, specification, and proof is sufficient to show the target-level guarantees. Furthermore, even if the provided functions are compiled in a way that does not fully preserve their semantics, if they still fulfill the provided list of implications in the target environment, then the proof compiler is still invokable and these guarantees still hold. If the function compiler is so egregiously broken that the functions no longer fulfill the implication

[6] Note that the specification compiler need not be sound with respect to *all* proofs, but it must be sound with respect to the *particular* input source proof. If the specification compiler is sound in general, then this proof obligation is trivially satisfied.

database (recall Definition 3.1 in Section 3.3), then the proof compiler is simply not invokable. For an example of this, see the `min` example in Section 5.2.

The key technical insight that allows us to achieve these guarantees is that the Hoare rule of consequence is the only Hoare rule that depends on the semantics of the program being proved—all other rules are strictly syntactic. Without the rule of consequence, every proof on a program constructed using the traditional Hoare backward reasoning deduction rules would have an identical tree structure, and all intermediate assertions are completely determined by the postcondition. More formally, if we define a function $hoareTreeStructure$ that takes a Hoare proof and returns a tree where the nodes are labelled by the Hoare rule applied, then for any two consequence-less proofs P and P' of the same program, we have $hoareTreeStructure(P) = hoareTreeStructure(P')$, regardless of what properties P and P' prove about the program.

For example, consider the following program, where `a` and `b` are arithmetic expressions:

```
y := a;
x := b
```

For any proof of any property P, a consequence-less proof will result in a Hoare triple of the form:

```
{P[y → a][x → b]}
y := a;
x := b
{P}
```

This proof is purely mechanical—we do not need to even care what P is and it is still a valid proof. However, if we prepend a rule of consequence to the precondition, then we end up with a proof of the form:

```
{Q}
y:= a;
x := b
{P}
```

where what Q can be is dependent on the semantics of `a`, `b`, and P. In other words, in order to construct this proof, we now need to actually inspect the semantics of the program and the assertion in order to verify the soundness of the implication. Unfortunately, automatically translating consequence across compilation would require a compiler correctness guarantee, as allowing for arbitrary implications to be translated requires a full, sound, semantic translation, i.e., compiler correctness. On the other hand, this means that if a compiler preserves the control-flow structure of a program, it is relatively straightforward to reconstruct a consequence-less proof from the source language in the target language.

The solution that we use is to have the user specify which implications they are using in their Hoare proof, and then manually prove that these implications commute soundly across compilation. More formally, we require all of the implications in our source-level implication database to have the following property:

Definition 6.3. An Imp implication $P \Rightarrow Q$ has a valid translation with respect to φ and k and function environment Λ, if for all σ, Δ, σ_s, if $(\sigma, \Delta) \approx_\varphi \sigma_s$, then

- $\sigma, \Delta \models P$ if and only if $\sigma_s \models \text{comp}^{\varphi,k}_{\text{spec}}(P)$
- $\sigma, \Delta \models Q$ if and only if $\sigma_s \models \text{comp}^{\varphi,k}_{\text{spec}}(Q)$
- If $\sigma_s \models \text{comp}^{\varphi,k}_{\text{spec}}(P)$, then $\sigma_s \models \text{comp}^{\varphi,k}_{\text{spec}}(Q)$, i.e., $\sigma_s \models \text{comp}^{\varphi,k}_{\text{spec}}(P) \Rightarrow \text{comp}^{\varphi,k}_{\text{spec}}(Q)$.

Since $\sigma, \Delta \models P \Rightarrow Q$ is defined as if $\sigma, \Delta \models P$, then $\sigma, \Delta \models Q$, we know that $\sigma, \Delta \models P \Rightarrow Q$ if and only if $\sigma_s \models \text{comp}^{\varphi,k}_{\text{spec}}(P) \Rightarrow \text{comp}^{\varphi,k}_{\text{spec}}(Q)$. This preserves meaning through proof compilation,

since we are unable to destroy meaning by, for example, compiling every P and Q to $\bot$. With these implications in hand, we can automatically translate the syntactic Hoare rules to construct the subtrees, and stitch them back together into a complete proof using the manually translated implications. This approach allows for optimizing—or even incorrect—compilation, so long as it preserves the required implications.

This set of required implications can be extracted via a weakest precondition calculation. While this allows us to automate the construction of the source Hoare tree and would theoretically allow us to mechanically find the structure of the target Hoare tree, this still amounts to a roughly equivalent proof burden as the user will still have to prove that the resulting implication databases in source and target are sound.

6.2 Proof Sketch

With these insights in mind, we can now give a sketch of a proof for the precise statement of Theorem 6.2 (also proven in Coq for our concrete implementation).

Theorem 6.2. *For a set of program and specification compilers that satisfy the equations in Figure 11 and purity conditions on a program c, a source-level Hoare proof $H = \{P\}\ c\ \{Q\}$, an implication database I for that Hoare proof, and the two user proof burdens described in Section 7.3, we can construct a proof compiler that satisfies the following:*

- $\mathrm{comp}_{\mathrm{pf}}^{\varphi,k}(H) = \left\{\mathrm{comp}_{\mathrm{spec}}^{\varphi,k}(P)\right\}\ \mathrm{comp}_{\mathrm{code}}^{\varphi,k}(c)\ \left\{\mathrm{comp}_{\mathrm{spec}}^{\varphi,k}(Q)\right\}$ *is a valid Hoare-deduction proof at the target level*
- *The compiled proof proves the same Hoare triple as the original, i.e., for all σ, Δ, σ_s such that $(\sigma, \Delta) \approx_\varphi \sigma_s$, $(\sigma, \Delta) \vDash P \Leftrightarrow \sigma_s \vDash \mathrm{comp}_{\mathrm{spec}}^{\varphi,k}(P)$ and $(\sigma, \Delta) \vDash Q \Leftrightarrow \sigma_s \vDash \mathrm{comp}_{\mathrm{spec}}^{\varphi,k}(Q)$.*

Proof. Because the second condition can be dispatched exactly by the given user proof obligations in Section 7.3, we need only prove the first. To that end, we proceed by induction on the structure of the Imp program logic, utilizing the fact that the code compiler must preserve control flow for the compiled program.

The Assign case is a straightforward application of the first and third equations in Figure 11:

$$\begin{aligned}&\{\mathrm{comp}_{\mathrm{spec}}^{\varphi,k}(P[x \to a])\}\mathrm{comp}_{\mathrm{code}}^{\varphi,k}(\mathrm{x} := \mathrm{a})\{\mathrm{comp}_{\mathrm{spec}}^{\varphi,k}(P)\}\\ =\ &\{(\mathrm{comp}_{\mathrm{spec}}^{\varphi,k}(P))[\varphi(x) \to \mathrm{comp}_{\mathrm{code}}^{\varphi,k}(a)]\}\#\varphi(\mathrm{x}) := \mathrm{comp}_{\mathrm{arith}}^{\varphi,k}(\mathrm{a})\{\mathrm{comp}_{\mathrm{spec}}^{\varphi,k}(P)\}\end{aligned}$$

The If and While rules are similarly simple applications of the second, fourth, and fifth equations from Figure 11, and Seq is a straightforward application of the induction hypothesis.

For the last case, we must compile the rules of consequence. If we are compiling the left rule of consequence rule in Imp, we have the following.

$$\frac{(P_1, P_2) \in I \qquad P_1 \Rightarrow P_2 \qquad \{P_2\}\ i\ \{Q\}}{\{P_1\}\ i\ \{Q\}}\ \textsc{RoCLeft}$$

In order to translate this in into the Stack left rule of consequence, we want to show the following:

$$\frac{\begin{array}{c}\mathrm{comp}_{\mathrm{spec}}^{\varphi,k}(P_1) \Rightarrow \mathrm{comp}_{\mathrm{spec}}^{\varphi,k}(P_2)\\ \left\{\mathrm{comp}_{\mathrm{spec}}^{\varphi,k}(P_2)\right\}\ \mathrm{comp}_{\mathrm{code}}^{\varphi,k}(i)\ \left\{\mathrm{comp}_{\mathrm{spec}}^{\varphi,k}(Q)\right\}\end{array}}{\left\{\mathrm{comp}_{\mathrm{spec}}^{\varphi,k}(P_1)\right\}\ \mathrm{comp}_{\mathrm{code}}^{\varphi,k}(i)\ \left\{\mathrm{comp}_{\mathrm{spec}}^{\varphi,k}(Q)\right\}}\ \textsc{RoCLeftStk}$$

Table 2. The proof engineering effort that went into stating and formalizing PotPie, as well as creating the infrastructure to support the code and specification languages, the logics, and all of the compilers. (Here, "specs" means the number of `Definitions`, `Fixpoints`, and `Inductives`.) WF stands for "well-formed."

Category	Imp			Stack			Base	Compiler			Case	Total
	Lang	Logic	WF	Lang	Logic	Frame	Assertions	Code	Spec	Proof	Studies	
LOC	1,246	1,800	3,488	2,857	575	5,329	905	5,268	1,227	5,553	3,410	31,658
Theorems	20	55	98	86	17	204	37	136	25	129	143	950
Specs	51	36	51	49	51	49	30	82	40	66	176	681

From preservation of the implications in I, we already have $\text{comp}^{\varphi,k}_{\text{spec}}(P_1) \Rightarrow \text{comp}^{\varphi,k}_{\text{spec}}(P_2)$, and

$$\left\{\text{comp}^{\varphi,k}_{\text{spec}}(P_2)\right\}\ \text{comp}^{\varphi,k}_{\text{code}}(i)\ \left\{\text{comp}^{\varphi,k}_{\text{spec}}(Q)\right\}$$

simply follows from the induction hypothesis. The right rule of consequence follows similarly. □

7 IMPLEMENTATION

We implement PotPie in Coq and prove that it is sound (Section 7.1). We keep the *trusted computing base* (TCB) small (Section 7.2), but impose some proof burden on the user on a per-program-and-proof basis (Section 7.3). As a historical artifact of the proof engineering process, there are still minor differences between the PotPie approach and implementation (Section 7.4). The primary outstanding implementation challenge we face relates to obtaining source-independent proof objects at the target level after invoking the PotPie proof compiler (Section 7.5)—something that is not needed for basic use, but that would open up new and exciting use cases for PotPie.

7.1 Formal Proof

We formalize PotPie in Coq. Our formalization includes formal proofs of the soundness guarantees from Section 6, along with the case studies from Section 5. Table 2 summarizes the number of lines of code, proofs, and specifications for different components of PotPie.

In the Coq formalization, the soundness theorem from Section 6 holds *by construction*. That is, the PotPie proof compiler is a function in Coq whose type specifies its correctness:

```
hl_compile : ∀ (P Q : log) (i : imp) (fenv : fun_env) (HL : hl P i Q fenv)
(facts : fact_environment) (facts' : stk_fact_environment),
  ∀ (map : list ident) (args : nat) (P' Q': AbsState) (i' : stk) (fenv' : fun_env_stk),
    (* ... well-formedness conditions ... *) →
    logic_transrelation args map P P' →
    logic_transrelation args map Q Q' →
    i' = compile_code i →
    imp_trans_valid_def facts facts' fenv fenv' map args →
    hl_stk P' i' Q' fenv'.
```

where the type above is simplified for clarity, and where the omitted conditions are described in Section 7.3. In other words, the `hl_compile` function takes an Imp Hoare logic proof `HL` that shows $\{P\}\ i\ \{Q\}$ in function environment `fenv`, and compiles it to a Stack Hoare logic proof that shows $\{P'\}\ i'\ \{Q'\}$ in function environment `fenv'`, where each of P', i', Q', and `fenv'` are correctly related to their Imp counterparts.

We chose this correct-by-construction design for its elegance and simplicity. In particular, this design lets us take advantage of Coq's trusted type-checking kernel in two different ways:

(1) **Prohibiting Invalid Inputs**: The `hl_compile` function cannot even be invoked on invalid inputs, like when the program compiler is incorrect in a way that will break the correctness guarantee (preservation of the specification across abstraction levels) over the particular input source-level program. In such a case, the hypotheses of `hl_compile` will not be provable, and so it will be impossible to invoke `hl_compile` on that particular input.
(2) **Producing Free Proofs**: Invoking `hl_compile` on a source-level proof with the requisite hypotheses proven produces not just a target-level proof, but also a proof that the proof is correct (that is, that the target-level proof satisfies a specification that is analogous to the source-level specification). Both the proof and the correctness guarantee about it are checkable even without reducing the application of `hl_compile`, since Coq's trusted kernel checks that the application produces a term with the correct type.

The details of our proof development can be found in the associated artifact.

7.2 Trusted Computing Base

PotPie's TCB consists of just the Coq kernel, the mechanized semantics, and two localized axioms for reasoning about equalities between dependent types. The correctness proof for the correct program compiler provides some confidence that the semantics are implemented correctly, at least with respect to one another.

Both axioms that we assume are localized instances of Uniqueness of Identity Proofs (UIP) over particular types, without broadly implying UIP. UIP, which is consistent with Coq, states that all equality proofs are equal to each other for *all* types—we instead assume that all equality proofs are equal to each other for *two particular types*. Like more general UIP, this localization of UIP is convenient for proof engineering.[7] We place this restriction on UIP rather than assuming the more general version because assuming UIP in general would be computationally undesirable, as UIP does not have a constructive interpretation in Coq. We assume UIP for `AbsEnvs` (abstract environments, the Coq implementation of SM in Equation 1 in Section 3.2) and function environments:

Axiom 1 (UIP for `AbsEnvs`). *If* $a, a' : \texttt{AbsEnv}$*, then if* $p_1, p_2 : a = a'$*, we have* $p_1 = p_2$*.*

Axiom 2 (UIP for Function Environments). *If* Λ, Λ' *are Imp function environments, and we have proofs* $p_1, p_2 : \Lambda = \Lambda'$*, then* $p_1 = p_2$*.*

Axiom 1 is reasonable to assume since we do not compute on the contents of `AbsEnvs`. Standing in the way of provable UIP is the presence of functions (representing predicates in our base assertions from Section 3.1) in the constructors of `LogicProp`, one of the types that makes up `AbsEnv`. But we never compute on the contents of these functions. Furthermore, even though Axiom 1 implies UIP for `LogicProp`, by requiring that `LogicProp`'s parameters V and A have decidable equality, UIP provably holds for V and A, so Axiom 1 does not imply the general version of UIP—though it *does* imply UIP for pairs and lists of `AbsEnvs`, a fact that we prove and use in our proof development. Likewise, Axiom 2 is a reasonable assumption because the function environment in both languages is considered to be global—there can only ever be one function environment. Furthermore, as function environments are functions whose domain and codomain both have decidable equality and thus UIP, Axiom 2 also does not imply the general version of UIP.

[7] UIP is a common axiom to assume in proof engineering efforts because it makes it simple to deal with the interactions between equalities and dependent types. Its general form is incompatible with univalence, but Coq is not univalent.

7.3 User Proof Burden

To minimize our TCB, PotPie requires additional proof obligations from the caller, beyond just the source proof that is being compiled. These proof obligations can be split into two major categories: well-formedness conditions and specification translation conditions.

Well-Formedness Conditions. While the syntax of Imp prevents type errors, it allows for functions to be called an any number of arguments, and for any parameter index to be called within the program body. Our proof compiler requires that all components of the source proof be well-formed, in that neither of these cases occur. Because the assertion languages are inductively built off of the program syntax, we not only need to recurse through the program to check for these conditions, but we also need to recurse through the proof body to check the assertions, as well as through the bodies of the functions called from both source and target function environments. We implement these as inductive relations that must be satisfied upon invoking the proof compiler for a particular program-and-proof pair.

In the future, we hope to be able to automate many of these well-formedness obligations, as they should be relatively simple static analyses. The main challenge in doing so is with wrangling the data inside of the dependent types we use to represent Hoare proof trees into a form that is nice to structurally recurse over. MetaCoq [Sozeau et al. 2020] may be a useful tool to that end as it would allow us to build automation that inducts over the syntax of our Coq inductive propositions and relations, which would be useful in manipulating this data. Using MetaCoq for this purpose may also help us eject Axiom 1, which is more powerful than we need—syntactic equality of propositions should be perfectly workable for our purposes.

Specification Translation Conditions. Because of the challenges with translating the rule of consequence and potentially incorrect compiler behavior, we require the user to prove two conditions about specification translation: (1) the implications invoked by the rule of consequence must be preserved, and (2) the specification compiler must be sound with regards to the precondition and postcondition of the proof being compiled. These conditions require no proofs of *language-wide* properties, nor do they require *full compiler correctness*. Rather, they require specific correctness properties of *concrete programs*, which often require only finite reasoning about finite objects.

To simplify the user proof burden right now, we provide some simple tactics to dispatch common goals. We are yet to build more sophisticated automation for this, though we have plans: In implementing the case studies from Section 5, we found that the most tedious part of proving these obligations when calling the PotPie proof compiler was constructing large finite inhabitants of inductive relations (through, for example, `repeat constructor`, though sometimes we had to provide more information). Because of this, we are particularly hopeful about building reflective tactics for this category of proof obligations, since these tactics excel at dispatching goals that can be proven by large finite applications of constructors (see Chlipala [2013b], Chapter 15, "Proof by Reflection").

Lastly, we note that while generating the implication database for a given proof as a verification condition is technically automateable, we have not yet implemented this automation.

7.4 Differences in Theory and Practice

The approach we display in Figure 10 can technically be parameterized over any set of code, specification, and proof compilers that satisfy the equations in Figure 11 for the fixed proof being compiled (both of these figures are in Section 4.4). In our current implementation, the way we check that these equations are satisfied is to check the compiled specifications against the relational version of a correct compiler. Additionally, we restrict all incorrect/nontrivial compiler behavior to the function environment compilation. While this was the most lightweight way of checking these

constraints in the timeline of our proof development as we already had all of these components implemented, there is no theoretical reason the correct compilers need to already exist in order to construct a proof compiler. Instead of checking against a reference correct compiler, a program to simply check the equations in Figure 11 for a concrete instantiation of a proof would be perfectly adequate for our purposes. We plan to do this in future iterations of this work.

Another oddity of this specific implementation is that the target-level program logic has a full-power rule of consequence, and this rule of consequence is actually invoked in the proof compilation. This is due to an idiosyncrasy of the way that we construct conjunctions in the target If and While Hoare rules. The specification constructed out of the boolean condition for each of these rules sets the minimum stack size to be 0. We use the rule of consequence to transform that 0 into $|V| + k$. In our implementation, this is the only non-one-to-one translation of Hoare rules.

One additional caveat is that currently, it is impossible to prove implications of the form $\top \Rightarrow P$, where P is some assertion referencing parameters. Even if we just want to prove $\top \Rightarrow \text{param } k = \text{param } k$, this necessitates proving that for all σ, Δ such that $\sigma, \Delta \vDash \top$, we have $\sigma, \Delta \vDash \text{param } k = \text{param } k$. However, knowing that $\sigma, \Delta \vDash \top$ does not mean that Δ is long enough to contain a k^{th} parameter, and in fact, we know nothing about σ and Δ from $\sigma, \Delta \vDash \top$. This is mainly a problem for proving Hoare triples about function bodies, where the Hoare triple is of the form $\{\top\}\ i_{\text{body}}\ \{P_{\text{post}}\}$, since function bodies are likely to contain parameters, and the rule of consequence is often necessary for proving anything nontrivial. This was not a problem for the `max` example (see Figure 1), since all of the implications include uses of parameters in the antecedent. There are several ways we may be able to fix this, for ease of use. One way would be to add a minimum args size expected to the Imp assertions, similar to the minimum stack size in Stack assertions. A workaround for now would be to instead change the Hoare triple for the function body from $\{\top\}\ i_{\text{body}}\ \{P_{\text{post}}\}$ to $\{\text{param } 0 = \text{param } 0 \wedge \cdots \wedge \text{param } k-1 = \text{param } k-1\}\ i_{\text{body}}\ \{P_{\text{post}}\}$, where k is the number of parameters expected by the function.

7.5 Challenge: Source-Independent Proof Objects

One potential benefit of compiling proof objects from a source logic to a target logic is that the compiled proofs, once reduced, could truly be free of references to the source language and logic. This is certainly true of our on-paper proof compiler, but it is not yet true of the PotPie implementation. That this is not yet true is not an issue of correctness since, thanks to Coq's trusted kernel, the PotPie approach to proof compilation works even without reducing an application of the proof compiler down to a source-independent proof object. But we would like this to be true of our implementation as well.

There are many reasons to want source-independent proof objects at the target level in the PotPie implementation: These proof objects could be robust to changes in the source language or logic. Once generated, they could exist forever, even without access to the source. They could be independently checkable even by someone who does not trust the initial verification effort, or by someone who lacks access to the source entirely. They could be directly repaired in response to changes in target-level programs, in the style of recent work on proof repair in an embedded separation logic [Gopinathan et al. 2023]. They could potentially even be linked directly with other proofs compiled from other source languages and logics.

We believe that the barriers to source-independent proof objects in the PotPie implementation are nontrivial, but surmountable. For one, the correct-by-construction design means that proofs carry correctness proof information, so currently, source code well-formedness principles may show up in compiled proofs. In addition, Coq's proofs are *opaque* by default—Coq does not unfold

constants corresponding to theorem names unless it is specifically told the contents matter—which may prohibit reduction that would otherwise remove source program and proof references.

We see three paths toward full source-independence:

(1) relaxing proof obligations for the user,
(2) decoupling the correctness proof from the computation, and
(3) setting opaque proofs to transparent.

Each path comes with its own challenges—the first to engineering, the second to elegance and simple specifications, and the third to efficiency. We are actively exploring these paths.

Once we have source-independent proof objects, we hope to implement linking of target-level proof objects. That way, proof engineers will be able to write proofs about components of systems implemented in different languages, and easily get machine-checkable proofs at the target level about the entire system. The main barriers to direct linking of target-level proof objects beyond removing the remaining source-dependent fragments of compiled proofs are generalizing our Stack program logic semantics, implementing proof compilers for multiple languages, and reasoning about the interactions between compiled components.

8 RELATED WORK

Proof-Transforming Compilation. Early work compiling proofs positions itself as an extension of *proof-carrying code* [Necula 1997], in which the compiler produces proofs of important properties about compiled code. Barthe et al. [2006] stated a theorem relating source and target program logics reasoning about languages similar to those we use in our work. There was not yet transformation of proof objects themselves. Transformation of proof objects followed shortly after, going by the names *proof-transforming compilation* and *certificate translation.*

Early proof-transforming compilation work [Müller and Nordio 2007] transformed proofs about Java-like programs to proofs about bytecode. Proofs were written in Hoare-style program logics defined over the respective source and target languages, and implemented by way of a prototype targeting proofs written in XML. Later work [Nordio et al. 2008] implemented proof-transforming compilation from Eiffel to bytecode, formalizing the specification compiler in Isabelle/HOL, and writing a hand-written proof of correctness of the proof compiler. A subsequent thesis [Hauser 2009] showed how to embed the compiled proofs about bytecode into Isabelle/HOL. Our work is the first we know of to formally verify the proof of correctness of the proof compiler, and to use it to support correctness proofs at the target level even when the program compiler is incorrect.

Certificate translation [Kunz 2009] compiles proofs in a program logic as well, but focuses on compiler optimization passes, like dead code elimination or constant propagation. While it does not come with an accompanying formal proof development, revisiting work on certificate translation may prove fruitful should we move to support more classic compiler optimization passes as well.

Proof Term Transformations. There is some work on proof term transformations that happen directly over proof assistant proofs, not inside of any program logic. Proof transformations for higher-order logic were introduced in 1987 to bridge automation and usability [Pfenning 1987]. They have since been used for many different purposes, like automatic generalization [Hasker and Reddy 1992; Johnsen and Lüth 2004; Kolbe and Walther 1998], reuse [Magaud 2003], and repair [Ringer 2021] of proofs. Our work implements a proof term transformation directly within the Coq proof assistant for an embedded program logic, which empowers us to fully prove the transformation correct within the proof assistant itself.

Correct Compilation. Our work is part of the broader space of correct compilation. The two main approaches to correct compilation in a proof assistant are *certified* and *certifying* compilation.

Certified compilation refers to formally proving compilers correct. One example is the CompCert verified C compiler [Leroy 2006, 2009], which was proven correct in Coq, and has been shown to lack bugs present in other compilers [Yang et al. 2011]. The CakeML [Kumar et al. 2014] verified implementation of ML in HOL4 includes a verified compiler in HOL4. Oeuf [Mullen et al. 2018] and CertiCoq [Anand et al. 2017] are both certified compilers for programs written in Gallina, the language underpinning Coq. *Certifying compilation*, in contrast, produces proofs that a given compilation is correct alongside the compiled code. The COGENT language, for example, has a certifying compiler that, for a given compiled program from COGENT to C, proves that the compiled C code correctly implements a generated high-level semantics embedded in Isabelle/HOL [Amani et al. 2016; Rizkallah et al. 2016]. A comprehensive survey of work in certified and certifying compilation is in QED at Large [Ringer et al. 2019].

Our work is both certifying (the proof compiler produces a target proof object, whose type shows that the target program proves the target specification) and certified (the proof compiler is proven correct, so that the source and target programs, specifications, and proofs must be analogous). It provides a clear specification of correctness that must always be satisfied, but also allows for proofs to be checked at the target level (though there is still work to do before said target proofs are fully source-independent in our implementation). *Rupicola* [Pit-Claudel et al. 2022], which phrases correct compilation from proof assistants to low-level code as proof search, also has both certified and certifying components. Our addition to this rich space is a technique that correctly compiles proofs from source to target embedded program logic, soundly producing target-level proofs even in some cases when the program compiler itself is sometimes incorrect.

Our work is in the spirit of *translation validation*, a method of certifying that a compiler preserves the behavior of the source program for a particular compilation run. Through the use of simulation relations, one can check whether two programs are semantically equivalent [Necula 2000]. Much like translation validation, our work does not rely on full compiler correctness, yet still allows for strong guarantees. However, we also take this a step further and allow for the compiled program to contain mistakes that sabotage full semantic equivalence with the source program, so long as those mistakes do not sabotage the particular property proven.

Program Logics. Our work correctly compiles programs and proofs from a source embedded language and program logic to a target embedded language and program logic. Our source language and logic were developed using Leroy [2021] as a base. Frameworks based on embedded program logics have seen a lot of success helping proof engineers write proofs in a proof assistant about code with features that the proof assistant lacks. Example frameworks for reasoning with embedded program logics are Iris [Jung et al. 2015; Krebbers et al. 2017], VST-Floyd [Cao et al. 2018], Bedrock [Chlipala 2011, 2013a], YNot [Nanevski et al. 2008], CHL [Chen et al. 2015], Sepref [Lammich 2019], and CFML [Charguéraud 2011]. Recent work introduced proof repair for proofs embedded in CFML, directly using embedded proof objects for the repair process [Gopinathan et al. 2023].

The VST project that VST-Floyd is a part of is especially related to our work, as it aims to create machine-checked proofs about systems that hold all the way down to machine code. VST provides three main tools to produce verified software written in C: a program logic (Verifiable C), a Coq tactic and lemma library for proof automation (VST-Floyd), and a semantic model of programs (which interacts heavily with CompCert) [Beringer and Appel 2021]. By composition with CompCert, source C programs verified in the VST program logic are guaranteed to preserve their specifications once compiled, all the way down to assembly code. Our primary contribution to this space is allowing the program compiler used in such a toolchain to be unverified or even incorrect, and yet still proving properties at the target level that are guaranteed to relate correctly to those proven at the source level.

For now, relative to the program logics in practical frameworks like Iris and VST, our embedded logics are fairly simple. And unlike VST chained with CompCert, our source and target languages are also fairly similar. We hope to extend our work to handle more practical logics and lower-level target languages in the future, so that users of toolchains like Iris and VST can use unverified or sometimes-incorrect compilers and still get strong guarantees about compiled programs.

9 CONCLUSIONS

We showed how compiling proofs across program logics can empower proof engineers to reason directly about source programs, and still obtain proofs about compiled programs—even when the program compiler is unverified or sometimes incorrect. Our implementation PotPie in Coq is formally verified, guaranteeing that compiled proofs not only prove their respective specifications, but also are correctly related to their source counterparts. Our hope is to provide an alternative to relying on verified program compilers, without sacrificing satisfaction of important specifications. Our next steps are ensuring target proof objects are source-independent, supporting different source languages and logics, supporting linking of target-level proofs, implementing optimizing compilers, and bringing the benefits of proof compilation to more practical frameworks.

ACKNOWLEDGEMENTS

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.

REFERENCES

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. 2016. Cogent: Verifying High-Assurance File System Implementations. In *International Conference on Architectural Support for Programming Languages and Operating Systems*. Atlanta, GA, USA, 175–188. https://doi.org/10.1145/2872362.2872404

Abhishek Anand, Andrew W. Appel, Greg Morrisett, Matthew Weaver, Matthieu Sozeau, Olivier Savary Belanger, Randy Pollack, and Zoe Paraskevopoulou. 2017. CertiCoq: A verified compiler for Coq. In *CoqPL*. http://www.cs.princeton.edu/~appel/papers/certicoq-coqpl.pdf

Gilles Barthe, Tamara Rezk, and Ando Saabas. 2006. 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*. Springer, 112–126.

Lennart Beringer and Andrew W. Appel. 2021. Abstraction and Subsumption in Modular Verification of C Programs. *Formal Methods in System Design* 58, 1 (Oct. 2021), 322–345. https://doi.org/10.1007/s10703-020-00353-1

Simon Byrne. 2023. Beware of fast-math. https://simonbyrne.github.io/notes/fastmath

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

Arthur Charguéraud. 2011. Characteristic Formulae for the Verification of Imperative Programs. *SIGPLAN Not.* 46, 9 (sep 2011), 418–430. https://doi.org/10.1145/2034574.2034828

Haogang Chen, Daniel Ziegler, Tej Chajed, Adam Chlipala, M. Frans Kaashoek, and Nickolai Zeldovich. 2015. Using Crash Hoare Logic for Certifying the FSCQ File System. In *Proceedings of the 25th Symposium on Operating Systems Principles* (Monterey, California) *(SOSP '15)*. Association for Computing Machinery, New York, NY, USA, 18–37. https://doi.org/10.1145/2815400.2815402

Adam Chlipala. 2011. Mostly-Automated Verification of Low-Level Programs in Computational Separation Logic. *SIGPLAN Not.* 46, 6 (jun 2011), 234–245. https://doi.org/10.1145/1993316.1993526

Adam Chlipala. 2013a. The Bedrock Structured Programming System: Combining Generative Metaprogramming and Hoare Logic in an Extensible Program Verifier. *SIGPLAN Not.* 48, 9 (sep 2013), 391–402. https://doi.org/10.1145/2544174.2500592

Adam Chlipala. 2013b. *Certified Programming with Dependent Types - A Pragmatic Introduction to the Coq Proof Assistant*. MIT Press. http://mitpress.mit.edu/books/certified-programming-dependent-types

Kiran Gopinathan, Mayank Keoliya, and Ilya Sergey. 2023. Mostly Automated Proof Repair for Verified Libraries. *Proc. ACM Program. Lang.* 7, PLDI, Article 107 (jun 2023), 25 pages. https://doi.org/10.1145/3591221

Robert W Hasker and Uday S Reddy. 1992. Generalization at higher types. In *Proceedings of the Workshop on the λProlog Programming Language*. 257–271.

Bruno Hauser. 2009. *Embedding proof-carrying components into Isabelle.* Master's thesis. ETH, Swiss Federal Institute of Technology Zurich, Institute of Theoretical ….

C. A. R. Hoare. 1969. An Axiomatic Basis for Computer Programming. *Commun. ACM* 12, 10 (oct 1969), 576–580. https://doi.org/10.1145/363235.363259

Einar Broch Johnsen and Christoph Lüth. 2004. 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*. Springer, Berlin, Heidelberg, 152–167. https://doi.org/10.1007/978-3-540-30142-4_12

Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. 2015. Iris: Monoids and Invariants as an Orthogonal Basis for Concurrent Reasoning. *SIGPLAN Not.* 50, 1 (jan 2015), 637–650. https://doi.org/10.1145/2775051.2676980

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

Robbert Krebbers, Amin Timany, and Lars Birkedal. 2017. Interactive Proofs in Higher-Order Concurrent Separation Logic. *SIGPLAN Not.* 52, 1 (jan 2017), 205–217. https://doi.org/10.1145/3093333.3009855

Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. 2014. CakeML: A Verified Implementation of ML. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Diego, California, USA) *(POPL '14)*. ACM, New York, NY, USA, 179–191. https://doi.org/10.1145/2535838.2535841

César Kunz. 2009. *Certificate Translation Alongside Program Transformations.* Ph. D. Dissertation. ParisTech, Paris, France.

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

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

Xavier Leroy. 2009. Formal Verification of a Realistic Compiler. *Commun. ACM* 52, 7 (July 2009), 107–115. https://doi.org/10.1145/1538788.1538814

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

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

Eric Mullen, Stuart Pernsteiner, James R. Wilcox, Zachary Tatlock, and Dan Grossman. 2018. Oeuf: Minimizing the Coq Extraction TCB. In *CPP* (Los Angeles, CA, USA). ACM, New York, NY, USA, 172–185. https://doi.org/10.1145/3167089

Peter Müller and Martin Nordio. 2007. Proof-Transforming Compilation of Programs with Abrupt Termination. In *SAVCBS*. 39–46. https://doi.org/10.1145/1292316.1292321

Aleksandar Nanevski, Greg Morrisett, Avraham Shinnar, Paul Govereau, and Lars Birkedal. 2008. Ynot: Dependent Types for Imperative Programs. *SIGPLAN Not.* 43, 9 (sep 2008), 229–240. https://doi.org/10.1145/1411203.1411237

George C. Necula. 1997. Proof-Carrying Code. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '97)*. Association for Computing Machinery, New York, NY, USA, 106–119. https://doi.org/10.1145/263699.263712

George C. Necula. 2000. Translation Validation for an Optimizing Compiler. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation* (Vancouver, British Columbia, Canada) *(PLDI '00)*. Association for Computing Machinery, New York, NY, USA, 83–94. https://doi.org/10.1145/349299.349314

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

Frank Pfenning. 1987. *Proof Transformations in Higher-Order Logic.* Ph. D. Dissertation. Carnegie Mellon University.

Clément Pit-Claudel, Jade Philipoom, Dustin Jamner, Andres Erbsen, and Adam Chlipala. 2022. 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* (San Diego, CA, USA) *(PLDI 2022)*. Association for Computing Machinery, New York, NY, USA, 918–933. https://doi.org/10.1145/3519939.3523706

Talia Ringer. 2021. *Proof Repair.* Ph. D. Dissertation. University of Washington.

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

Talia Ringer, Alex Sanchez-Stern, Dan Grossman, and Sorin Lerner. 2020. REPLica: REPL Instrumentation for Coq Analysis. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs* (New Orleans, LA, USA) *(CPP 2020)*. Association for Computing Machinery, New York, NY, USA, 99–113. https://doi.org/10.1145/3372885.3373823

Christine Rizkallah, Japheth Lim, Yutaka Nagashima, Thomas Sewell, Zilin Chen, Liam O'Connor, Toby Murray, Gabriele Keller, and Gerwin Klein. 2016. A Framework for the Automatic Formal Verification of Refinement from Cogent to C. In *Interactive Theorem Proving*. Springer, Cham, 323–340. https://doi.org/10.1007/978-3-319-43144-4_20

Matthieu Sozeau, Abhishek Anand, Simon Boulier, Cyril Cohen, Yannick Forster, Fabian Kunze, Gregory Malecha, Nicolas Tabareau, and Théo Winterhalter. 2020. The metacoq project. *Journal of automated reasoning* 64, 5 (2020), 947–999.

Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and Understanding Bugs in C Compilers. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation* (San Jose, California, USA) *(PLDI '11)*. Association for Computing Machinery, New York, NY, USA, 283–294. https://doi.org/10.1145/1993498.1993532