# 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

Guaranteeing correct compilation is nearly synonymous with compiler verification. However, compiler verification is still an intensive endeavor, and the correctness theorems proved for certified compilers can be stronger than we need. While many compilers do have incorrect behavior, many programmers do not encounter these behaviors. Even when a compiler bug, rather than a code one, occurs it may not change the program's behavior meaningfully with regards to its specification. Many real-world specifications are necessarily partial in that they do not completely specify all of a program's behavior. While compiler verification and formal methods have had great success for safety-critical systems, there are magnitudes more code, such as math libraries, compiled with unverified compilers, that would benefit from a guarantee of its partial specification.

This paper explores a technique to get guarantees about compiled programs even in the presence of an unverified, or even incorrect, compiler. Our workflow compiles programs, specifications, and proof objects, from an embedded source language and logic to an embedded target language and logic. We implement such a proof compiler in Coq, moving from a simple 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 five variants of a program compiler: one that is incomplete, two that are incorrect, one that is correct but unverified, and one that is correct and verified. We are also able to leverage compiled proofs to assist in proofs of larger programs. 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.

## 1 INTRODUCTION

Program logic frameworks help proof engineers do more advanced reasoning about program-specific properties, such as effects or concurrency. Iris [Jung et al. 2015; Krebbers et al. 2017], VST [Cao et al. 2018], CHL [Chen et al. 2015], and Sepref [Lammich 2019] are just a few examples of such program logics. Typically, if you want to get strong guarantees for compiled programs, you need to chain a program logic proof with a verified compiler [Cao et al. 2018]. However, there are three related drawbacks to this workflow:

(1) Many compilers have bugs or incorrect behavior: gcc keeps a live tally of bug reports [GCC 2023], and many new ones are created every day. Users of gcc can even opt into unsound optimizations such as `-ffast-math` that have cases of incorrect behavior [Byrne 2023].
(2) Compiler verification is expensive (and impossible for an incorrect compiler): CompCert spans more than 100,000 lines of Coq code and six person-years of effort [AbsInt 2023].
(3) Many partial specifications *do not require* global compiler correctness about all programs and can be resilient to minor compiler bugs.

In short, sometimes totally correct compilation is unnecessarily strong. In order to remove the requirement for totally correct compilation, we present the *proof compiler* PotPie. PotPie takes a program, specification, and proof of the specification and compiles all three such that the specification's meaning is preserved, and compiled proof shows that the compiled program meets the

* Co-first authors

compiled specification. PotPie was formally verified in Coq and is parametrized over the choice of compilers for code and specifications (which we will refer to as the code or program compiler and the specification compiler, respectively). This work is a kind of certifying compilation that certifies program-specific properties rather than general ones, such as type or memory safety (for a more detailed discussion of how they relate, please see Section 9).

To get a sense of how the PotPie differs from similar techniques, imagine a proof engineer wants to show a program-specific specification $P$ about their program $c$ in language $A$, which gets compiled to a property $P'$ about program $c'$ in language $B$. Using PotPie, they first prove that $P$ holds for $c$ using the program logic for language $A$. Then, they prove a few proof obligations required by PotPie, which are about the well-formedness of programs and Hoare trees and are mostly syntactic, and therefore automatable. Finally, they pass their choice of code and specification compilers to PotPie, as well as their proof of $P$, their program $c$, and the proof obligations. As long as the proof engineer is able to prove the obligations, PotPie is *guaranteed* to return a valid proof in the program logic for language $B$ that $P'$ holds for $c'$. Nowhere in this workflow does the code or specification compiler need to be proven correct. Further, because PotPie compiles proofs of user defined *partial* specifications, incorrect compiler behavior sometimes does not break the desired specification of the program and thus can be compiled despite the incorrect behavior.

This paper makes the following contributions:

(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. In lieu of full program compiler correctness, we have several compiler and user proof obligations (Section 5). Some of these proof obligations are limitations on the kinds of optimizations that can be performed, such as control flow-altering ones (Section 5.1). This was a simplifying decision we made to make the problem initially tractable, but we do not believe it is inherent to our approach.
(2) We demonstrate PotPie on case studies, using an incorrect code compiler to *correctly* compile proofs of programs for multiplication, exponentiation, infinite series approximation, and square root approximation (Section 6), with further examples in our code artifact. Further, we demonstrate how we can utilize *previously compiled proofs* to *verify additional programs*.
(3) We prove PotPie sound, both informally (Section 7) and formally in Coq (Section 8). Specifically, whenever PotPie's proof compiler is called and the proof obligations are met, it is guaranteed to return a valid proof in the target language program logic.
(4) We discuss of how PotPie differs from similar techniques, such as translation validation and certifying compilation (Section 9), as well as additional work in this space.

## 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 is called with its proof obligations, the compiled proof proves the compiled specification about the compiled program, and the compiled specification 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 demonstrates this with a *correct* program compiler, and we elide the user proof obligations required to invoke the proof compiler (as they are always satisfiable when the program compiler is correct). For the user proof obligations, see Section 5; for examples of correctly compiled proofs with an *incorrect* compiler, see Section 6.

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.

$$\dfrac{\dfrac{(\top \wedge p_0 > p_1) \Rightarrow (p_0 \le p_0 \wedge p_1 \le p_0) \quad \dfrac{}{\{p_0 \le p_0 \wedge p_1 \le p_0\}\ \texttt{z := p0}\ \{p_0 \le z \wedge p_1 \le z\}}\ \textsc{Assign}}{\{\top \wedge p_0 > p_1\}\texttt{z := p0}\{p_0 \le z \wedge p_1 \le z\}}\ \textsc{RoCLeft} \quad \dfrac{\top \wedge \neg(p_0 > p_1) \Rightarrow (p_0 \le p_1 \wedge p_1 \le p_1) \quad \dfrac{}{\{p_0 \le p_1 \wedge p_1 \le p_1\}\ \texttt{z := p1}\ \{p_0 \le z \wedge p_1 \le z\}}\ \textsc{Assign}}{\{\top \wedge \neg(p_0 > p_1)\}\texttt{z := p1}\{p_0 \le z \wedge p_1 \le z\}}\ \textsc{RoCLeft}}{\{\top\}\ \texttt{if p0 > p1 then z := p1 else z := p0}\ \{p_0 \le z \wedge p_1 \le z\}}\ \textsc{If}$$

$$\dfrac{\dfrac{(3, \top \wedge \#2 > \#3) \Rightarrow (3, \#2 \le \#2 \wedge \#3 \le \#2) \quad \dfrac{}{\{3, \#2 \le \#2 \wedge \#3 \le \#2\}\ \#1\ :=\ \#2\ \{3, \#2 \le \#1 \wedge \#3 \le \#1\}}\ \textsc{Assign}}{\{3, \top \wedge \#2 > \#3\}\ \#1\ :=\ \#2\ \{3, \#2 \le \#1 \wedge \#3 \le \#1\}}\ \textsc{RoCLeft} \quad \dfrac{(3, \top \wedge \neg(\#2 > \#3)) \Rightarrow (3, \#2 \le \#3 \wedge \#3 \le \#3) \quad \dfrac{}{\{3, \#2 \le \#3 \wedge \#3 \le \#3\}\ \#1\ :=\ \#3\ \{3, \#2 \le \#1 \wedge \#3 \le \#1\}}\ \textsc{Assign}}{\{3, \top \wedge \neg(\#2 > \#3)\}\ \#1\ :=\ \#3\ \{3, \#2 \le \#1 \wedge \#3 \le \#1\}}\ \textsc{RoCLeft}}{\{3, \top\}\ \texttt{if \#2 > \#3 then \#1 := \#2 else \#1 := \#3}\ \{3, \#2 \le \#1 \wedge \#3 \le \#1\}}\ \textsc{If}$$

$$\dfrac{\dfrac{\dfrac{}{\{2, \top\}\texttt{push}\{3, \top \wedge \#1 = 0\}}\ \textsc{Push} \quad (3, \top \wedge \#1 = 0) \Rightarrow (3, \top)}{\{2, \top\}\ \texttt{push}\ \{3, \top\}}\ \textsc{RoC} \quad \vdots}{\{2, \top\}\ \texttt{push; if \#2 > \#3 then \#1 := \#2 else \#1 := \#3}\ \{3, \#2 \le \#1 \wedge \#3 \le \#1\}}\ \textsc{Seq}$$

Fig. 1. The Hoare tree for the program max (above) and the compiled tree (below). Here, $p_i$ and pi represent the $i^{\text{th}}$ 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 mean the $n^{\text{th}}$ stack index. Stacks are 1-indexed, and the top of the stack is #1. RoC stands for "rule of consequence."

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, 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.

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 $\text{comp}_{\text{pf}}$ 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.

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

Fig. 2. Imp (left) and Stack (right) language syntax, where $a$ describes arithmetic expressions, $b$ boolean expressions, $i$ imperative statements, $\lambda$ 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.

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

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

## 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 proof compiler, yet still have interesting properties and changes in abstraction. 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. Both languages have a big step semantics[1].

Our semantics judgments for Imp are of the form $\langle \sigma, \Lambda, \Delta, e \rangle \Downarrow r$, where $\sigma$ is the variable environment, $\Lambda$ is the function environment, and $\Delta$ 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 function calls can occur inside other 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. 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 $\Lambda$ is again the function environment, but $\sigma$ 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

[1]Imp and Stack's basic infrastructure is loosely based on materials from a course on mechanized semantics [Leroy 2021].

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

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

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

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

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

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

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

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

Fig. 5. Syntax (left) and semantics (right) for base assertions for both Imp and Stack. $map\_eval_\sigma$ is a relation from lists of expressions to lists of values. The semantic interpretation is parametric over the types of $v$, $\sigma$, and $map\_eval_\sigma$. Interpretations for $\wedge$ and $\vee$ are standard.

inference rules for Stack that are either non-standard, or that are different from Imp's semantics. Stack adds several complexities—in Stack, 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.

### 3.2 Specifications

The assertions for both languages follow the same syntactic structure, given in 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. First, it makes the logic compilation very easy. In order to compile these assertions, all that has to be done is to compile the expressions. Second, 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 a predicate.

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 an Imp expression that evaluates to $v$, then $p_1\ a$ is true if and only if calling the Coq definition of $p_1$ with $v$ is a true `Prop`.

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:

$$SM ::= SM_e \mid SM \wedge SM \mid SM \vee SM \tag{1}$$
$$TM ::= (n, TM_e) \mid TM \wedge TM \mid TM \vee TM$$

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

Fig. 6. The two nonstandard Hoare rules in our source language. RoC stands for "rule of consequence." $I$ here is the implication database satisfying Definition 3.1.

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:

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

The reader may note that function calls are allowed within preconditions and postconditions. This design decision is not essential to our approach—one could easily disallow effectful constructs from expressions as in CLight [Blazy and Leroy 2009] and proceed from there. However, for the current framework, we find it more natural to reason about effectful expressions in Imp.

### 3.3 Proofs

Both Imp and Stack use a Hoare logic [Hoare 1969] as their program logic, and we have proven them sound in Coq. Our Imp logic (see Figure 6) is standard modulo the fact that we modify the rule of consequence to be split two separate rules (one for weakening and one for strengthening), where all implications used in applications of these rules must draw from an *implication database*, which is a list of pairs of specifications. This list must satisfy the following definition:

*Definition 3.1.* An implication database $I$ is valid if, for each pair (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). The main way that the Stack rules differ from their Imp counterparts is the preservesStack premise in the Assign, If, and While cases. For any expression $e$, preservesStack $e$ says that all function call subexpressions of $e$ do not modify the rest of the stack frame. 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 a lack of effects. 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 $\sigma, \sigma'$, 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

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

$$\frac{}{\{A\}\ \text{skip}\ \{A\}}\ \textsc{StkSkip} \qquad \frac{}{\{n, P\}\ \text{push}\ \{n+1, \text{inc}(P) \wedge \#1 = 0\}}\ \textsc{StkPush}$$

$$\frac{}{\{(n+1, \text{inc}(P) \wedge Q)\}\ \text{pop}\ \{(n, P)\}}\ \textsc{StkPop} \qquad \frac{\text{preservesStack}(a, n)}{\{n, P[\#k \rightarrow a]\}\ \#k := a\ \{n, P\}}\ \textsc{StkAssign}$$

$$\frac{\{n_1, P\}\ i_1\ \{n_2, Q\} \quad \{n_2, Q\}\ i_2\ \{n_3, R\}}{\{n_1, P\}\ i_1; i_2\ \{n_3, R\}}\ \textsc{StkSeq} \qquad \frac{\text{preservesStack}(b, n) \quad \{n, P \wedge b\}\ i\ \{n, P\}}{\{n, P\}\ \text{while}\ b\ \text{do}\ i\ \{n, P \wedge \neg b\}}\ \textsc{StkWhile}$$

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

Fig. 7. Hoare rules for Stack. The left and right rules of consequence are omitted, as they are identical to the Imp rules. Stack does not need to be compiled, and thus has access to the full rule of consequence.

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 to get strong guarantees at the target level, even if 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 describe the compiler and user proof obligations in Section 5, and define soundness and prove this approach sound in Section 7.

### 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, for each code compiler, we define an equivalence between variable environments and stacks so "sound translation" is a well-defined concept.

*Definition 4.1.* Let $V$ be a finite set of variable names, and let $\varphi : V \rightarrow \{1, \ldots, |V|\}$ be bijective with inverse $\varphi^{-1}$. Then for all variable stores $\sigma$, parameter stores $\Delta$, and stacks $\sigma_s$, we say that $\sigma$ and $\Delta$ are $\varphi$-equivalent to $\sigma_s$, written $(\sigma, \Delta) \approx_\varphi \sigma_s$, if (1) for $1 \leq i \leq |V|$, we have $\sigma_s[i] = \sigma(\varphi^{-1}(i))$, and (2) for $|V| + 1 \leq i \leq |V| + |\Delta|$, we have $\sigma_s[i] = \Delta\,[i - |V|]$.

This equivalence is entirely dependent on our choice of mapping between variables and stack slots. It has this form since parameters are always at the top of the stack at the beginning of a function call, and are then pushed down as space for local variables is allocated, so parameters are "after" (i.e., appended to the end of) the local variables. Note that this implies $|V| + |\Delta| \leq |\sigma_s|$ 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 6, capturing both incorrect and correct program compilation. All

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

Fig. 8. An arithmetic expression compiler $\mathrm{comp}_{\mathrm{a}}$ (left) and a boolean expression compiler $\mathrm{comp}_{\mathrm{b}}$ (right).

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

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

build on common infrastructure to translate Imp arithmetic and boolean expressions (Figure 8). This infrastructure is a straightforward extension of the variable mapping function $\varphi$ 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 $\varphi$ plus $k$, where $k$ is the number of arguments.

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 $\varphi$, $k$, and an assertion $P$, we want to be able tell when $\mathrm{comp}_{\mathrm{spec}}^{\varphi,k}$ 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 $\mathrm{comp}_{\mathrm{spec}}^{\varphi,k}(P)$ and vice versa.[2]

To call the proof compiler, the user must prove a proof obligation that shows the specification compiler is sound with respect to the user's source proof (see Section 5). This ensures that the compiled proof actually proves the analogous property even when the program compiler is incorrect.

### 4.4 Compiling Proofs

The proof compiler is defined recursively over the Hoare proof data type, parametrizing over the program and specification compilers. While the proof compiler is fairly simple, the program compiler must commute with the Hoare rules' syntactic assertion transformations. This manifests as a compiler proof obligation; we describe this and other proof obligations in the next section.

[2]The "if and only if" prevents situations where all specifications compile to true, which leads to a trivial proof of nothing.

## 5 PROOF OBLIGATIONS

PotPie implements a kind of specification-preserving compilation that is robust to incorrect compilation. There are two costs to this: compiler proof obligations and user proof obligations. Compiler proof obligations are needed to ensure that for a subset of programs, the program compiler commutes with respect to the compiled specifications (Section 5.1). It is up to the user, however, to check that compiled specifications are *meaningful* (Section 5.2). The user must also prove well-formedness conditions that are specific to our choice of languages and program logics (Section 5.3). For example, compiled functions are not allowed to alter the call stack outside of their frame, which is required by the preservesStack relation in Stack logic (see Figure 7). Even taken together, these proof obligations do not require full semantic preservation; they allow for some miscompilation of programs so long as that compilation does not break the (possibly partial) specification. In this way, PotPie captures a middle ground between translation validation and certified compilation.

### 5.1 Commutativity Equations

The code and specification compiler proof obligations concern how compiled programs and specifications relate, and must be proved to create a new proof compiler from a code compiler. We first define a subset of *valid* Imp in terms of the programs and expressions for which the functions $\text{check}_{\text{code}} : i \to \mathbb{B}, \text{check}_{\text{a}} : a \to \mathbb{B}, \text{check}_{\text{b}} : b \to \mathbb{B}$ evaluate to true. A similar function, $\text{check}_{\text{spec}}$, should be defined for the specification compiler. To satisfy this obligation, 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 some $P$ for which $\text{check}_{\text{spec}}(P) = \texttt{true}$, in order to compile an application of the assignment rule, we want (2) in Figure 10 to hold. If we have this equality, we have the following, where $P' = \text{comp}^{\varphi,k}_{\text{spec}}(P)$:

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

As long as the code compiler commutes with these Hoare-rule induced syntactic transformations and the user proof obligations are satisfied, 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 relevant to this compiler proof obligation can be found in Figure 10.

As a consequence of our 1-to-1 Hoare rule translation, the source and target Hoare rules must actually be *in* 1-to-1 correspondence. Proof obligations (4)-(8) require that the source and target Hoare trees are structurally the same. In a less formal sense, this means that the control flow of the source and compiled programs must be the same. While constraining, this limitation does not seem fundamental to our approach. Because the important, largely nontrivial pieces of the translation lie only in the applications of the rules of consequence, other control flow operations could in theory change so long as they link up at the applications of the rule of consequence. Additionally, careful relaxation of Definition 4.1 could allow for changes such as common subexpression elimination.

Aside from the control flow limitations, the main reason for the equations is to ensure that the specification compiler is "aware" of the way that arithmetic and boolean expressions are compiled. For example, consider the case where a code compiler adds 1 to every arithmetic statement after the arithmetic compiler compiles down these expressions. This immediately breaks the compilation of all of the assignment rule applications, as the specification compiler is "unaware" of a transformation that affects a Hoare rule application. Equations 2-4 and 7-8 in Figure 10 are to prevent such cases.

In practice, so long as the program compilers are executable, these conditions tend to be dispatchable with simple equality proofs in Coq.

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

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

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

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

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

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

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

Fig. 10. Equations the specification and program compilers must satisfy before applying a proof compiler.

### 5.2 Specification Translation Conditions

This set of user proof obligations has to do with ensuring that PotPie not just compiles proofs, but that it does so *meaningfully*. That is, we would like PotPie to compile source program proofs to target program proofs that, in some sense, *prove the same thing*. We introduce *specification translation conditions* to make this possible.

The key technical insight that lets 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. 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. Unfortunately, automatically compiling consequence would require compiler correctness, since translating arbitrary implications requires a fully sound semantic translation, i.e. compiler correctness.

Our solution is to have the user specify which implications they are using in their Hoare proof, and then manually prove that these implications translate soundly across compilation. More formally, we require that implications in our Imp implication database have the following property:

*Definition 5.1.* An Imp implication $P \Rightarrow Q$ has a valid translation with respect to $\varphi$ and $k$ and function environment $\Lambda$, if for all $\sigma, \Delta, \sigma_s$, if $(\sigma, \Delta) \approx_{\varphi} \sigma_s$, then $\sigma_s \models \text{comp}^{\varphi,k}_{\text{spec}}(P) \Rightarrow \text{comp}^{\varphi,k}_{\text{spec}}(Q)$.

Using the manually translated implications, we can automatically translate the syntactic Hoare rules that make up the proof tree. However, while this would let us to construct *a* proof in the target language about the compiled program, it would not necessarily construct a proof of *the same* property, as the meaning of assertions could be destroyed by, for instance, compiling all assertions to $\bot$. To prevent this, we define the following notion of *soundness with regards to a specification.*

*Definition 5.2.* A specification compilation function $\text{comp}^{\varphi,k}_{\text{spec}}$ is sound with respect to $P$ if for all $\sigma, \Delta, \sigma_s$ such that $(\sigma, \Delta) \approx_{\varphi} \sigma_s$, we have $\sigma, \Delta \models P \Leftrightarrow \sigma_s \models \text{comp}^{\varphi,k}_{\text{spec}}(P)$.

To proof compile a program $c$ with specification $\{P\}c\{Q\}$, the user must prove the specification compiler is sound with regards to $P$ and $Q$.[3] This way, we guarantee that while program behavior can change, the specification remains the same property. This approach allows for optimizing or incorrect compilation, so long as it preserves Definitions 5.1 and 5.2.

[3]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.

Note that 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. In practice, we have found these proofs to be quite simple and repetitive, and we have built some proof automation to help dispatch these goals. We have yet to build proof automation to generate a given proof's implication database as a verification condition but we suspect this could be done via a weakest precondition calculation.

### 5.3 Well-formedness Conditions

The second, and final, set of user proof obligations are specific to our choice of languages and logics. Specifically, while the syntax of Imp prevents type errors, it allows for functions to be called on any number of arguments, and for any parameter index to be called within the program body. Our implementation of a proof compiler requires as a user proof burden that all components of the source proof be *well-formed*, in that neither of these cases occur. Additionally, any compiled functions should preserve the stack, so as to meet the preservesStack condition of the Stack logic. We have largely automated these proof burdens with custom tactics in our case studies.

## 6 CASE STUDIES

This section describes two sets of case studies highlighting the tradeoffs of the PotPie approach:

(1) **Partial Correctness with Incorrect Compilation** (Section 6.1): We prove meaningful partial correctness properties of arithmetic approximation functions that are slightly incorrectly compiled. This set of case studies highlights two benefits of PotPie:
   (a) **Specification-Preserving Compilation**: We invoke PotPie with a slightly buggy program compiler to produce proofs that meaningfully preserve the correctness specifications down to the target level. Importantly, we obtain these meaningful target-level correctness proofs of our specification even though the program compiler *does not* preserve the full semantic behavior of the arithmetic approximation functions.
   (b) **Compositional Proof Compilation.** We use PotPie to separately compile the correctness proofs of helper functions common to both approximation functions. Composition of those helper proofs within the target-level proof of the arithmetic function comes essentially "for free," modulo termination conditions.

(2) **PotPie Four Ways** (Section 6.2): We instantiate PotPie with four different 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**. For a simple set of source-level programs and proofs, we briefly explore the tradeoffs of which proofs we can compile for each of these instantiations, and, in each case, where the user proof burdens manifest.

All case studies are available in our Coq formalization.

### 6.1 Partial Correctness with Incorrect Compilation

We have written and proven correct two mathematics approximation programs in Imp. Both approximation programs use common helper functions, which we also prove correct (Section 6.1.1). We then build on and compose the helper proofs to prove our approximation programs correct up to specification even in the face of incorrect compilation (Sections 6.1.2 and 6.1.3). Our incorrect compiler is described in Figure 11. This compiler is mostly correct, except that it miscompiles $<$ into $\leq$. We instantiate PotPie to this program compiler and use it throughout this subsection.

[4]While we don't have a less than operator in the Imp language, in our implementation, we have a macro for it that desugars into $a_1 \leq a_2 \wedge \neg(a_1 \leq a_2 \wedge a_2 \leq a_1)$. For simplicity, we will simply use $<$ in this paper.

$$\begin{aligned}
\mathrm{comp}'^{\varphi}_{\mathrm{badb}}(a_1 < a_2) &\triangleq \mathrm{comp}^{\varphi}_{\mathrm{a}}(a_1) \leq \mathrm{comp}^{\varphi}_{\mathrm{a}}(a_2) \\
\mathrm{comp}'^{\varphi}_{\mathrm{badb}}(b) &\triangleq \mathrm{comp}^{\varphi}_{\mathrm{b}}(b) \\
\mathrm{comp}'^{\varphi,k}_{\mathrm{badc}}(\mathrm{skip}) &\triangleq \mathrm{skip} \\
\mathrm{comp}'^{\varphi,k}_{\mathrm{badc}}(x := a) &\triangleq \#(\varphi(x)) := \mathrm{comp}^{\varphi}_{\mathrm{a}}(a) \\
\mathrm{comp}'^{\varphi,k}_{\mathrm{badc}}(c_1; c_2) &\triangleq \mathrm{comp}'^{\varphi}_{\mathrm{badc}}(c_1); \mathrm{comp}'^{\varphi,k}_{\mathrm{badc}}(c_2)
\end{aligned}$$

$$\begin{aligned}
\mathrm{comp}'^{\varphi,k}_{\mathrm{badc}}\begin{pmatrix}\text{if } b\\ \text{then } c_1\\ \text{else } c_2\end{pmatrix} &\triangleq \begin{matrix}\text{if } \mathrm{comp}'^{\varphi}_{\mathrm{badb}}(b)\\ \text{then } \mathrm{comp}'^{\varphi,k}_{\mathrm{badc}}(c_1)\\ \text{else } \mathrm{comp}'^{\varphi,k}_{\mathrm{badc}}(c_2)\end{matrix} \\
\mathrm{comp}'^{\varphi,k}_{\mathrm{badc}}\begin{pmatrix}\text{while } b\\ \text{do } c\end{pmatrix} &\triangleq \begin{matrix}\text{while } \mathrm{comp}'^{\varphi}_{\mathrm{badb}}(b)\\ \text{do } \mathrm{comp}'^{\varphi,k}_{\mathrm{badc}}(c)\end{matrix}
\end{aligned}$$

Fig. 11. An incorrect compiler. $\mathrm{comp}'_{\mathrm{badb}}$ is a buggy boolean expression compiler that turns our less-than macro into a less-than-or-equal-to expression[4]. $\mathrm{comp}'_{\mathrm{badc}}$ is a code compiler, that is implemented using the buggy boolean expression compiler, and a correct arithmetic expression compiler $\mathrm{comp}_{\mathrm{a}}$.

*6.1.1 Helper Functions.* We describe our correctness proofs about two helper functions: multiplication and exponentiation. For clarity, we omit environments in the lemmas we state here, since the main constraint was that function environments should contain all of the functions used in our larger programs. More details can be found in our formalization.

*Multiplication.* The first helper function is a multiplication function, which behaves as expected (code in green is actually wrapped Coq terms' arithmetic logic, whereas code in black is an expression in our language substituted into a Coq term as per the semantics of our logic in Figure 5):

```
{ ⊤ }
x := param 0; y := 0;
while (1 ≤ x) do
  y := y + param 1;
  x := x - 1;
{ y = (param 0) · (param 1) }
```

The proof of this Hoare triple at the source level is straightforward. Because the function is simple and the body of the function does not encounter the incorrect behavior of the compiler, user proof obligations from Section 5 are easy to satisfy, and it is simple to compile this proof to the target level.

While this proof alone is of a simple function, by using it in downstream functions, we can demonstrate the benefits of compositionality in PotPie. By combining this triple with a termination proof, we are able to generate a helper lemma that reflects applications of the multiplication function in the Imp langauge into Coq's built-in natural number multiplication:

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

This lemma lets us reason more directly using Coq's available numeric reasoning automation. We use this lemma in our downstream proofs about downstream programs. Using this workflow with PotPie, we are able to reuse the source Hoare proof of this triple to get the target-level version of this lemma *almost* for free—though not quite for free, as we have to reprove termination at the target level, something we hope to address in future work.

*Exponentiation.* Exponentiation is similarly straightforward:

```
{ ⊤ }
  x := param 0;
  y := 1;
  while (1 ≤ x) do
    y := mult(y, param 1);
```

```
      x := x - 1;
  { y = (param 1)^(param 0) }
```

Our exponentiation helper function refers to our multiplication helper function already. Here, the compositionality benefits of our approach begin to shine. Using the `mult_aexp_wrapper` lemma, we easily reduced many user proof obligations for the specification translation conditions from Section 5.2 to simple arithmetic reasoning in Coq. We can then dispatch these easily using Coq's tactics, and prove an analogous reflection lemma for exponentiation, which we also use downstream.

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

*6.1.2 Geometric Series.* One example use case for partial correctness specifications is floating point estimation of mathematical functions, like $\sin(x)$ and $e^x$, by way of computing a series like the Taylor series. One property of a Taylor series, for example, is that there is an upper bound for the error of the $n^{\text{th}}$ Taylor polynomial, which makes it convenient to estimate the values of functions within a specific bound. Since floating point numbers are unable to represent all of the real numbers, we can only approximate many of these functions within some error bound.

As a simple version of this use case, we consider a program for calculating the geometric series $\sum_{i=1}^{\infty} \frac{1}{x^i}$ within an error bound of $\epsilon = \frac{\delta_n}{\delta_d}$. We require $x \geq 2$ so that the series converges, which simplifies some of our assertions for this example. Of course, this is a toy example that would be easier to compute in its closed form—the series $\sum_{i=0}^{\infty} a \cdot r^i$ is known to converge to $\frac{a}{1-r}$ for $|r| < 1$—but suffices as a simple example of using PotPie compositionally with an interesting partial specification. Section 6.1.3 covers a more practical example.

The program we use to compute this series is as follows. For brevity, we omit assertions outside of the pre- and postcondition, loop invariant, and loop postcondition, and we show wrapped Coq prop and arithmetic terms in green, as in $\delta_n \cdot (x-1)$. Terms and operations in black are Imp expressions.

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

The loop postcondition must imply the program postcondition. Note that, in compiling this program, we encounter the bug in our program compiler, which miscompiles the < in the while loop conditional. However, we are still able to compile this program and its proof to Stack because (1) the pre- and postconditions' meaning is preserved by compilation, and (2) the implication database is still valid, i.e., every compiled Imp implication is still an implication in Stack.

To see (1), we will need to look at the underlying representation of our assertions. As given in Figure 5, our precondition and postcondition actually have the following form:

$$\texttt{(fun x' rn' rd' i' =>}\ 2 \le \texttt{x'} \land \texttt{x'} = x \land \delta_n \neq 0 \land \delta_d \neq 0 \land \texttt{rn'} = 1 \land \texttt{rd'} = x \land \texttt{i'} = 2)\ x\ 1\ x\ 2$$
$$\texttt{(fun rn' rd' =>}\ \delta_d \cdot \texttt{rd'} \le \delta_n \cdot (x-1) \cdot \texttt{rd'} + \delta_d \cdot (x-1) \cdot \texttt{rn')}\ \texttt{rn rd}$$

Everything after the anonymous function is actually an expression in the Imp language. These are the only parts of the assertions that are compiled by the specification compiler. For instance, $x$ is a constant arithmetic expression in Imp, which wraps Coq's `nat` type. The arithmetic compiler function, $\text{comp}_\text{a}$, from Fig. 8 simply compiles these to natural number constants in the Stack language. For the variables `rn` and `rd` however, $\text{comp}_\text{a}^{\varphi,k}(\texttt{rn}) = \#\varphi(\texttt{rn})$. After compiling, we get the postcondition $\{\ \delta_d \cdot \#5 \le \delta_n \cdot (x-1) \cdot \#5 + \delta_d \cdot (x-1) \cdot \#2\ \}$, or symbolically: $\frac{1}{x-1} - \frac{\#2}{\#5} \le \frac{\delta_n}{\delta_d}$.

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

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

After negation, this is equivalent to the below inequality, which represents $\frac{1}{x-1} - \frac{\#2}{\#5} < \frac{\delta_n}{\delta_d}$, which still implies the compiled postcondition. This proof obligation is easily dispatched by Coq's linear integer arithmetic reasoning.

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

*6.1.3 Square Root.* The second approximation program we consider also interacts with the same miscompilation, and still meaningfully preserves the source specification. We consider a square root approximation program that, given numbers $a, b, \epsilon_n, \epsilon_d$, calculates some $x, y$ such that $|\frac{x^2}{y^2} - \frac{a}{b}| \le \frac{\epsilon_n}{\epsilon_d}$. We can project the postcondition entirely into Coq terms, multiplying through both sides by the denominator so we can express it in our language. After writing the program, we come up with the following loop condition ($\cdot$ is syntactic sugar for `mult`, and $<$ is actually the Imp less-than macro):

$$\begin{aligned} \textit{loop\_condition} \triangleq\ & (y \cdot y \cdot b \cdot \epsilon_n < y \cdot y \cdot a \cdot \epsilon_d - x \cdot x \cdot a \cdot \epsilon_d) \\ & \lor\ (y \cdot y \cdot b \cdot \epsilon_n < x \cdot x \cdot b \cdot \epsilon_d - y \cdot y \cdot a \cdot \epsilon_d) \end{aligned}$$

This loop condition represents $\frac{\epsilon_n}{\epsilon_d} < \left|\frac{x^2}{y^2} - \frac{a}{b}\right|$ multiplied through as with the postcondition. Our square root program is given by the following.

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

Most of the rules of consequence are straightforward. The only nontrivial implication involved is the final rule of consequence for the postcondition. The loop's postcondition is $\neg\left(\frac{\epsilon_n}{\epsilon_d} < \left|\frac{x^2}{y^2} - \frac{a}{b}\right|\right) \equiv \left|\frac{x^2}{y^2} - \frac{a}{b}\right| \le \frac{\epsilon_n}{\epsilon_d}$, which directly gets us the program postcondition after reflecting into Coq.

During compilation, note that the loop condition is miscompiled. Instead of correctly compiling $<$, the broken compiler changes it into $\leq$. In the parlance of our target language, we get the following, where again, `mult` is represented by $\cdot$. Note this also represents a program in one of our languages and not a Coq arithmetic expression, and thus is not green.

$$\begin{aligned} \mathit{target\_loop\_condition} \triangleq\ & \#1 \cdot \#1 \cdot b \cdot \epsilon_n \leq \#1 \cdot \#1 \cdot a \cdot \epsilon_d - \#4 \cdot \#4 \cdot b \cdot \epsilon_d \\ \vee\ & \#1 \cdot \#1 \cdot b \cdot \epsilon_n \leq \#4 \cdot \#4 \cdot b \cdot \epsilon_d - \#1 \cdot \#1 \cdot a \cdot \epsilon_d \end{aligned}$$

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

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

While the loop condition is indeed miscompiled and the specification of the program is the negation of the loop condition, the final rule of consequence reflecting the negation of the loop condition into Coq means that even though the loop condition is miscompiled, the final postcondition is *not*. Even though the unsound behavior of the compiler changes the semantics of the loop invariant, it is not enough to break the implication between the unwrapped loop condition and the Coq-wrapped loop condition. Further, because of the way that the postcondition projects into Coq immediately after taking the arguments, the final implication is almost completely provable via applications of wrappers, inversion over the hypotheses, and simple linear integer arithmetic.

### 6.2 PotPie Four Ways

PotPie makes it easy to swap out control-flow-preserving program compilers and still reuse the same infrastructure. We instantiate PotPie with four different variants of a program compiler:

(1) An **incomplete program compiler** that is missing entire cases of the source language grammar. Despite this compiler being incomplete, we can still satisfy the user proof obligations needed to invoke the proof compiler to get guarantees at the target level, so long as the programs we consider do not include those cases.
(2) An **incorrect program compiler** that contains a mistake and an unsafe optimization. We can sometimes invoke its associated proof compiler to get guarantees at the target level even when the program compiles incorrectly, so long as we can prove the user proof obligation showing that the (possibly incomplete) specification is preserved.
(3) Fixing the bugs gets an **unverified correct program compiler** that always preserves program and specification behavior. We can now *always* satisfy the user proof obligations needed to invoke its associated proof compiler for any source program and proof.

Table 1. For each compiler variant, the table lists which guarantees about three different source programs we can preserve down to the target level using PotPie, and where the user proof burden is centered.

| | 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 |

(4) Proving the compiler correct gets a **verified correct program compiler**. User proof obligations can now be derived from the compiler correctness proof.

Table 1 outlines these case studies and tradeoffs for a simple set of programs we consider. For those programs, we have three primary takeaways. First, PotPie can be used to get target-level proofs about certain programs even when the program compiler is still under development. Second, compilation of proofs about partial specifications in the face of incorrect compilation is sometimes possible and sometimes not. The difference is not simply whether the compilation bug manifests in the compiled program, but rather whether it manifests *in a way that is relevant to meaningful specification preservation*. Finally, as expected, there is no difference in what proofs we can compile between the unverified and verified correct compiler. Rather, proving the compiler correct helps automatically dispatch more of the user proof obligations, moving the user proof burden more upfront. For more details, please see our Coq implementation.

## 7 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. When a proof is produced by the proof compiler, the proof will (1) be correct, and (2) have the same meaning as the source proof. The PotPie proof compiler in Coq satisfies the following theorem *by construction*, meaning it is simultaneously defined and proven correct (see Section 8 for implementation details); we also provide an informal proof sketch.

Theorem 7.1. *For code and specification compilers that satisfy the equations in Figure 10 on a program c, a source-level Hoare proof* $H : \{P\}\ c\ \{Q\}$ *and its implication database I, and the user proof burdens described in Section 5, we can construct a proof compiler such that* $\text{comp}_{\text{pf}}^{\varphi,k}(H) : \left\{\text{comp}_{\text{spec}}^{\varphi,k}(P)\right\}\ \text{comp}_{\text{code}}^{\varphi,k}(c)\ \left\{\text{comp}_{\text{spec}}^{\varphi,k}(Q)\right\}$ *is a valid Hoare-deduction proof at the target level.*

Note that the specification compiler being sound with regards to the preconditions and postconditions of the program is a user proof obligation for invoking the proof compiler. This guarantees that the compiled proof "proves the same thing" as the source proof.

Proof. 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 10:

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

The If and While rules are similarly simple applications of the second, fourth, and fifth equations from Figure 10, and Seq is a straightforward application of the induction hypothesis. In the last

case, for the left rule of consequence, we want to show

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

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

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

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

## 8 IMPLEMENTATION

We implement PotPie in Coq and prove that it is sound (Section 8.1). We keep the *trusted computing base* (TCB) small (Section 8.2). 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 8.3)—something that is not needed for basic use, but that would open up new and exciting use cases for PotPie. Additional implementation details are explained in our artifact.

### 8.1 Formal Proof

We formalize PotPie in Coq. Our formalization includes formal proofs of the soundness guarantees from Section 7, along with the case studies from Section 6. Our proof development resulted in a seemingly large codebase of about 48k lines of code (LOC), with about 3k LOC for the Imp language and logic, 3.6k for the Stack language and logic, and around 8k for the proof compiler. This may be surprising, especially compared to the size of Xavier Leroy's referenced course materials, but there are several key differences. First, our languages include functions, making our semantics mutually inductive and therefore inherently more difficult to reason about than Leroy's course's semantics. However, functions also give us the opportunity to reason about the composition of programs and their proofs as we saw in Section 6.1, so we still stand by this design choice. Second, our target language is far less well-behaved than either of the languages in Leroy's course, which required additional wrangling to reason about. Third, the case studies—of which we have seven total—comprise nearly a third of the codebase, ~15k LOC, and include multiple instantiations of the code, specification, and proof compilers, as well as machinery to reason about our case study programs. More details of our proof development—including how we calculated these numbers—can be found in the associated artifact.

In the Coq formalization, our soundness theorem (see Section 7 for an informal proof) holds *by construction*. That is, the PotPie proof compiler is a function in Coq whose type guarantees its correctness (for clarity, the type below is simplified and the well-formedness conditions from Section 5 are omitted):

```
1 hl_compile : ∀ (P Q : log) (i : imp) (fenv : fun_env) (HL : hl P i Q fenv)
2   (facts : fact_environment) (facts' : stk_fact_environment),
3   ∀ (map : list ident) (args : nat) (P' Q' : AbsState) (i' : stk) (fenv' : fun_env_stk),
4     comp_spec map args P = P' → comp_spec map args Q = Q' →
5     i' = comp_code map args i →
6     imp_trans_valid_def facts facts' fenv fenv' map args →
7     hl_stk P' i' Q' fenv'.
```

Informally, 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 proof that shows $\{P'\}\ i'\ \{Q'\}$ in function environment `fenv'`, where $P'$, $i'$, $Q'$, and `fenv'` are correctly related to their Imp counterparts.

Experienced Coq proof engineers may find our decision to prove correctness by construction at odds with the norm of separating computation from proof. We made this decision because translating Hoare proofs necessitates translating implications for the rule of consequence, so it was initially easier to construct a function between the Hoare proofs' dependent types than to separate the proof information from a Hoare data structure. This design also lets us take advantage of Coq's trusted type-checking kernel to both prohibit invalid inputs and produce free proofs at the target level that can be checked by type checking alone. This decision also comes at a cost, particularly with respect to getting source-independent compiled proofs at the target level (see Section 8.3). We believe this was the right decision for our initial system but may revisit it for future versions.

### 8.2 Trusted Computing Base (TCB)

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 confidence that the semantics are implemented correctly 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. We assume UIP for `AbsEnvs` (abstract environments, the Coq implementation of *SM* in Equation 1 in Section 3.2) and function environments.

### 8.3 Challenge: Source-Independent Proof Objects

One potential benefit of compiling proof objects across program logics is that the compiled proofs, once reduced, could be free of references to the source language and logic. This is true of our on-paper proof compiler, but it is not yet true of the PotPie implementation. This is not an issue of correctness since Coq's trusted kernel typechecks the formal proof in Section 8.1, which means that the proof compiler must return a valid target proof.

We would however like this to be true of our implementation, since there are many reasons to want source-independent proof objects at the target level: These proof objects could be robust to changes in the source language or logic. Once generated, they could exist forever separately from the source. They would be independently checkable by people who do not trust the source verification effort or lack access to the source. They could be repaired directly in response to changes in target-level programs, in the style of Gopinathan et al. [2023]. They could even be potentially linked with other proofs compiled from other source languages and logics.

The barriers to source-independent proof objects in the PotPie implementation are nontrivial but surmountable. For one, the correct-by-construction design means that the compiled proofs carry not only the proof structure and rules, but also their proof of correctness, so source code well-formedness principles may show up in compiled proofs. In addition, Coq's proofs are *opaque* by default—Coq does not usually unfold theorems' proof terms—which may prohibit reduction that would otherwise remove source program and proof references. Decoupling the correctness proof from the computation and setting theorems to transparent may help address this in the future.

Once we have source-independent proof objects, we hope to implement linking of target-level proof objects compiled from different source languages and logics. That way, proof engineers will be able to write proofs about components of systems implemented in different languages, and easily get target-level proofs about the entire system. We already have a promising prototype

workflow for separately compiled proofs about helper functions implemented in the *same* source language and logic by way of reflecting back into Coq. The main barriers to multi-language-and-logic linking beyond those we discussed are generalizing our Stack logic semantics, implementing proof compilers for multiple languages, and reasoning about compiled components' interactions.

## 9 RELATED WORK

**Proof-Transforming Compilation.** Early work compiling proofs positioned itself as an extension of *proof-carrying code* [Necula 1997]. Barthe et al. [2005] stated a theorem relating source and target program logics. Transformation of proof objects followed, going by the name *proof-transforming compilation*. Early work for the former [Müller and Nordio 2007] transformed Hoare-style proofs about Java-like programs to proofs about bytecode implemented in XML. Later work [Nordio et al. 2008] implemented proof-transforming compilation from Eiffel to bytecode, formalizing the specification compiler in Isabelle/HOL, with a hand-written proof of correctness of the proof compiler. Hauser [2009] showed how to embed the compiled bytecode proofs into Isabelle/HOL. Our work is the first we know of to formally verify the correctness of the proof compiler, and to use it to support specification-preserving compilation in the face of incorrect program compilation. Revisiting unformalized work about *certificate translation* [Kunz 2009], which is similar but focuses on compiler optimizations, may prove fruitful for relaxing control-flow restrictions.

**Type-Preserving Compilation.** Our work is related to *type-preserving compilation*: compiling programs in a way that preserves their types. There is work on this defined on a subset of Coq for CPS [Bowman et al. 2017] and ANF [Koronkevitch et al. 2022] translations. As the source and target languages both have dependent types, this can likewise be used to compile proofs while preserving specifications. Our work provides three advantages by compiling program-logic proofs instead: (1) it is naturally suited to proofs about imperative programs, (2) it does not depend on the type system of the source or target language, and (3) it makes it possible to swap in different program compilers. Of course, these advantages come at the cost of user proof obligations.

**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 purposes, like 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, and proves said transformation correct and meaningful in Coq.

**Certified Compilation.** Our work is related to *certified compilation*, or formally proving compilers correct. The CompCert verified C compiler [Leroy 2006, 2009], which was proven correct in Coq, lacks 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. Oeuf [Mullen et al. 2018] and CertiCoq [Anand et al. 2017] are both certified compilers for programs written in Gallina, the language underpinning Coq. Certified compilation is desirable when possible, but real compilers may be unverified, incomplete, or incorrect. Our work explores an underexplored point in this design space: compilation that is *specification-preserving* for a given source program and (possibly partial) specification, even when the compilation itself may not be fully meaning-preserving for the given source program. The original CompCert paper [Leroy 2006] brought up the possibility of such specification-preserving compilation as part of a potential design space that is complementary to, rather than in contradiction with, certified compilation. We agree; it expands the space of (possibly partial) guarantees one can get about compiled programs (especially those produced by unverified and possibly incorrect compilers), as well as the means by which one may get said guarantees.

**Certifying Compilation.** Our work can be thought of as akin to *certifying compilation*: producing compiled code and a proof that its compilation is correct. The COGENT language's certifying compiler proves that, for a given program compiled from COGENT to C, target code correctly implements a high-level semantics embedded in Isabelle/HOL [Amani et al. 2016; Rizkallah et al. 2016]. Some work on correct compilation (like *Rupicola* [Pit-Claudel et al. 2022], which phrases correct compilation to low-level code as proof search) includes both certified and certifying components. Certifying compilation and our work share the benefit that the compiler may be incorrect or incomplete, yet still produce proofs about the compiled program. The main thing that distinguishes our work is that we certify *the specification's translation*. Prior work on certifying compilation that we are aware of targets general properties (like type safety) rather than program-specific specifications. Our work adds to the space of certifying compilation to support preservation of program-specific specifications proven at the source level, with added benefits for compositionality.

**Translation Validation.** 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. In translation validation, the compiler produces a proof script of the correctness of a particular program's compilation, which then needs to be checked [Necula 2000]. Both our work and translation validation do not use full compiler correctness, yet can still get strong guarantees on a per-compilation basis. Our work, however, allows for varying degrees of correctness (i.e., preserving partial specifications about incorrectly compiled programs), rather than checking a single notion of correctness for all programs as is common in translation validation.

**Compositionality.** One motivation for our approach is its potential for compositionality. Our work already shows this potential in a limited context for compositional verification of separately compiled components implemented in the same source language and logic. Similar motivation is behind work in compositional certified compilation, e.g., CompCertX [Gu et al. 2015; Wang et al. 2019] and CompCertO [Koenig and Shao 2021]. DimSum [Sammler et al. 2023] defines an elegant and powerful language-and-logic-agnostic framework for language interoperability. However, in order to get guarantees about said interoperability, it leans heavily on data refinement arguments that show a simulation property stronger than what our frameowrk requires. We hope that in the future, we will make our compositional workflow more systematic, and extend it to fill the gap of compositional multi-language reasoning in a relaxed correctness setting—by linking compiled *proofs* directly in a common target logic. Similar motivations are behind linking types [Patterson and Ahmed 2017], which are extensions to a language's type system that make it possible to reason about correct linking in a multilanguage setting. We expect tradeoffs similar to those between our work and type-preserving compilation to arise in this setting.

**Program Logics.** Frameworks based on embedded program logics (e.g., Iris [Jung et al. 2015; Krebbers et al. 2017], VST-Floyd [Cao et al. 2018], Bedrock [Chlipala 2011, 2013], YNot [Nanevski et al. 2008], CHL [Chen et al. 2015], Sepref [Lammich 2019], and CFML [Charguéraud 2011]) have seen success helping proof engineers write proofs in a proof assistant about code with features that the proof assistant lacks. VST is related to our work, as it creates machine-checked proofs about systems that hold all the way down to machine code. By composition with CompCert, source C programs verified in the VST program logic are guaranteed to preserve their specifications even after compilation to assembly code [Beringer and Appel 2021]. Our primary contribution to this space is in creating a similar toolchain that allows the program compiler to be unverifed or even incorrect without sacrificing target-level guarantees. Of course, relative to practical frameworks like Iris and VST, our work is much less mature. We hope to extend our work to more practical logics and lower-level target languages in the future, so that users of toolchains like VST can get guarantees about compiled programs even in the face of incorrect compilation.

## 10 CONCLUSION

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 alleviating the control flow restriction, ensuring target proof objects are source-independent, supporting different source languages and logics, supporting additional linking of target-level proofs, implementing optimizing compilers, and bringing the benefits of proof compilation to more practical frameworks.

## 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

AbsInt. 2006-2023. Structure of CompCert. https://www.absint.com/compcert/structure.htm

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. 2005. 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

Sandrine Blazy and Xavier Leroy. 2009. Mechanized Semantics for the Clight Subset of the C Language. *Journal of Automated Reasoning* 43, 3 (Oct. 2009), 263–288. https://doi.org/10.1007/s10817-009-9148-3

William J. Bowman, Youyou Cong, Nick Rioux, and Amal Ahmed. 2017. Type-Preserving CPS Translation of  and  Types is Not Not Possible. *Proc. ACM Program. Lang.* 2, POPL, Article 22 (dec 2017), 33 pages. https://doi.org/10.1145/3158110

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. 2013. 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

GCC. 2023. GCC Bugzilla. https://gcc.gnu.org/bugzilla/buglist.cgi?chfield=%5BBug%20creation%5D&chfieldfrom=24h

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

Ronghui Gu, Jérémie Koenig, Tahina Ramananandro, Zhong Shao, Xiongnan (Newman) Wu, Shu-Chun Weng, Haozhong Zhang, and Yu Guo. 2015. Deep Specifications and Certified Abstraction Layers. *SIGPLAN Not.* 50, 1 (jan 2015), 595–608. https://doi.org/10.1145/2775051.2676975

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

Jérémie Koenig and Zhong Shao. 2021. CompCertO: Compiling Certified Open C Components. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (Virtual, Canada) *(PLDI 2021)*. Association for Computing Machinery, New York, NY, USA, 1095–1109. https://doi.org/10.1145/3453483.3454097

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

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

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.

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

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.

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

Michael Sammler, Simon Spies, Youngju Song, Emanuele D'Osualdo, Robbert Krebbers, Deepak Garg, and Derek Dreyer. 2023. DimSum: A Decentralized Approach to Multi-Language Semantics and Verification. *Proc. ACM Program. Lang.* 7, POPL, Article 27 (jan 2023), 31 pages. https://doi.org/10.1145/3571220

Yuting Wang, Pierre Wilke, and Zhong Shao. 2019. An Abstract Stack Based Approach to Verified Compositional Compilation to Machine Code. *Proc. ACM Program. Lang.* 3, POPL, Article 62 (jan 2019), 30 pages. https://doi.org/10.1145/3290375

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

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

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

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

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

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

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

Fig. 12. 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.

$$\frac{}{\{P\}\ \mathrm{skip}\ \{P\}}\ \textsc{ImpSkip} \qquad \frac{}{\{P[x \to a]\}\ x := a\ \{P\}}\ \textsc{ImpAssign}$$

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

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

Fig. 13. The full set of Hoare rules for the Imp language.

## A EXTRA FIGURES