Correct Compilation of Proofs About Embedded Programs

Audrey Seo*
University of Washington, USA

Chris Lam*
University of Illinois Urbana-Champaign, USA

Dan Grossman
University of Washington, USA

Talia Ringer
University of Illinois Urbana-Champaign, USA

Abstract

Embedded programming languages and program logics make it possible to prove properties about programs written in a variety of languages, all within a proof assistant. These embeddings consist of implementations of the language and logic inside of the proof assistant language, sometimes combined with frameworks for effective reasoning. By chaining these embeddings with verified program compilers, proof engineers can reason about the source program, and get strong guarantees about the target program by composition. But something is lost in that composite workflow: there is no machine-checkable, source-independent proof object about the target program.

In this paper, we explore a different workflow that produces proof objects at the target level. The 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 the Coq proof assistant, moving from an imperative language with variables and functions, to a language where the only memory is a single call stack. Our proof compiler is formally verified—it is constructive and correct by construction, with its type specifying the relation between the source and target proofs. We believe this is the first proof compiler across program logics that is formally proven correct.

2012 ACM Subject Classification Software and its engineering → Compilers; Software and its engineering → Formal software verification

Keywords and phrases proof transformations, certified compilation, program logics, proof engineering

Digital Object Identifier 10.4230/LIPIcs.CVIT.2016.23

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.

1 Introduction

Proof engineers sometimes need to escape the confines of the proof assistant language. They may wish to reason instead about programs written in languages with features that the proof assistant lacks, like effects or concurrency. In this position, they may turn to a framework that embeds program logics for these languages directly within the proof assistant [41], like Iris [21, 23], VST [9], CHL [10], or SEPREF [26]. By chaining these program logics with a verified compiler, they can get strong guarantees about compiled programs [9].

* Co-first authors

$$
\begin{tikzcd}
\{P\} \arrow[d, "\mathrm{comp}_{\mathrm{spec}}"] & c \arrow[d, "\mathrm{comp}_{\mathrm{code}}"] & \{Q\} \arrow[d, "\mathrm{comp}_{\mathrm{spec}}"] \\
\{P'\} & c' & \{Q'\}
\end{tikzcd}
\qquad
\begin{array}{c}
\dfrac{H_1 \cdots H_n}{\{P\}\, c\, \{Q\}} \\
\big\downarrow \mathrm{comp}_{\mathrm{pf}} \\
\dfrac{\mathrm{comp}_{\mathrm{pf}}(H_1) \cdots \mathrm{comp}_{\mathrm{pf}}(H_n)}{\{P'\}\, c'\, \{Q'\}}
\end{array}
\qquad
\begin{array}{c}
\sigma_1 \models P \\
\Updownarrow \\
\sigma'_1 \models P'
\end{array}
\qquad
\begin{tikzcd}
\sigma_1 \arrow[r, "c"] \arrow[d, leftrightarrow, "\approx_\varphi"'] & \sigma_2 \arrow[d, leftrightarrow, "\approx_\varphi"] \\
\sigma'_1 \arrow[r, "c'"'] & \sigma'_2
\end{tikzcd}
\qquad
\begin{array}{c}
\sigma_2 \models Q \\
\Updownarrow \\
\sigma'_2 \models Q'
\end{array}
$$

Figure 1 The idea behind proof compiler correctness. As our proof compiler's actual correctness proof is via dependent types, its representation is merely the transformation of proof trees. Here, we feature code and spec compiler correctness. Section 4 describes the assumptions each arrow needs to satisfy in this diagram.

In this paper, we explore a different approach to getting strong guarantees about compiled programs. Our approach is to compile embedded programs, specifications, and proofs all at once—in the style of certificate translation [25] and proof-transforming compilation [33, 37, 17]. The key benefit to this approach is that it transforms proofs about source programs directly to machine-checkable proofs about their compiled counterparts. Furthermore, it does so in a way that preserves the original specification, up to changes in the abstraction level.

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. The proofs about compiled programs that PotPie produces are machine-checkable, and reduce away references to the source code whenever possible.

PotPie is constructive and correct by construction. In particular, it is defined as a dependently typed function that compiles embedded source programs, specifications, and proofs in a way that is guaranteed by virtue of its type signature to respect the correspondence between the source and target language and logic. Accomplishing this was a significant proof engineering undertaking, which we discuss in detail. Our main contributions are twofold:

1. We introduce PotPie, a formally verified, correct-by-construction proof compiler in Coq that compiles embedded programs, specifications, and proofs (Sections 2, 3, and 4). PotPie is, to the best of our knowledge, the first formally verified proof compiler across program logics (Section 6 discusses other projects, including those that did proof compilation but with only an informal proof of proof compiler correctness).
2. We describe the proof engineering considerations key to building a proof compiler like PotPie (Section 5). This includes a discussion of how to address open challenges related to reducing away remaining references to the source language, compiling proofs efficiently, and removing the sole remaining dependency on total program compiler correctness.

Throughout the paper, we annotate claims supported by formalizations with a circled number, like 42. These icons are links to the PotPie code, and are detailed further in `GUIDE.md`. Our hope moving forward is to reopen inquiry into work on producing source-independent proofs about compiled programs, with an eye on eventually linking proof objects about components of complex systems in which components may not share a common source language, and on eventually removing the need to prove the program compiler itself correct (Section 7).

2 Overview

PotPie compiles source programs, specifications, and proofs to target programs, specifications, and proofs. It does so soundly, meaning that the compiled proof proves the compiled

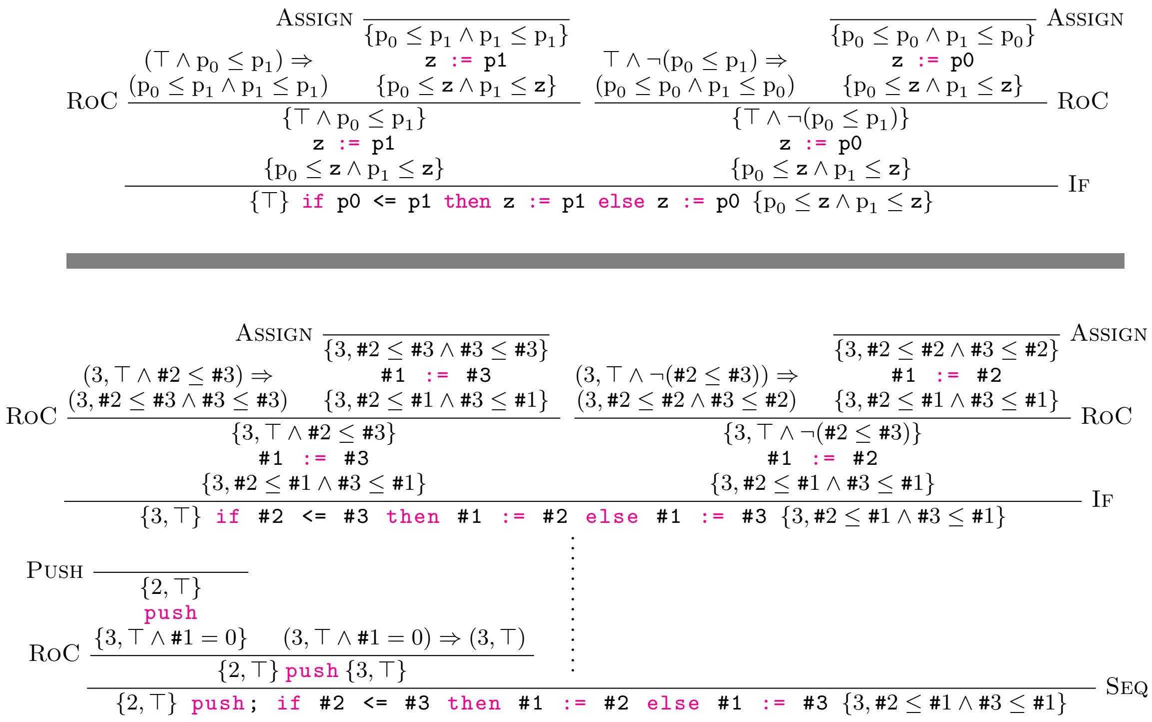

Figure 2 The Hoare tree for the program `max` (above) and its compiled version (below). For brevity, we use p_i and `pi` as shorthand for the i^{th} parameter in the specification and code, respectively.

specification about the compiled program, and that the compiled specification is the same as the source specification up to the change in abstraction from the source language to the target language. This is illustrated in Figure 1.

Our source and target languages are Imp and Stack. While both are imperative, Imp uses variables that are local to a function 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 as well as manage the stack size (via push and pop commands) to compile function calls. Due to this change in abstraction, their specification languages (which encode the Hoare triple pre and post conditions) are also different. For more details about the languages and logics, see Section 3.

To demonstrate end-to-end proof compilation, we consider a program that computes the max of two numbers, along with a specification and proof that the specification is met:

```
{⊤}
if (param 0 ≤ param 1) then
  z := param 1
else
  z := param 0
{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, $\top$, indicates that there are no requirements on its input. The program's postcondition, `z` $\geq$ `param 0` $\wedge$ `z` $\geq$ `param 1`, indicates `z` is greater than both of the inputs.

We represent the proof of this specification explicitly as a proof object that corresponds to a Hoare tree (Figure 2, top). Our goal is to compile this program, specification, and

$$
\begin{array}{rl}
a ::= & \mathbb{N} \,|\, x \,|\, \text{param } k \,|\, a_1 + a_2 \,|\, a_1 - a_2 \\
| & f(a_1, a_2, a_3, \ldots, a_n) \\
b ::= & T \,|\, F \,|\, \neg b \,|\, a_1 \le a_2 \,|\, b_1 \wedge b_2 \,|\, b_1 \vee b_2 \\
i ::= & \text{skip} \,|\, x := a \,|\, i_1; i_2 \\
| & \text{if } b \text{ then } i_1 \text{ else } i_2 \,|\, \text{while } b \text{ do } i \\
\lambda ::= & (f, n_{\texttt{args}}, x_{\texttt{ret}}, i_{\texttt{body}}) \\
p ::= & (\{\lambda_1, \ldots, \lambda_n\}, i_{\texttt{main}})
\end{array}
\qquad
\begin{array}{rl}
a ::= & \mathbb{N} \,|\, \#k \,|\, a_1 + a_2 \,|\, a_1 - a_2 \\
| & f(a_1, a_2, a_3, \ldots, a_n) \\
b ::= & T \,|\, F \,|\, \neg b \,|\, a_1 \le a_2 \,|\, b_1 \wedge b_2 \,|\, b_1 \vee b_2 \\
i ::= & \text{skip} \,|\, \text{push} \,|\, \text{pop} \,|\, \#k := a \,|\, i_1; i_2 \\
| & \text{if } b \text{ then } i_1 \text{ else } i_2 \,|\, \text{while } b \text{ do } i \\
\lambda ::= & (f, n_{\texttt{args}}, i_{\texttt{body}}, \text{return } a_{\texttt{ret}}\ n_{\texttt{pop}}) \\
p ::= & (\{\lambda_1, \ldots, \lambda_n\}, i_{\texttt{main}})
\end{array}
$$

Figure 3 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 STACK functions, the construct return $a_{\texttt{ret}}$ $n_{\texttt{pop}}$ means that the function returns the result of evaluating $a_{\texttt{ret}}$ and then pops off $n_{\texttt{pop}}$ variables from the stack.

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

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

Figure 4 Big step semantics for the relevant cases of the IMP language.

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 2, bottom. Note that the target proof object is independent of the source language and logic.

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 3), except for variables vs. stack slots ($\#k$) 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 [29]. Taking inspiration from his approach, we define a big step semantics for our languages. Our semantics judgments for IMP are 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

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

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

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

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

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

Figure 5 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 $|\sigma|$.

$$\begin{array}{rl} M ::= & T \mid F \mid p_n\ e_{\text{list}} \\ \mid & M \wedge M \mid M \vee M \end{array}$$

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

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

Figure 6 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.

expression, and a natural number if e is an arithmetic expression. Figure 4 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 (13).

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, and r is a stack if e is a statement. But if e is an arithmetic expression, r is a *tuple* of a stack and a natural number, since arithmetic expressions in Stack can have side effects (similarly for booleans). Figure 5 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 on its own private environment, function calls involve pushing variables and arguments onto the stack and later popping them. While 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. Later, we will utilize that our compiled programs and functions *do* behave nicely in this respect and thus do not interfere with the execution of other code. The semantics are formalized in Coq (14).

$$\boxed{\{P\}\, c\, \{Q\}}$$

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

$$\frac{\{P \wedge b\}\, i_1\, \{Q\} \quad \{P \wedge \neg b\}\, i_2\, \{Q\}}{\{P\}\ \mathrm{if}\ b\ \mathrm{then}\ i_1\ \mathrm{else}\ i_2\, \{Q\}}\ \textsc{ImpIf} \qquad \frac{\{P \wedge b\}\, i\, \{P\}}{\{P\}\ \mathrm{while}\ b\ \mathrm{do}\ i\, \{P \wedge \neg b\}}\ \textsc{ImpWhile} \qquad \frac{P_1 \Rightarrow P_2 \quad \{P_2\}\, i\, \{Q_1\} \quad Q_1 \Rightarrow Q_2}{\{P_1\}\, i\, \{Q_2\}}\ \textsc{RoC}$$

Figure 7 Hoare deduction rules for the Imp language, where RoC stands for "rule of consequence."

3.2 Specifications

Our program logics have the same base for assertions over the respective languages (Figure 6, 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 anyone writing proofs could use it in their specifications by defining a 2-ary predicate. At the same time, PotPie does not have to know extra mathematics.

The evaluation rules for assigning a truth value to a formula (Figure 6, right) define predicates as parameterized over the value types. For example, if we have the proposition $p_1\ a$ where a is a language expression that evaluates to v, p_1 then evaluates on v as a Coq proposition.

We can instantiate the framework described in Figure 6 using the arithmetic and boolean semantics relations as the evaluators. For Imp we refer to this instantiation as SM_e, and for Stack as TM_e. We then use this to construct the following specification grammars:

$$\begin{aligned} SM &::= SM_e \,|\, SM \wedge SM \,|\, SM \vee SM \\ TM &::= (n, TM_e) \,|\, TM \wedge TM \,|\, TM \vee TM \end{aligned} \tag{1}$$

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

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

For implementation details, see Section 5.1.

3.3 Proofs

Both Imp and Stack use a Hoare logic as their program logic, and we have proven them sound 16 17. 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.

Our Imp logic is standard (Figure 7; see the original paper on Hoare logic for more [18]). While later on we need to introduce some termination conditions (see Section 5.2) in order to properly compile the proofs, our Imp logic as a whole imposes no such condition.

$$\boxed{\{n, P\}\, c\, \{m, Q\}}$$

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

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

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

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

Figure 8 Hoare rules for the STACK language. RoC is omitted, as it is identical to the IMP rule.

The STACK proof rules (Figure 8) are similar to the IMP ones, except for obvious language differences (stack index accesses instead of variables, the addition of push and pop). 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 the output stack of the expression 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 5), there are no guarantees about side 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 all the changes introduced by expressions, as well as how to alter the Hoare logic rules to still keep them sound in such a setting.

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, this means that we have the following lemma:

▶ **Lemma 1** (Stack Preservation ①). *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 (such as the codomain of our compiler), 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.

4 Proof Compilation

Now that we have our programming languages, specification languages, and program logics defined, we describe in this section how we compile them in POTPIE.

In order to define our three compilers, we need to define an equivalence between the environments of IMP (i.e., the variable store σ and parameter array Δ) and the stacks of STACK (which we will represent by σ_s). 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 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 $1 \leq i \leq |V|$, we have $\sigma_s[i] = \sigma(\varphi^{-1}(i))$, and*
- *for $|V| + 1 \leq i \leq |V| + |\Delta|$, we have $\sigma_s[i] = \Delta[i - |V|]$.*

Note that this definition implies that $|V| + |\Delta| \leq |\sigma_s|$ while also saying nothing about the stack indices beyond $|V| + |\Delta|$. This will be important for ensuring that our compiler correctness theorems also work for evaluating function bodies.

Building on this mapping from environments to stacks, Sections 4.1, 4.2, and 4.3 describe our program, specification, and proof compilers, respectively.

4.1 Compiling Programs

Once we have a way to relate environments and stacks, our program compiler is fairly standard. Given Imp program $(\{f_1, \ldots, f_n\}, i_{\text{main}})$, where V contains the free variables of i_{main}, we can construct bijective $\varphi : V \to \{1, \ldots, |V|\}$. 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. While this compiler is simple, the transformation does allow us to examine how program logics compile across abstraction gaps. Compiling functions and function environments follows directly from program compilation.

We have proven the following correctness theorem for our simple compiler (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 1** (Compiler Correctness 15). *For Imp arithmetic expressions e_a, boolean expressions e_b, and commands c, function environment Λ, variable environment σ, and compiled function environment Λ', we have the following:*

- $\langle\, \sigma,\, \Lambda,\, \Delta,\, e_a \,\rangle \Downarrow_a v \wedge (\sigma, \Delta) \approx_\varphi \sigma_s \implies \langle\, \sigma_s,\, \Lambda',\, \text{comp}_{\text{expr}}(e_a) \,\rangle \Downarrow_a (\sigma_s, v)$
- $\langle\, \sigma,\, \Lambda,\, \Delta,\, e_b \,\rangle \Downarrow_a v \wedge (\sigma, \Delta) \approx_\varphi \sigma_s \implies \langle\, \sigma_s,\, \Lambda',\, \text{comp}_{\text{expr}}(e_b) \,\rangle \Downarrow_a (\sigma_s, v)$
- $\langle\, \sigma,\, \Lambda,\, \Delta,\, c \,\rangle \Downarrow_i \sigma' \wedge (\sigma, \Delta) \approx_\varphi \sigma_s \wedge (\sigma', \Delta) \approx_\varphi \sigma'_s \implies \langle\, \sigma_s,\, \Lambda',\, \text{comp}_{\text{code}}(c) \,\rangle \Downarrow_i \sigma'_s$

Because the relation $\approx_\varphi$ ignores the stack beyond index $|V| + |\Delta|$, 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. Further, if we assume termination of source programs, we get the backwards direction of compiler correctness for free via determinism: if a compiled program evaluates to a value, then the source program will evaluate to the same value (11, 12).

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 don't use the traditional simulation diagram to model the usual observable behavior mimicry.

4.2 Compiling Specifications

Once we have a program compiler and $\varphi : V \to \{1, \ldots, |V|\}$, compiling specifications is also relatively simple: recurse over the source logic formula and compile the leaves, i.e., Imp expressions. If k is then 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.

Table 1 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`.)

Category	Imp			Stack			Base Assertions	Compiler			**Total**
	Lang	Logic	WF	Lang	Logic	Frame		Code	Spec	Proof	
LOC	776	1,832	3,465	3,142	1,064	5,344	921	3,858	2,680	6,256	29,338
Theorems	10	56	98	82	16	204	37	109	54	165	831
Specs	43	34	51	55	49	49	30	34	32	117	494

For any given specification compilation function $\mathrm{comp}_{\mathrm{spec}}^{\varphi,k}$ for given φ and k, we want this $\mathrm{comp}_{\mathrm{spec}}^{\varphi,k}$ to be sound. In this context, sound means that if a variable and parameter environment satisfy a formula P, then we want the translation of those environments to also satisfy $\mathrm{comp}_{\mathrm{spec}}^{\varphi,k}(P)$. More formally:

▶ **Theorem 2.** *A specification compilation function $\mathrm{comp}_{\mathrm{spec}}^{\varphi,k}$ is sound if for all $P, \sigma, \Delta, \sigma_s$ such that $(\sigma, \Delta) \approx_\varphi \sigma_s$, we have $\sigma, \Delta \models P \Leftrightarrow \sigma_s \models \mathrm{comp}_{\mathrm{spec}}^{\varphi,k}(P)$.*

We have formalized that our translation of specifications is sound for surely terminating specifications in Coq as ⑧ and ⑨. Because of the way that our specifications are designed, this proof followed relatively straightforwardly from the proof of compiler correctness.

4.3 Compiling Proofs

Because the control flow of the compiled program mirrors that of the source, the structure of the Hoare tree can largely stay the same. Therefore, in order to compile a proof, we need to prove two more results: (1) the constructors for each Hoare rule in our Hoare proof tree types commute across compilation, and (2) the implications inside of the rule of consequence hold across compilation. These two conditions, in addition to the correctness of the specification compiler, are sufficient to prove our correct-by-construction proof compiler. More formally:

▶ **Theorem 3.** *Given an Imp Hoare proof of $\{P\}\, c\, \{Q\}$ (denoted as x), code compiler $\mathrm{comp}_{\mathrm{code}}$, and sound specification compiler $\mathrm{comp}_{\mathrm{spec}}$, a proof compiler $\mathrm{comp}_{\mathrm{pf}}$ is sound if $\mathrm{comp}_{\mathrm{pf}}(x)$ is a sound Stack Hoare proof of the triple $\{\mathrm{comp}_{\mathrm{spec}}(P)\}\, \mathrm{comp}_{\mathrm{code}}(c)\, \{\mathrm{comp}_{\mathrm{spec}}(Q)\}$.*

For the precise formulation of the statement of the proof compiler as well as a discussion of its implementation details, see Section 5.

5 Proof Engineering

This was a substantial proof engineering effort; Table 1 summarizes the number of lines of code, proofs, and specifications for different components of PotPie. Here we detail important design decisions (Section 5.1), assumptions (Section 5.2), and challenges (Section 5.3).

5.1 Design

We discuss five design decisions from engineering PotPie: (1) making PotPie correct by construction, (2) building modular assertion types, (3) separating language and logical reasoning, (4) using custom induction principles, and (5) building higher-order relations.

Correct-by-Construction Proof Compilation. We made the deliberate decision to make our proof compiler correct by construction. That is, PotPie is a function in Coq whose type specifies its correctness ⑦:

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

where the type above is simplified for clarity, and where the omitted conditions are described in Section 5.2. In other words, this 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 design for its elegance and simplicity. Notably, it echoes three approaches to encoding proofs in proof assistants:

1. The **LCF Approach**: The `hl` and `hl_stk` types that encode Imp and Stack Hoare logic proofs are each a datatype whose constructors are precisely the rules of the corresponding logic, hence all terms of that type must be valid proofs that are correct by construction—as with terms that have the ML `thm` type in the *LCF approach* [15].
2. The **de Bruijn Criterion**: These correct-by-construction proofs are themselves explicit proof terms in Coq's proof term language that represent Hoare trees, and that can be checked by a small proof checking kernel—thereby satisfying the *de Bruijn criterion* [3, 4].
3. The **Curry-Howard Correspondence**: Thanks to Coq's rich type system, we can encode the specification that each Hoare logic proof term proves as its type, by making the `hl` and `hl_stk` types dependent—taking advantage of (generalized) *Curry-Howard* [14, 19].

From these combined approaches, we get correct-by-construction, machine-checkable, explicit proof objects coupled with simple specifications that correspond to types in Coq. But this design is not without cost. Most notably, it tightly couples proofs and computation, which poses challenges for efficiency and reduction (see Section 5.3). We are currently weighing whether to decouple proofs and computation, considering what we may lose in the process.

Modular Assertions. Our implementation of assertions in Coq are modular, in that the datatype `LogicProp` that defines the syntax for assertions M is parameterized over two types: an expression type A and a value type V. As a result, we can provide any one of our expression types, its corresponding value type, and the semantics relation for that type, and end up with a model of this logic. Because of this modularity, the atoms SM_e and TM_e referenced in Section 3.2 are actually each two instantiations of the `LogicProp` framework, one each for arithemtic and boolean in both Imp and Stack.

Separating Language Reasoning and Logical Reasoning. Our system separates *language semantic* reasoning and *logical* reasoning. The language semantics reasoning occurs entirely inside of the expressions e that are arguments to the logical predicates p, whereas the logical reasoning occurs only once the expressions have been evaluated to values. This is needed because, when compilation changes the semantics of programs and programmatic functions are included in arithmetic statements, program semantics can no longer be abstracted away in specifications, as different languages may have different semantics. With this design, the proof engineer can separate their logical reasoning from the language. For example, if they need to use the fact that 5 is less than 7, then they can do so in the logical fragment. If they need to use the fact that the language expression $5 + 2$ evaluates to 7, then that is also expressible, and distinct from what it means purely mathematically.

Custom Induction Principles. One tried-and-true [44, 13] proof engineering design principle we employed was the use of custom induction principles. For example, for arithmetic expressions in IMP, we represented function arguments as lists of arbitrary lengths. We then realized that Coq's automatically generated induction principles were too weak. With the help of Clément Blaudeau [7], we defined a better induction principle ⑤:

```
∀ (P : aexp → Prop),
  (* ... other cases ... *) →
  (∀ (f : ident) (aexps : list aexp), Forall P aexps → P (f · aexps)) →
  ∀ (a : aexp), P a.
```

where `Forall P aexps` is inhabited whenever the inductive motive `P` holds on all elements of the list `aexps`. This produces a stronger inductive hypothesis in the function application case. We used a similar trick to define an induction principle for IMP that gives informative inductive hypotheses about arithmetic and boolean subexpressions. We hope, going forward, that third-party automation [43, 30] for producing better induction principles for containers will make it into Coq's standard library, sparing proof engineers this kind of effort.

Higher-Order Relations. We used higher-order relations (relations that take in other relations) to facilitate engineering and proving. This allowed us to lift relations from arithmetic and boolean expressions to the assertions those expressions were embedded within. From an engineering perspective, this reduced the number of names we had to remember, since we could simply lift expression-level relations without defining new names. From a proof perspective, it gave us some adequacy theorems "for free," by lifting proofs about expressions to the relation level. For example, this is how we proved Theorem 4, which allowed us to extend adequacy up to types that were parameterized on that type:

▶ **Theorem 4** (Higher-Order Adequacy For Free ②)**.** *If V, A are types and l, l' : $\texttt{LogicProp}\ V\,A$, for any $\varphi : A \to A$ and $\psi : A \to A \to$ `Prop`, if for all $a_1, a_2 \in A$ we have $a_2 = \varphi\ a_1$ if and only if $\psi\ a_1\ a_2$, then $l' = \texttt{transform\_prop\_exprs}\ l\ \varphi \Leftrightarrow \texttt{transformed\_prop\_exprs}\ \psi\ l\ l'$.*

5.2 Assumptions

POTPIE produces proof objects that are both correct by construction and checkable against the compiled specification. Its *trusted computing base* (TCB) consists of just the Coq kernel, the mechanized semantics, and two localized axioms for reasoning about dependent types. Its compiler correctness proof provides some confidence that the semantics are implemented correctly, at least with respect to one another. To keep the TCB small, it takes as input additional proof obligations, which we plan to relax in the near future.

Localized Axioms. 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*. We place this restriction on UIP because assuming UIP in general would be computationally undesirable, since UIP does not have a constructive interpretation in Coq. We assume UIP for `AbsEnvs` (the Coq implementation of SM in Equation 1) and function environments.

▶ **Axiom 1** (UIP for `AbsEnvs` ③). *If a, a' : `AbsEnv`, then if $p_1, p_2 : a = a'$, we have $p_1 = p_2$.*

▶ **Axiom 2** (UIP for Function Environments ④). *If Λ, Λ' are 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 ever compute on the contents of `AbsEnvs`; they really are computationally irrelevant. 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 ⑩, UIP provably holds for V and A, so Axiom 1 does not imply UIP more generally. 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 UIP more generally.

We may be able to avoid these axioms by adapting the definitions of `AbsEnv` and function environments, or by rephrasing some inductive relations as `Fixpoints` that project into `Prop`. We have done the latter for one of our later theorems, but have found adapting earlier definitions and relations to be invasive. Assuming UIP over these two types made it possible to write proofs by dependent induction with minimal changes to our proof development.

Proof Obligations. To minimize our TCB, PotPie requires some proof obligations from the caller, beyond just the source proof that is being compiled. One such proof obligation is to prove that Hoare proofs must have only surely terminating assertions. This simplifies the backwards program compiler correctness proof, which follows from forward correctness and determinism. It also simplifies the semantics of our Hoare logic. However, this comes with a tradeoff. Beyond the additional user proof obligation imposed, this assumption also prohibits calling functions that may have non-terminating behavior, ruling out entire classes of useful proofs. We hope to sort out the semantics of our Hoare logic in the face of potentially nonterminating preconditions and postconditions so that we can relax this in the future.

We also take well-formedness of the source code and function environments as proof obligations, requiring that, for example, functions are called on the correct number of arguments. We plan to relax this assumption as well. One potential way to relax this assumption is to replace our current program compiler with one that rejects ill-formed source programs. Another potential way is to prove an adequacy condition that lets us loosen the proof obligation to proving an equality with the result of calling our current compiler. These assumptions are documented in the assumptions required to call the proof compiler ⑦.

5.3 Challenges

In developing PotPie, we encountered several interesting challenges, both specific to our project and also more broadly related to Coq. Here we describe some of these challenges and how we addressed them: obtaining source-independent proofs, efficiently compiling proofs, expanding our source logic, and handling mostly-correct compilers.

Getting Source-Independent Proofs. One major benefit of compiling proof objects from a source logic to a target logic is that the compiled proofs can be free of references to the source language and logic. We are well on our way to source-independence—but fully achieving this introduces its own challenges. For example, due to the tight coupling of our correctness proof and computation in PotPie, source code well-formedness principles may show up in compiled proofs. In addition, opaque proofs in Coq 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.

Efficient Proof Compilation. One of the factors that prevented us from computing source-independent proofs was the inefficiency of Coq's term reduction, especially when combined with proof script automation. Proof script automation through specialized tactics can be effective for reducing engineering effort. But such automation rarely creates simple proof terms, which is important when computation matters, as it does for PotPie. For example, automation often introduces highly nested proof terms that may take minutes to reduce. In one case, reducing such a term even crashed CoqIDE!

One way to get more efficient computation is to use opaque proofs, which is the default in Coq when one uses `Qed` instead of `Defined` to end a proof. For us, this was often not an option, since opaque proofs block reduction and thus computation. We needed transparent proof terms since computation was relevant, so we often used `Defined`. The only way to ensure our terms computed while staying transparent was to *manually* optimize proof terms. We used this process to do so:

1. Prove a given theorem using automation.
2. Print the term automatically generated by the proof script for that theorem. Locate inefficiencies in that term, and match those inefficiencies to tactics in the proof script.
3. Simplify the term, typically by working at the tactic level.
4. Repeat until the term efficiently reduces.

The details varied by the source of inefficiency. For example, we often had to simplify proof terms generated using the `inversion` tactic, which performs (possibly dependent) pattern matching to pull premises into the proof state. Consider the `WHILE` case of the mutually inductive well-formedness relation for function applications ⑤:

```
| fun_app_while : ∀ fenv wf_funcs b i,
    fun_app_bexp_well_formed fenv wf_funcs b → (* If the guard b is well formed, *)
    fun_app_imp_well_formed fenv wf_funcs i → (* and so is the expression i, *)
    fun_app_imp_well_formed fenv wf_funcs (WHILE b i). (* then so is the loop. *)
```

In proofs with hypotheses applying `fun_app_imp_well_formed` to some loop `WHILE b i`, it was helpful to invert those hypotheses to derive the well-formedness of the guard `b` and the Imp expression `i`. This pattern was so common that, for large relations like this one, we had defined custom inversion tactics. These custom tactics very quickly led to case explosion. To optimize case-exploding proofs, we refactored inversion into opaque lemmas, like ⑥:

```
Lemma inv_fun_app_imp_while : ∀ fenv funcs b i,
  fun_app_imp_well_formed fenv funcs (WHILE b i) →
  fun_app_bexp_well_formed fenv funcs b ∧ fun_app_imp_well_formed fenv funcs i.
```

We then left the rest of those proofs transparent. This workflow brought down the reduction time of the term that had previously crashed CoqIDE to the order of seconds.

Some of our reduction woes may be preventable by replacing some of our inductive relations with `Fixpoints` that project into `Prop`. We would also like a notion of semi-opaque subterms, on which Coq's full reduction (normalization) process could act like the `simpl` tactic (by *refolding* [8] constants) without ever fully reducing. Going forward, we would love more work on making Coq's reduction faster, easier to control, and simpler to debug. Finally, we would love tools for automatic proof term optimization.

Expanding Our Stack Logic For convenience of implementation, the Stack program logic semantics we introduced in Section 3.2 is limited in the programs it can describe. While it is sufficient to handle proofs compiled from Imp, we would eventually love to be able to link those proofs with other Stack programs' proofs. To do this, we plan to derive a program logic semantics that fully describes what happens with side effects for Stack. For example, if we know that a function may access up to the fifth index outside of its stack frame, we could remove all knowledge in our assertions having to do with the affected stack indices. We could even define our current Stack logic semantics as a special case of the richer semantics, so proofs produced by PotPie could still interface with proofs that use the full Stack logic.

Handling Sometimes-Incorrect Compilation PotPie currently assumes that the program compiler itself is verified. This is a reasonable start, but we would love to avoid the need for a verified program compiler. In fact, we suspect that one of the key benefits of proof compilation ought to be handling compilation that is sometimes *incorrect*—for example, unsafe optimizations, so long as they do not impact the truth of the compiled specification.

We attempted to build PotPie without this assumption initially, but we found this challenging. In order to translate the rule of consequence, PotPie as implemented must ensure that arbitrary logical implications hold through the logic translation, which requires a proof of compiler correctness; this is the only place in which that proof is needed.

This challenge arises from including functions calls in arithmetic expressions. Unlike in traditional uses of Hoare logic, the semantics of arithmetic do not entirely line up with our semantics—one must also reason about the control flow of functions. This means that the implications inside an application of consequence cannot be proven just using arithmetic, but rather must reason about the semantics of the program. When compiling programs, the semantics and level of abstraction change, and so soundly compiling proofs relies on those semantics changing soundly. We have three options: (1) add a proof obligation that all source and compiled implications hold, (2) rely on a proof of compiler correctness, or (3) rework the rule of consequence so that it no longer requires a proof of compiler correctness.

For now, we choose option (2), since it is easy to generalize and, unlike option (1), avoids exposing confusing proof obligations about target proofs to users. One path toward option (3) that we are considering is specifying that all implications must be of the form of a Hoare triple for a function. Such a change would allow us to view all expressions inside of our specifications exclusively as deeply embedded language constructs and eject all Coq-level reasoning to the relation in our predicate construct. The end result would still require some target-level proof obligations from the user, but those proof obligations would be simpler than those from option (1). We will continue to investigate option (3), alongside ways to relax the proof obligations in option (1), to later smoothly handle sometimes-incorrect compilation.

6 Related Work

Proof-Transforming Compilation. Early work compiling proofs positions itself as an extension of *proof-carrying code* [35], in which the compiler produces proofs of important properties about compiled code. Work in 2006 [5] 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 [33] 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 [37] 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 [17] 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.

Certificate translation [25] 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 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 [38]. They have since been used for many different purposes, like automatic generalization [16, 22, 20], reuse [31], and repair [40] 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 [28, 27], which was proven correct in Coq, and has been shown to lack bugs present in other compilers [45]. The CakeML [24] verified implementation of ML in HOL4 includes a verified compiler in HOL4. Oeuf [32] and CertiCoq [2] 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 [1, 42]. A comprehensive survey of work in certified and certifying compilation is in QED at Large [41].

Our work is both certifying (the proof compiler produces a target proof object, showing 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. *Rupicola* [39], 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, producing target proof objects.

Proof assistants are not the only way to certify correct compilation. *Translation validation* is a method of certifying that a compiler preserves the behavior of the original source program for a particular compilation run. Through the use of simulation relations, one can check whether two programs are semantically equivalent [36]. Our work instead focuses on producing proof objects that can be checked in a target program logic.

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 Xavier Leroy's course materials [29] as a base. Frameworks based on embedded program logics have seen a lot of success in helping proof engineers write proofs in a proof assistant about code with features that the proof assistant lacks. Some example frameworks for reasoning with embedded program logics are Iris [21, 23], VST-Floyd [9], Bedrock [11, 12], YNot [34], CHL [10], and SEPREF [26].

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) [6]. 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 producing a proof object at the target level—one that is still guaranteed to relate correctly to the source proof of the source program. In contrast, chaining program logics with certified compilers preserves guarantees about compiled code, but does not produce a target proof object. We hope that producing target proof objects will help overcome barriers for understanding and checking target proofs independently, forming composite guarantees about compiled components written in different source languages, optimizing and repairing compiled proofs, and keeping proofs about compiled code immune to changes in the source language or logic.

For now, relative to the program logics found in practical frameworks like Iris and VST, our embedded program 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 program logics and lower-level target languages in the future, while still producing correct proof objects at the target level.

7 Conclusions & Future Work

We defined and formally verified a correct-by-construction proof compiler, POTPIE, in Coq. POTPIE produces proofs about the target STACK language, given proofs about the source IMP language. Both proofs about IMP and their compiled STACK counterparts are explicit, machine-checkable proof terms in Coq, guaranteed to prove their respective specifications, and to be correctly related. This is a major step toward allowing proof engineers to reason directly about source programs, and still obtain source-independent proofs about compiled programs. We believe this opens up two particularly promising directions for future work:

1. **Linking proofs**: What if proof engineers could write proofs about components of systems implemented in different languages, and easily get machine-checkable proofs at the target level about the entire system? This is the equivalent of linking for proofs, and proof compilation offers a path toward it. The main barriers left are removing the remaining source-dependent fragments of compiled proofs, implementing proof compilers for multiple languages, and reasoning about the interactions between compiled components.
2. **Correctness in the face of unsafe compilation**: What if proof engineers could write proofs about source programs, and from those proofs derive strong guarantees about compiled programs, even when the compiler itself is sometimes unsafe? With proof compilation, there is a chance for freedom from relying on certified and certifying program compilers, without sacrificing satisfaction of important specifications. The one thing

standing in the way is the dependency on full program compiler correctness that our proof compiler imposes to support the rule of consequence; this is at least in part surmountable. We are excited to explore these directions.

References

1 Sidney Amani, Alex Hixon, Zilin Chen, Christine Rizkallah, Peter Chubb, Liam O'Connor, Joel Beeren, Yutaka Nagashima, Japheth Lim, Thomas Sewell, Joseph Tuong, Gabriele Keller, Toby Murray, Gerwin Klein, and Gernot Heiser. Cogent: Verifying high-assurance file system implementations. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 175–188, Atlanta, GA, USA, April 2016. doi:10.1145/2872362.2872404.

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

3 Henk Barendregt and Erik Barendsen. Autarkic computations in formal proofs. *Journal of Automated Reasoning*, 28(3):321–336, 2002. doi:10.1023/A:1015761529444.

4 Henk Barendregt and Freek Wiedijk. The challenge of computer mathematics. *Philosophical Transactions of the Royal Society of London A: Mathematical, Physical and Engineering Sciences*, 363(1835):2351–2375, 2005. doi:10.1098/rsta.2005.1650.

5 Gilles Barthe, Tamara Rezk, and Ando Saabas. Proof obligations preserving compilation. In *Formal Aspects in Security and Trust: Thrid International Workshop, FAST 2005, Newcastle upon Tyne, UK, July 18-19, 2005, Revised Selected Papers*, pages 112–126. Springer, 2006.

6 Lennart Beringer and Andrew W. Appel. Abstraction and subsumption in modular verification of C programs. *Formal Methods in System Design*, 58(1):322–345, October 2021. doi:10.1007/s10703-020-00353-1.

7 Clément Blaudeau. Untitled, 2022. URL: https://pastebin.com/BAvg3Jdh.

8 Pierre Boutillier. *De nouveaux outils pour calculer avec des inductifs en Coq*. Theses, Université Paris-Diderot - Paris VII, February 2014. URL: https://theses.hal.science/tel-01054723.

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

10 Haogang Chen, Daniel Ziegler, Tej Chajed, Adam Chlipala, M. Frans Kaashoek, and Nickolai Zeldovich. Using crash hoare logic for certifying the fscq file system. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, page 18–37, New York, NY, USA, 2015. Association for Computing Machinery. doi:10.1145/2815400.2815402.

11 Adam Chlipala. Mostly-automated verification of low-level programs in computational separation logic. *SIGPLAN Not.*, 46(6):234–245, jun 2011. doi:10.1145/1993316.1993526.

12 Adam Chlipala. The bedrock structured programming system: Combining generative metaprogramming and hoare logic in an extensible program verifier. *SIGPLAN Not.*, 48(9):391–402, sep 2013. doi:10.1145/2544174.2500592.

13 Adam Chlipala. Formal reasoning about programs, 2017. URL: http://adam.chlipala.net/frap/.

14 H. B. Curry. Functionality in combinatory logic. *Proceedings of the National Academy of Sciences of the United States of America*, 20(11):584–590, 1934. URL: http://www.jstor.org/stable/86796.

15 Michael J. C. Gordon, Robin Milner, and Christopher P. Wadsworth. Edinburgh LCF: A mechanised logic of computation. In *Lecture Notes in Computer Science*, volume 78. 1979. doi:10.1007/3-540-09724-4.

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

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

18 C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, oct 1969. doi:10.1145/363235.363259.

19 W. A. Howard. *The formulae-as-types notion of constructions.* Academic Press, 1980.

20 Einar Broch Johnsen and Christoph Lüth. Theorem reuse by proof term transformation. In *Theorem Proving in Higher Order Logics: 17th International Conference, TPHOLs 2004, Park City, Utah, USA, September 14-17, 2004. Proceedings*, pages 152–167. Springer, Berlin, Heidelberg, 2004. doi:10.1007/978-3-540-30142-4_12.

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

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

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

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

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

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

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

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

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

30 Bohdan Liesnikov, Marcel Ullrich, and Yannick Forster. Generating induction principles and subterm relations for inductive types using metacoq. *arXiv preprint arXiv:2006.15135*, 2020.

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

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

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

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

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

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

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

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

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

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

41 Talia Ringer, Karl Palmskog, Ilya Sergey, Milos Gligoric, and Zachary Tatlock. Qed at large: A survey of engineering of formally verified software. *Foundations and Trends® in Programming Languages*, 5(2-3):102–281, 2019. URL: `http://dx.doi.org/10.1561/2500000045`, `doi:10.1561/2500000045`.

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

43 Enrico Tassi. Deriving Proved Equality Tests in Coq-Elpi: Stronger Induction Principles for Containers in Coq. In John Harrison, John O'Leary, and Andrew Tolmach, editors, *10th International Conference on Interactive Theorem Proving (ITP 2019)*, volume 141 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 29:1–29:18, Dagstuhl, Germany, 2019. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. URL: `http://drops.dagstuhl.de/opus/volltexte/2019/11084`, `doi:10.4230/LIPIcs.ITP.2019.29`.

44 Doug Woos, James R. Wilcox, Steve Anton, Zachary Tatlock, Michael D. Ernst, and Thomas Anderson. Planning for change in a formal verification of the Raft consensus protocol. In *Proceedings of the 5th ACM SIGPLAN Conference on Certified Programs and Proofs*, CPP 2016, pages 154–165, New York, NY, USA, 2016. ACM. `doi:10.1145/2854065.2854081`.

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