# Adapting Verified Compilation for Target-Language Errors

A THESIS PRESENTED BY

PRATAP SINGH

TO

THE DEPARTMENT OF COMPUTER SCIENCE

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
BACHELOR OF ARTS WITH HONORS
IN COMPUTER SCIENCE

ADVISOR: PROFESSOR STEPHEN CHONG
HARVARD UNIVERSITY
CAMBRIDGE, MASSACHUSETTS
NOVEMBER 19, 2021

Thesis advisor: Professor Stephen Chong Pratap Singh

# Adapting Verified Compilation for Target-Language Errors


## Abstract

Verified compilers have the potential to greatly improve users' trust in their code by providing machine-checked proofs of compiler correctness. In recent years they have become increasingly sophisticated and practical, compiling programs from convenient high-level programming languages to realistic target architectures and platforms. Ideally, compiler correctness requires every externally visible behavior of the compiled code to also be a valid behavior of the source it was compiled from. However, modern compilers often translate between languages with very different semantics; in particular, the target language may have error conditions, such as memory exhaustion, integer overflow, and process interruption, that are not expressed or modeled at the source level.

In this work, we explore and evaluate the design space faced by verified compiler writers when adapting their systems for target-language errors. We provide template compiler correctness theorems that account for target-language errors in different ways. We consider a variety of target-language errors that appear in real-world target platforms, examining ways to modify each component of the verified compiler to adapt for them. Among these errors, we include failure modes of external components of the system that affect the running compiled code, such as process interruption and I/O device failure. We describe adaptations to a standard compiler correctness proof technique that allow verified compiler writers to prove one of our proposed correctness theorems when the target language contains limited nondeterminism introduced by target-language errors. Finally, we discuss ways to weaken a traditional compiler correctness statement to obtain guarantees that hold for all executions, including those that encounter target-language errors. Our systematization helps verified compiler writers decide how to appropriately account for target-language errors, allowing them to expose correctness guarantees that more accurately reflect the real-world behavior of compiled code.

# Contents

# Acknowledgments

To my advisors, Professor Stephen Chong and Dr. Shrutarshi Basu: thank you for your deeply insightful guidance and constant support. Your patience and capacity for getting me unstuck time and again were fundamental to my being able to complete this project. Steve, thank you for teaching the classes that sparked my interest in programming languages and compilers, and for shaping my inchoate ideas into a tractable and meaningful project. Basu, thank you for your perceptive advice and careful feedback on all aspects of this work, from its high-level direction to the vagaries of Coq.

To Professor Nada Amin: thank you for introducing me to Coq and formal verification, for exposing me to the power and elegance of programming languages ideas and techniques, and for serving as my thesis reader.

To Professor Adam Chlipala: thank you for your helpful guidance on thorny issues I encountered with coinduction in Coq, as well as for teaching the class that showed me how to think about formal verification of programs.

To the members of the Harvard Programming Languages group: thank you for welcoming me into your community and for your insightful questions and feedback on this project.

To Dr. Holly Hedgeland: thank you for setting me on the path that has brought me to this point. I will forever be grateful for your confidence in me.

To Katherine, Nivi, Paige, Emily, and Tamara: thank you for taking the time to read and give me feedback on each draft of this thesis. Your thoughtful comments have made this document so much more readable and coherent.

To Katherine, Nivi, Tamara, Brinkley, Paige, Emily, Dhilan, Esther, Simon, Alex, and Hari: thank you for your constant support and friendship through the joys and struggles of this thesis. Your kindness, humor, and generosity have never failed to uplift me, and it is thanks to you that my time at Harvard will always be among my most cherished memories.

To my family, near and far: thank you for filling my life with joy, comfort, and warmth.

To Vikram: thank you for being my deepest friend, and for always being present for me.

To Mama and Papa: thank you for your immeasurable love and care, for your unwavering support and encouragement, and for helping me become who I am today.

*Can you trust your compiler?*

Xavier Leroy [28]

# 1 Introduction

Verified compilers are compilers with machine-checked proofs of compiler correctness: the property that the code the compiler emits has the same behavior as the source program it was compiled from. They promise to greatly increase users' trust in the executables they produce. As such, they have become increasingly prevalent and sophisticated in recent years, seeing usage in the development of a variety of safety-critical systems. For example, the CompCert verified C compiler was recently integrated into the build pipeline of engine-control-unit software deployed in diesel generators in nuclear power plants [24]. The compiler correct-

ness theorem of CompCert was combined with static analyses at the source and target level to produce a more trustworthy executable output.

The growing use of verified compilers necessitates a careful understanding of exactly what guarantees these tools provide, under what conditions those guarantees hold, and what potentially unexpected behaviors may occur in compiled code without violating those guarantees. Even CompCert, for example, has been found to contain miscompilation bugs that lie outside of the region covered by its correctness theorem [45].

Existing compiler correctness theorems, both in the literature and in production compilers such as CompCert, typically simplify the model of the target language or machine to avoid dealing with corner cases that are deemed too difficult to reason about, too obscure, or otherwise not worth considering. Many such corner cases are associated with *errors* or *failure modes* of a target machine or architecture, which cannot be represented or reasoned about at the level of source programs. For example, consider memory exhaustion: existing verified compilers often assume that target memory allocations cannot fail [43], allowing them to avoid reasoning about memory exhaustion errors. This means that the verified compiler provides no guarantees at all for a target-language execution that does encounter a memory exhaustion error.

In this work, we study adaptations to standard verified compilation systems that can account for target-language errors, allowing us to relax some of these simplifying assumptions. Based on work in the field (elaborated in Chapter 5), we can identify several important considerations for designing a verified compiler and its accompanying correctness theorem when the target has a complex error model. These include:

- **Precision.** The correctness theorem should give strong guarantees about the behavior of compiled programs, ideally with only small relaxations of standard statements of compiler correctness.
- **Simplicity.** The correctness theorem should be straightforward to understand so that users of the compiler can easily reason about the guarantees that are provided for their compiled code, and can straightforwardly compose those guarantees with other guarantees produced by verification at other stages in the code creation pipeline.
- **Realism.** The machine model should be faithful to the actual machine (be it hardware or a language runtime) on which the target code will be run.
- **Provability.** It should be possible to prove that the compiler satisfies the theorem, ideally using standard compiler correctness proof techniques.

In practice, it may be impossible to entirely eliminate all cases of failure mode mismatch—a compiler for identical source and target languages is hardly of much interest. However, a more in-depth understanding of the considerations involved when targeting realistic machine models would help us move closer to the goal of a fully end-to-end verified system. Our work addresses one aspect of this goal: we aim to provide a simple and principled systematization of the design space of compilers and compiler correctness theorems when target languages have complex error models.

## 1.1 Our contributions

Our contributions are as follows:

- We provide template compiler correctness theorems for target languages with complex error models, accounting for errors at the level of execution traces or of whole programs. These theorems balance the four criteria in different ways, and each may be suitable in different situations. This contribution is presented in Section 3.1.
- We describe and compare a variety of adaptations to address different types of target-language errors when designing a verified compiler and its associated correctness theorem. We consider a wide range of specific errors that occur in practical machine models, including floating-point imprecision, integer overflow, memory exhaustion, and process interruption. For each of these, we discuss ways to adapt each of the major components of a verified compilation system in order to account for the error. This contribution is presented in Section 3.2.
- We present a modification to a standard compiler correctness proof technique that accounts for the limited nondeterminism introduced by target-language errors. We demonstrate that our modification does not impose a significant additional proof burden on the verified compiler writer. To validate our proof technique, we implement a verified compiler for a target with process interruption and prove in the Coq proof assistant that our compiler satisfies one of our proposed correctness theorems. This contribution is presented in Section 3.3.

- We consider ways to weaken traditional compiler correctness to obtain guarantees of basic safety properties that hold over all target-language executions, including those that encounter errors. Such weaker guarantees can prevent unreasonable outcomes in cases where standard definitions of correctness cannot hold. This contribution is presented in Chapter 4.

The structure of this thesis is as follows. Chapter 2 gives background on verified compilation and defines the formalisms used in the rest of the thesis. Chapter 3 gives a systematic description of the design space of errors and adaptations, and evaluates which responses are most suited for each error. Chapter 4 describes weaker guarantees that can hold even in the absence of traditional compiler correctness results. Chapter 5 discusses related work, and Chapter 6 concludes and proposes avenues for future work.

# 2

# Background

In this chapter, we provide background on compiler verification and proof assistants, and give definitions and notation used in the rest of the thesis.

## 2.1 Preliminaries

Informally, the purpose of a compiler is to translate a source program into a target program with the same meaning. In order to reason about compiler correctness, then, we must make precise the notion of the "meaning" of a program. To do so, we must specify what proper-

ties of program execution we wish to preserve from source to target, and we must define a mapping from programs to these properties.

We follow past work [27, 43] in defining the behavior of a program as a *trace of externally visible events*. Depending on the language and system under consideration, these events could include calls to external functions, system calls, or even direct interaction with hardware. We choose this model for program behavior since during compilation, we generally do not require internal details of program execution such as data representation or execution time to be preserved. Compiler users can generally only observe their programs' explicit interactions with the outside world, so it is sufficient for the compiler to preserve these interactions appropriately.* Let:

- *Event* be the set of externally visible events that a program can emit;
- *Trace* be the set of all finite traces of events in *Event*;
- $Trace^\infty$ be the set of all (finite or infinite) traces of events in *Event*, s.t. $Trace \subset Trace^\infty$;
- for $m \in Trace, t \in Trace^\infty$, let $m < t$ if and only if $m$ is a finite prefix of $t$;
- for $e \in Event, t \in Trace^\infty$, let $e :: t$ be the trace consisting of $e$ followed by $t$.
- for $m \in Trace, t \in Trace^\infty$, let $m +\!\!+ t$ be the trace consisting of of $m$ followed by $t$.

We must now specify the mapping from programs to their traces. We do so by defining *small-step operational semantics* [37] for our source and target languages, specifying the possible atomic transitions between states of an abstract machine during execution of a program. We specify the semantics of a language $L$ with a relation $\longrightarrow \ : S_L \times (Event \cup \{\varepsilon\}) \times S_L$

*Note that certain security properties such as cryptographic constant-time [6], which (informally) requires that all executions of a program perform the same number of steps, cannot be reasoned about at the level of externally visible events. Such properties are beyond the scope of this thesis.

where $S_L$ is the set of states of the abstract machine of $L$ and $\varepsilon$ denotes a silent event (i.e., we say a transition emits event $\varepsilon$ if it does not have any externally visible effects). We use the notation $\sigma \xrightarrow{e} \sigma'$ to denote $(\sigma, e, \sigma') \in \longrightarrow$, and $\sigma \longrightarrow \sigma'$ to denote $(\sigma, \varepsilon, \sigma') \in \longrightarrow$.

We can further define the relation $\longrightarrow^* \; : S_L \times \mathit{Trace} \times S_L$ as the reflexive transitive closure of $\longrightarrow$, using the following three rules:

$$\frac{}{\sigma \xrightarrow{[]}{}^* \sigma} \qquad \frac{\sigma \longrightarrow \sigma' \quad \sigma' \xrightarrow{t}{}^* \sigma''}{\sigma \xrightarrow{t}{}^* \sigma''} \qquad \frac{\sigma \xrightarrow{e} \sigma' \quad \sigma' \xrightarrow{t}{}^* \sigma''}{\sigma \xrightarrow{e :: t}{}^* \sigma''}$$

To reason about infinite executions, we define the relation $\longrightarrow^\infty \; : S_L \times \mathit{Trace}^\infty$ as the infinite transitive closure of $\longrightarrow$, using the following two rules:

$$\frac{\sigma \longrightarrow \sigma' \quad \sigma' \xrightarrow{t}{}^\infty}{\sigma \xrightarrow{t}{}^\infty} \qquad \frac{\sigma \xrightarrow{e} \sigma' \quad \sigma' \xrightarrow{t}{}^\infty}{\sigma \xrightarrow{e :: t}{}^\infty}$$

Here, the double bar indicates that the rules are to be interpreted coinductively [40]; that is, $\longrightarrow^\infty$ may be constructed as the greatest fixed point of the set defined by this rule.

The distinction between finite and infinite traces is not central to this work, though proving preservation of termination and nontermination is important when verifying a compiler in practice. We therefore define a final relation $\rightsquigarrow \; : L \times \mathit{Trace}^\infty$ as follows:

**Definition 1** ($\cdot \rightsquigarrow \cdot$)**.** For language $L$, $P \in L$, $t \in \mathit{Trace}^\infty$, let $P \rightsquigarrow t$ hold if and only if

$$(\exists \sigma_F,\ \sigma_P \xrightarrow{t}{}^* \sigma_F) \;\vee\; (\sigma_P \xrightarrow{t}{}^\infty)$$

where $\sigma_P$ is the starting state of language $L$ when running program $P$, and $\sigma_F$ is an irreducible state of language $L$ from which no further steps are possible.

Informally, $P \rightsquigarrow t$ means that program $P$ admits trace $t$, i.e., $t$ is a possible behavior of $P$. Note that $P \rightsquigarrow m$, where $m \in \mathit{Trace}$, does not require that $P$ terminates: $P$ could diverge silently after emitting the finite trace $m$.

Finally, we define the rest of the notation we use in this thesis. To help distinguish the various elements of the verified compiler system, we adopt a typographical convention inspired by Abate et al. [1], Patterson and Ahmed [36], and others: blue, sans-serif font denotes source-language elements, **red, bold font** denotes **target-language** elements, and *black, italic font* denotes *common* elements between source and target. We now define notation for the compiler and associated components. Let:

- $[\![\cdot]\!]_C$ denote compilation using compiler $C$;
- $P \rightsquigarrow t$ denote that program $P$ admits trace $t$;
- Source and **Target** be the sets of source- and target-language programs;
- **err** be a target-language error of interest.

## 2.2 The Coq proof assistant

Formal verification of software has been a goal of computer science research for decades, and recent years have seen the development of a growing number of tools and techniques for this problem. *Proof assistants* are a class of such tools: they allow users to construct machine-checked proofs of theorems in a computational representation of some logical system. Unlike

a theorem proved by hand, for which mathematicians must manually examine the proof for correctness before trusting it, theorems proved in a proof assistant can be trusted without reading the proof itself. As long as one trusts the logical kernel of the proof assistant, which is typically a small program implementing a simple logical calculus, one can trust any proof that it accepts as correct [11]. Popular existing proof assistants include Coq [12], Agda [3], Isabelle/HOL [22], and Lean [26].

Our implementation (described in Appendix A) was developed using the Coq proof assistant. Coq consists of three languages: Gallina, a pure, dependently-typed functional programming language with ML-like syntax; $L_{tac}$, a language of *tactics* that tell Coq how to construct low-level proof terms for theorems; and the Vernacular, a language of commands to the Coq kernel. Additionally, Coq supports extraction to OCaml, Haskell, or Scheme [38], allowing users to write programs in Gallina, prove properties of these programs, and extract an executable which satisfies those properties. This pattern requires users to trust both the extraction mechanism and the compiler for the extraction language. Ongoing work [4] aims to verify these components.

Coq has been used for a variety of seminal formal verification projects. These include the CertiKOS verified operating system kernel [19], the FSCQ verified crash-safe file system [10], and several verified implementations of cryptographic protocols [5, 7, 14]. One of the most significant verification projects conducted using Coq is the CompCert verified C compiler, which we discuss in Section 2.3.

## 2.3 Verified compilation

Compilers are among the most complex pieces of software that developers regularly interact with. For example, the GCC compiler contains over 15 million lines of code [16]. Despite extensive testing and the use of software engineering best practices, the size and complexity of compilers render them vulnerable to bugs and programming errors. Some of these bugs may only cause crashes or poor performance when the compiler is run, but the most pernicious compiler bugs are those that cause the compiler to silently emit incorrect code. Yang et al. [45] tested eleven different industrial-strength C compilers, some open-source and some commercial, and found that every single one had bugs that resulted in silent miscompilation. This disconcerting result suggests that new development techniques are required to build compilers that generate correct code.

The idea of giving a formal proof of correctness for a compiler dates back to 1967, when McCarthy and Painter [31] gave a pen-and-paper proof of correctness for a compiler from arithmetic expressions to a simple single-register abstract machine. Five years later, Milner and Weyhrauch [32] mechanized part of this proof in the LCF proof assistant. This was among the earliest significant uses of a proof assistant to verify a property of a program.

More recently, the CompCert project [27, 28] was the first to develop a practical verified compiler. CompCert compiles a large subset of C to several common assembly languages, including PowerPC, ARM, and x86. The compilation passes are implemented as Gallina functions and proved correct using Coq. CompCert's proof of correctness covers about 90% of the compiler, including all of the translation and optimization passes. From a correctness

point of view, Yang et al. [45] found only two miscompilation bugs in CompCert, and both turned out to originate in the unverified components of the code base. CompCert has been used for a handful of safety-critical industrial projects, including the development of fly-by-wire systems and flight control software.[†] It was recently integrated into the build pipeline of engine-control-unit software deployed in diesel generators in nuclear power plants [24], where its correctness theorem was composed with source-level static analyses to move closer to an end-to-end correctness guarantee for the whole system.

CakeML [25, 43] is a verified compiler for a large subset of Standard ML, built and verified using the HOL4 proof assistant. Unlike CompCert, CakeML is bootstrapped: it can compile itself [25]. It accomplishes this using a verified frontend that compiles HOL4 functions into CakeML abstract syntax trees; the compiler (which is essentially a large HOL4 function) can then be passed into this frontend and compiled.

In addition to these two well-known examples, there is significant existing work on both the theory of verified compilation and on developing implementations of verified compilers using various tools. We survey some of this work in Chapter 5.

### 2.3.1 Compiler correctness theorems

This thesis studies adaptations to the four major components of a verified compiler system: the source language, the target language, the compilation function, and the correctness theorem and proof. The first three of these should be familiar from study of any compiler (verified or not), so we elaborate the standard forms of compiler correctness theorems below.

[†]See https://www.absint.com/compcert/ for details.

Compiler correctness theorems are generally based on the notion of semantic preservation, which requires that the compiled code has the same behavior as the source program. There are several possible ways to state a semantic preservation theorem. These can be classified according to the specific relation they provide between the valid behaviors of the source and target programs, as follows [27, 36]:

FORWARD SIMULATION. Forward simulation theorems take the following form:

**Theorem 1** (Forward simulation). $\forall \mathsf{S} \in \mathsf{Source}, \forall t \in \mathit{Trace}^{\infty}, \mathsf{S} \rightsquigarrow t \implies [\![\mathsf{S}]\!]_C \rightsquigarrow t.$

That is, every valid behavior of the source program is also a valid behavior of the compiled target program. Forward simulation theorems are generally easy to prove: typically, they are proven by induction on the derivation of $\mathsf{S} \rightsquigarrow t$ [27]. However, forward simulation theorems allow the target program to have more behaviors than the source semantics allow, making them potentially unsatisfactory for verified compiler users. Furthermore, if the source language includes nondeterminism or undefined behavior, the compiler must emit target code that admits all possible source behaviors.

BACKWARD SIMULATION. Backward simulation theorems take the following form:

**Theorem 2** (Backward simulation). $\forall \mathsf{S} \in \mathsf{Source}, \forall t \in \mathit{Trace}^{\infty}, [\![\mathsf{S}]\!]_C \rightsquigarrow t \implies \mathsf{S} \rightsquigarrow t.$

That is, every valid behavior of the compiled program is also a valid behavior of the source program. Backward simulation theorems provide desirable correctness guarantees, especially when composed with guarantees about the source program: if the behavior of the source program satisfies some property, then the behavior of the compiled code is guaranteed to have

the same property. In the general case, however, proving a backward simulation requires *back-translation*: constructing a source state that corresponds to every target state in the execution of a compiled program, effectively inverting the compilation function. This can make proving backward simulation difficult.

BISIMULATION. Bisimulation theorems take the following form:

**Theorem 3** (Bisimulation)**.** $\forall \mathsf{S} \in \mathsf{Source}, \forall t \in \mathit{Trace}^\infty, \mathsf{S} \rightsquigarrow t \iff [\![\mathsf{S}]\!]_C \rightsquigarrow t.$

That is, bisimulation requires both forward and backward simulation. It is the strongest correctness guarantee, but is typically too strong for practical source and target languages: it requires that both languages feature exactly the same nondeterminism.

Ideally, we would like to prove backward simulation correctness theorems without defining a back-translation. To do so, we appeal to *determinism* in the target language [8]:

**Definition 2** (Determinism)**.** Language $L$ is deterministic if and only if:

$$\forall P \in L, \forall t, t' \in \mathit{Trace}^\infty,\ P \rightsquigarrow t \implies P \rightsquigarrow t' \implies t = t'.$$

**Theorem 4.** *If* **Target** *is deterministic, then Theorem 1 implies Theorem 2.*

*Proof.* Let $\mathsf{S} \in \mathsf{Source}$ and $t \in \mathit{Trace}^\infty$ such that $\mathsf{S} \rightsquigarrow t$. By Definition 1, we have $[\![\mathsf{S}]\!]_C \rightsquigarrow t$. Now, let $t' \in \mathit{Trace}^\infty$ such that $[\![\mathsf{S}]\!]_C \rightsquigarrow t'$. By determinism of **Target**, we have $t' = t$. Therefore $\mathsf{S} \rightsquigarrow t'$, as required. □

Theorem 4 allows us to prove forward simulation for our compiler, as well as determinism of the target language, and obtain backward simulation as desired. This is the approach taken

by CompCert [27]. In practice, most compilation targets can be modeled as deterministic languages. However, some of the errors considered in this work may make the target language nondeterministic; we discuss ways to adapt this standard proof technique to account for such forms of nondeterminism in Section 3.3.

# 3

# Characterizing the design space

In this chapter, we present our systematization of the design space of adaptations to a verified compiler system that account for target-language errors. Section 3.1 discusses the general classes of adaptations, Section 3.2 explores how these adaptations can be instantiated for particular target-language errors, and Section 3.3 discusses how to adapt compiler correctness proof techniques for our modified correctness theorems.

There are four major components in a verified compilation system: the source language and semantics, the target language and semantics, the translation from source to target, and the compiler correctness theorem and proof. Each component can be adapted for target-

language errors: we can adjust the source language to be more compatible with the restrictions of the target, we can use a less realistic target semantics to simplify reasoning about errors, we can modify the compilation strategy to avoid generating the error, or we can prove a more precise correctness theorem that explicitly accounts for the error. Some of these adaptations may make sense for particular target-language errors but be inapplicable or even absurd for others. In general, there is no adaptation that accounts for all target-language errors equally well. Instead, there is a space of design choices to be explored when building a verified compiler for a complex target language. We now explicitly enumerate this design space, exploring how different adaptations account for different target-language errors.

Table 3.1 summarizes the design space for a verified compiler with a target error model that differs from that of the source. We consider seven different types of target-language errors, each having distinct properties requiring different adaptations. For each error, we consider seven adaptations for the verified compiler system. Each cell in the table summarizes the particular design that results from applying the technique in the column to the error in the row. Several of the resulting designs are nonsensical: they would break the semantics of some component of the verified compiler, or would require unrealistic assumptions. We include (in *italics*) these absurd designs for the sake of comparison, and to underscore the fact that there is no single approach that appropriately accounts for all errors. On the other hand, some designs are particularly appropriate or commonly used to account for the error in question; these are shown in **bold**.

| Error | Modify source lang. | Modify target | Modify compilation strategy | Per-execution theorem | Prefix-correct theorem | Per-program theorem | Weaken value relation |
|---|---|---|---|---|---|---|---|
| Memory exhaustion | Allow source memory allocation to fail | *Assume machine has infinite memory* | *Garbage collection (does not allow any guarantees)* | **Any execution that doesn't exhaust memory is correct** | **Any execution is correct until it runs out of memory** | **Static resource-usage analyses such as cost semantics** | N/A |
| Integer overflow | **Source language with fixed-width integers** | *Target machine with mathematical integer registers* | **Compile mathematical integers to bignums** | *Any execution that never causes overflow is correct* | *Any execution is correct until integer overflow occurs* | *Interval analysis* | *Map source integer to its value modulo the max integer* |
| Floating-point imprecision | **Source language with floating-point numbers** | *Target machine with rational-number registers* | **Compile rationals to (numerator, denominator) pairs** | *Any execution in which no float differs by more than $\varepsilon$ from its "true" value is correct* | *Any execution is correct until some float differs by more than $\varepsilon$ from its "true" value* | Statically compute an approximation bound $\varepsilon$ | Map source rational $x$ to $[x-\varepsilon, x+\varepsilon]$ |
| Unrepresentable input | Source `input()` reads only byte strings | N/A | **Compile source and target values to be identical** | **Any execution in which inputs pass target-level validation is correct** | **Any execution is correct until target-level validation fails** | N/A | **Map one source value to many target values** |

| Error | Modify source lang. | Modify target | Modify compilation strategy | Per-execution theorem | Prefix-correct theorem | Per-program theorem | Weaken value relation |
|---|---|---|---|---|---|---|---|
| I/O failure | **Allow I/O calls to fail** | *Assume I/O never fails* | Buffer and try I/O operation again | **Any execution in which all I/O succeeds is correct** | **Any execution is correct until an I/O failure** | N/A | N/A |
| Dynamic linking failure | **Allow dynamic link loading to fail** | *Assume linking never fails* | *Try to download the linked file from a repository* | **Any execution in which all linking succeeds is correct** | **Any execution is correct until the first linking failure** | N/A | N/A |
| Process interruption | *Add constructs to handle interrupts* | *Run on a machine with no OS and no other processes* | *Trap the interrupt and continue* | **Any execution which is not interrupted is correct** | **Any execution is correct until it is interrupted** | N/A | N/A |

**Table 3.1:** The design space of target-language errors and adaptations to address them. Cells in **bold** indicate cases where this approach is particularly appropriate for this error. Cells in *italics* indicate cases where this approach is nonsensical for this error. Due to space constraints, a detailed description of each design is deferred to Section 3.2.

## 3.1 Types of adaptation

We now define the adaptations listed in Table 3.1, giving template theorem statements for the approaches that modify the compiler correctness theorem. Detailed descriptions of how each adaptation is instantiated for particular target-language errors are deferred to Section 3.2. Each adaptation offers different tradeoffs among the considerations of precision, simplicity, realism, and provability (as we describe in Chapter 1); we describe these tradeoffs here as well.

Modify source language. These adaptations add to, remove from, or change parts of the source language and semantics to be more compatible with the target-language constructs that cause the error in question. Such adaptations often result in simple and easily provable correctness theorems, potentially at the cost of convenient, desirable high-level language abstractions.

Modify target. These adaptations change the model of the target language used for verification. Note that we do not consider modifications to the target language itself: those often require changes to hardware or operating systems, and are thus beyond the scope of a compiler. These adaptations typically result in simple and easily provable correctness theorems, but lose some realism.

Modify compilation strategy. These adaptations compile source constructs in ways that avoid triggering the target-language error. These adaptations typically result in simple correctness theorems, but may incur a performance penalty or require a more difficult proof.

Furthermore, it may not be possible to develop a compilation strategy that avoids the error in *all* cases, as would be required for a compiler correctness guarantee.

The remaining four adaptations modify the compiler correctness theorem, maintaining a realistic target model and trading off the other three considerations. Following existing work (as described in Section 2.3.1), we take Theorem 2, the backward simulation correctness theorem, as the "ideal" theorem which we adapt to account for target-language errors.

PER-EXECUTION CORRECTNESS THEOREM. This theorem states that any target-level trace that does not encounter the error is also allowed by the source. Note that it gives no guarantees about any target trace that terminates with the error, including before the error occurs in that trace.

**Theorem 5** (Per-execution correctness theorem)**.** $\forall \mathsf{S} \in \mathsf{Source}, \forall t \in \mathit{Trace}^{\infty}$,

$$[\![\mathsf{S}]\!]_C \rightsquigarrow t \implies \left(\mathsf{S} \rightsquigarrow t \vee \exists m \in \mathit{Trace}, t = m +\!\!+ [\mathbf{err}]\right)$$

This theorem is relatively easy to prove and simple to understand, but it is quite imprecise—it allows compiler $C$ to emit code that has any finite behavior and then triggers the error, without regard to the source program.

PREFIX-CORRECT THEOREM. This theorem refines Theorem 5 by additionally requiring that target traces that end with the error are correct up until the error occurs. That is, target traces that are not also source traces must consist of a prefix of a valid source trace followed by the error.

**Theorem 6** (Prefix-correct theorem)**.** $\forall \mathsf{S} \in \mathsf{Source}, \forall t \in \mathit{Trace}^{\infty},$

$$[\![\mathsf{S}]\!]_C \rightsquigarrow t \implies \left(\mathsf{S} \rightsquigarrow t \vee (\exists m \in \mathit{Trace}, \exists t' \in \mathit{Trace}^{\infty}, t = m +\!\!+ [\mathbf{err}] \wedge \mathsf{S} \rightsquigarrow m +\!\!+ t')\right)$$

In general, Theorem 6 provides an improvement in precision compared to Theorem 5, with minimal loss of simplicity or provability. We use a specialization of this theorem in our implementation of a verified compiler for a nondeterministic target language, as discussed in Section 3.3 and Appendix A.

Per-program correctness theorem. This theorem gives the full semantic preservation guarantee of Theorem 2, but only for programs that can be proven statically to never encounter the error. Define a source-level predicate $\mathsf{check}(\mathsf{S}, \dots)$ that holds for all programs $\mathsf{S}$ such that $[\![\mathsf{S}]\!]_C$ is guaranteed not to encounter **err**.

**Theorem 7** (Per-program correctness theorem)**.** $\forall \mathsf{S} \in \mathsf{Source},$

$$\mathsf{check}(\mathsf{S}, \dots) \implies \left(\forall t \in \mathit{Trace}^{\infty}, [\![\mathsf{S}]\!]_C \rightsquigarrow t \implies \mathsf{S} \rightsquigarrow t\right)$$

The predicate check may take other parameters: these may include machine properties or bounds on certain inputs. Users of compiler $C$ must prove $\mathsf{check}(\mathsf{S}, \dots)$ for their program $\mathsf{S}$ in order to get the correctness guarantee; in some instances this can be done by an automated static analysis. This theorem is simple to understand, but the complexity of check affects how easy it is to prove—we may need to use details of the definition of check in the proof to show that programs that pass check never encounter errors. Moreover, this theorem may be quite

imprecise, since check may conservatively exclude programs that would not have triggered the error.

WEAKENING THE TRACE RELATION. The following theorem, from Abate et al. [1], allows source and target languages to have different sets of observable events. Let $\mathsf{Trace}$ and $\mathsf{Trace}^\infty$ respectively be the sets of finite and possibly infinite traces containing source-level events, and let $\mathbf{Trace}$ and $\mathbf{Trace}^\infty$ respectively be the sets of finite and possibly infinite traces containing target-level events. Let $\approx_{\mathcal{T}} \subset \mathsf{Trace}^\infty \times \mathbf{Trace}^\infty$ be a relation between source-level and target-level traces.

**Theorem 8** (Trace-relating correctness theorem [1])**.** $\forall \mathsf{S} \in \mathsf{Source}, \forall \mathbf{t'} \in \mathbf{Trace}^\infty$,

$$[\![\mathsf{S}]\!]_C \rightsquigarrow \mathbf{t'} \implies \exists \mathsf{t} \in \mathsf{Trace}^\infty,\ \mathsf{t} \approx_{\mathcal{T}} \mathbf{t'} \wedge \mathsf{S} \rightsquigarrow \mathsf{t}$$

Abate et al. [1] show that with appropriate instantiations of $\approx_{\mathcal{T}}$, this theorem can encode a variety of existing compiler correctness theorems, including both Theorems 5 and 6. However, Theorem 7 *cannot* be encoded by choosing a particular $\approx_{\mathcal{T}}$, since Theorem 8 makes guarantees at the level of individual executions rather than whole programs. In this work, we will generally consider choices of $\approx_{\mathcal{T}}$ generated by lifting a relation $\approx_{\mathcal{V}}$ between source and target *values* to the level of traces. For example, if the source language has booleans but the target language has only integers, we could instantiate Theorem 8 with $\approx_{\mathcal{T}}$ generated by mapping $\approx_{\mathcal{V}} \triangleq \{(\mathsf{true}, 1), (\mathsf{false}, 0)\}$ to verify a compiler that translates booleans into integer values 1 and 0. When $\approx_{\mathcal{T}}$ is based on a simple lifting of $\approx_{\mathcal{V}}$, this theorem is fairly simple and precise, but it may be hard to prove if $\approx_{\mathcal{V}}$ is not one-to-one [1].

## 3.2 Types of errors

Having defined the classes of adaptations in the design space, we now explore how each adaptation can be instantiated for particular target-language errors. We define and discuss each of the errors listed in Table 3.1, evaluating how well each adaptation can account for each error. Note that in general, we do not discuss using the per-execution correctness theorem (Theorem 5), since its properties and suitability are essentially identical to those of the stronger prefix-correct theorem (Theorem 6). We also eschew discussion of modifying the target semantics in cases where it simply makes an unrealistic assumption about the underlying target machine.

### 3.2.1 Memory exhaustion

Real-world computers have finite memory, requiring programmers to deal with the possibility of memory exhaustion. Low-level programming languages typically feature a memory-allocation primitive such as `malloc` in C, which can return an error value to indicate failure. However, high-level source languages generally do not provide precise information about the memory costs of operations, and compilers for such languages may use memory in unexpected ways (e.g., by allocating heap memory to store local variables). More importantly, the semantics of high-level languages may assume infinite memory, with no way to express an out-of-memory condition. This presents an issue for verified compilers: out-of-memory errors cause the target program to behave in a way that is not representable in the source semantics.

Perhaps the easiest solution to this problem is to avoid it entirely: **modify the target** to assume infinite resources, such that the target-language semantics never generate the memory-exhaustion error. We can then retain the simple compiler correctness guarantee of Theorem 2. Notably, CompCert uses essentially this approach [30]: its memory model assumes that the allocation primitive cannot fail. Since most memory allocation in C is handled by the source programmer via functions such as `malloc` that can fail, this assumption is only relevant when reasoning about the allocation of stack frames.

However, this adaptation sacrifices realism if the target is a practical machine or architecture that we wish to verify against. It may also not be possible if the target language has finite-sized addresses: for example, on a 32-bit architecture, a program that allocates more than 4GB of memory will be unable to reference some of the addresses it has allocated.* Furthermore, if the target is an intermediate language in a compilation chain, we have merely pushed the problem further down that chain. We therefore do not consider this adaptation suitable for high-level programming languages that obscure details of memory consumption.

Seeking to avoid the issue in a different way, we might try to **modify the compilation strategy** by running a garbage collection routine when memory is exhausted. In practical settings, this is likely to be a sensible design choice. However, it quickly falls short in a verification context: it is always possible to construct a pathological program in which all allocated memory is still live at the time of exhaustion, such that the garbage collector is unable to free any memory. For example, one could imagine a program that allocates an extremely large linked list,

*CompCert avoids this issue by raising a compiler error and refusing to compile programs that request a stack frame with more than 4GB of local variables [27].

such that every node is always live. Thus, modifying the compilation strategy is not a useful adaptation for memory exhaustion errors.

A potentially more practical strategy is to **modify the source language** to explicitly represent memory exhaustion errors. This could be done through error-reporting mechanisms such as exceptions. For example, Java [18] has an `OutOfMemoryError` which can be thrown at various points in the program. Notably, the `OutOfMemoryError` can be thrown during operations such as implicit boxing conversion (between primitive types and their wrapper classes); Java programmers might not be aware that such operations require any memory allocation. This error can in principle be caught, although the Java 16 specification states that "most simple programs do not try to handle errors" such as `OutOfMemoryError`. Many other popular high-level languages, including OCaml, Python, JavaScript, and C#, take a similar approach of reporting memory-exhaustion errors via existing source-language constructs. This approach is practically useful, as evidenced by its widespread use.

However, modifying the source language to deal with memory exhaustion may negate some of the conveniences of a high-level language. We can instead state and prove a modified **prefix-correct theorem** for our compiler, making explicit exactly what guarantees the compiled code obeys. This is the approach taken by many existing verified compilers. For example, CakeML [43] originally dealt with memory exhaustion by proving a correctness theorem equivalent to Theorem 6. Specifically, they show a theorem equivalent to

$$\forall \mathsf{S} \in \mathsf{Source}, \forall t \in \mathit{Trace}^{\infty}, [\![\mathsf{S}]\!]_C \rightsquigarrow t \implies$$
$$\left(\mathsf{S} \rightsquigarrow t \vee (\exists m \in \mathit{Trace}, \exists t' \in \mathit{Trace}^{\infty}, t = m +\!\!+ [\mathbf{OOM}] \wedge \mathsf{S} \rightsquigarrow m +\!\!+ t')\right)$$

where **OOM** is the out-of-memory error.[†] CakeML users enjoyed preservation of safety properties, but not liveness properties [17], since the compiled code may prematurely terminate due to memory exhaustion. This theorem gives a clear guarantee both for executions that encounter memory exhaustion and those that do not, and the modification to the theorem entailed negligible additional proof effort.

Alternatively, it is possible to adapt the **per-program correctness theorem** for memory exhaustion using static analysis. There are a variety of resource-usage analyses for source programs which can be used to rule out memory exhaustion errors at compile time [17, 34]. These analyses compute a sound approximation to the maximum memory usage of the program, and can be incorporated into Theorem 7 as follows:

$$\forall \mathsf{S} \in \mathsf{Source},\ \exists sz,\ \mathsf{safe\_for\_space}(\mathsf{S}, sz) \implies \forall t \in \mathit{Trace}^{\infty},\ [\![\mathsf{S}]\!]_C \rightsquigarrow_{sz} t \implies \mathsf{S} \rightsquigarrow t$$

where the predicate $\mathsf{safe\_for\_space}(\mathsf{S}, sz)$ holds whenever $[\![\mathsf{S}]\!]_C$ will use at most $sz$ bytes of memory, and $\rightsquigarrow_{sz}$ denotes execution with at least $sz$ bytes of memory available. CakeML recently switched to a theorem of this form that preserves liveness properties [17]: users provide a memory bound for their code and prove the bound correct against a cost semantics. This stronger guarantee does not come for free: the premise $\mathsf{safe\_for\_space}(\mathsf{S}, sz)$ adds a new proof burden for CakeML users. Moreover, the cost semantics required for this proof are *not* for the source language but for an intermediate language whose memory model and semantics are likely unfamiliar to users. This makes proofs of memory safety quite difficult and verbose: Gómez-Londoño et al. [17] found that proving memory safety for a seven-line

[†]See Figure 5 and the definition of `extend_with_resource_limit` in Tan et al [43].

implementation of the `yes` utility required over 1000 lines and 8 person-days of proof effort. Other analyses may be simpler or even automated but compute more conservative approximations to the memory usage of the compiled code. Verified compiler designers seeking to use the per-program correctness theorem can choose analyses that make different tradeoffs between precision and provability. Having chosen an analysis, this theorem provides a simple guarantee for end users, and depending on the details of the safe_for_space premise, writing and verifying such a compiler may be relatively straightforward.

Note that it is impossible to **weaken the value relation** to account for memory-exhaustion errors. There is no relation on values in the source and target that can account for memory exhaustion, since that is a property of a whole program execution. As discussed in Section 3.1 and by Abate et al. [1], it is possible to define a trace relation that captures the prefix-correct theorem of the first version of CakeML.

### 3.2.2 Integer overflow

One of the most common sources of bugs in real-world programs is dealing with integer overflow [33]. High-level source languages would ideally expose arbitrary-precision integers as primitives in the language, so as to match the programmer's intuitive understanding of the arithmetic properties of mathematical integers. However, realistic target architectures support only fixed-width machine integers, whose arithmetic properties differ in subtle ways from mathematical integers. The resulting errors can be particularly difficult to debug.

The most common approach to this problem is to **modify the source language** to reflect the limitations of the target language. Many high-level languages like Java, OCaml, Haskell,

and others provide multiple integer types with fixed bit widths, allowing programmers to select the width they need to represent particular numbers in their code. This approach is well-established and suitable for many source languages, but similar to explicit memory management, it may negate some of the conveniences of programming in a high-level language. Indeed, languages such as Python do provide arbitrary-width mathematical integers [39], so we explore other adaptations that may allow us to do so below.

We can adapt the **prefix-correct theorem** to account for integer overflow. To do so, we would compile source-level mathematical integers to target-level machine integers with fixed bit width, then prove the following adaptation of Theorem 6:

$$\forall \mathsf{S} \in \mathsf{Source}, \forall t \in \mathit{Trace}^{\infty}, \ [\![\mathsf{S}]\!]_C \rightsquigarrow t \implies$$
$$\left(\mathsf{S} \rightsquigarrow t \ \vee \ (\exists m \in \mathit{Trace}, \exists t' \in \mathit{Trace}^{\infty}, t = m +\!\!+ [\mathbf{OF}] \wedge \mathsf{S} \rightsquigarrow m +\!\!+ t')\right)$$

where **OF** is the integer-overflow error. This theorem stipulates that every compiled program must either have the same trace as the source, or have the same trace as a finite prefix of the source followed by **OF**. In particular, since **OF** cannot be generated by the source program, every program that encounters integer overflow cannot have any observable behavior after that error. Unfortunately, this theorem is nonsensical and not useful: by making integer overflow effectively a fatal error, it forces programmers to use the mathematical integers of the source language as if they were machine integers.

Adapting the **per-program correctness theorem** does not produce a better result. We would still compile source mathematical integers to machine integers, but additionally we would provide a premise $\mathsf{no\_overflow}(\mathsf{S}, t)$ which states that when given the inputs of trace $t$, source program $\mathsf{S}$ does not encounter the integer overflow error. The correctness theorem

would then be:

$$\forall \mathsf{S} \in \mathsf{Source}, \forall t \in \mathit{Trace}^{\infty}, \mathsf{no\_overflow}(\mathsf{S}, t) \implies [\![\mathsf{S}]\!]_C \rightsquigarrow t \implies \mathsf{S} \rightsquigarrow t$$

With this theorem, source programmers still must use the source mathematical integers as machine integers. Moreover, in order to get any correctness guarantee at all they have to statically prove a property of every possible execution trace of the source program. This shows that good approaches for memory exhaustion may not translate to good approaches for integer overflow, exemplifying the need for a comprehensive enumeration of the design space of adaptations.

Instead, compilers for source languages with mathematical integers can **modify the compilation strategy** to support them directly in the target language. For example, Python represents a mathematical integer as an array of machine integers, [‡] with arithmetic operations defined accordingly. Many other languages provide "bignum" data types or libraries which expose arbitrary-precision integers implemented in this way. There is a performance cost to this design, especially since increasing the size of the bignum array may require a new memory allocation. However, optimizations, such as dynamically switching to bignums when a machine-integer value overflows, can reduce this cost in practice.

When verifying a compiler for these languages, the verifier must prove that the implementation of arithmetic and logical operations on bignum types is faithful to the semantics of mathematical integers as defined in the source language. While this task may be tedious, it does not require any novel or unusual verification techniques and should be feasible. Having

[‡]See https://github.com/python/cpython/blob/main/Include/cpython/longintrepr.h.

proved correctness of the bignum implementation, we can prove a general correctness theorem that does not mention the integer-overflow error at all. However, since bignum arithmetic may require memory allocation, this verified compiler would also require one of the modified theorems for memory exhaustion described in Section 3.2.1.

Finally, note that **weakening the value relation** does not make sense for integer-overflow errors. The weakened value relation mentioned in the "weaken value relation" column of Table 3.1 is $\approx_{\mathcal{V}} \triangleq \{(\mathsf{n}, \mathbf{n} \bmod 2^{\mathsf{k}}) \mid n \in \mathbb{Z}\}$, i.e., mapping mathematical integers in the source language to their value modulo $2^k$, where $k$ is the bit-width of integers in the target. This is clearly nonsensical—it does not preserve integer values larger than $2^k - 1$.

### 3.2.3 Floating-point imprecision

A common source of errors and unexpected behavior in programs is floating-point imprecision [13]. Suppose we have a high-level language that provides arbitrary-precision rational numbers as a primitive, and a target language that supports floating point numbers according to a standard such as IEEE 754 [20]. These designs match the source programmer's intuition for non-integer arithmetic and the properties of real-world computing hardware, respectively. The naïve approach is to simply compile these arbitrary-precision rational numbers to target floating-point numbers. However, floating-point rounding errors and imprecision can accumulate, causing the compiled program's behavior to diverge from that of the source.

Perhaps the most common adaptation is to **modify the source language**, allowing floating point numbers as primitives and not arbitrary precision rationals. This is the approach

taken by, for example, Java, OCaml, and Python.[§] Furthermore, it allows the compiler to take advantage of hardware-level support and optimizations for floating-point arithmetic, improving performance. Again, however, it may negate some of the conveniences of using high-level languages, since programmers must manually avoid generating floating-point imprecision errors.

Another possible adaptation is to **modify the compilation strategy**: compile rational numbers to (`numerator`, `denominator`) integer pairs. Similar to using bignums to represent arbitrary-width integers, this adaptation avoids the mismatch in representation between source and target, allowing us to prove a correctness theorem of the form of Theorem 2. However, this design may result in poor performance compared to target-language support for floating-point operations: simple operations at the source level such as addition and comparison now require multiple target-level additions and multiplications, and memory consumption may be higher (depending on the bit-widths used). In practice, a combination of these two adaptations may be preferable: many languages expose floating-point numbers as the primitive data type and provide library support for rational-number arithmetic using an integer pair or similar representation.

In cases where the loss of precision is acceptable, however, it might be preferable to compile rational numbers to floating-point numbers. Note that some loss of precision is inevitable, since some rationals, such as $\frac{1}{3}$, can not be represented exactly as a floating point number. In this case, we can parameterize Theorems 5 and 6 with a bound $\varepsilon$ for the acceptable loss of precision. The error would be that some floating-point variable contains a value that dif-

[§]Interestingly, as noted in Section 3.2.2, Python takes a different approach for integers, supporting unbounded integers in the source language.

fers by more than $\varepsilon$ from the value of the corresponding source rational variable. However, unlike the other target-language errors we discuss, this error is *unobservable*: a program execution cannot detect when the error exceeds the permitted bound. Thus the **per-execution** and **prefix-correct theorems** provide little value: they guarantee that either the execution is correct (i.e., within acceptable precision bounds) or it is not, and we have no simple mechanism to discover which is the case.

By contrast, using the **per-program correctness theorem** adaptation entails determining statically whether a particular program will encounter a floating-point imprecision error, avoiding the issue of imprecision being undetectable at runtime. Previous work has used a variety of techniques to approximate floating-point imprecision in programs. For example, Titolo et al. [44] use abstract interpretation to soundly approximate accumulated floating-point imprecision at each program point. Their tool, PRECiSA, produces both these approximate bounds and a formal proof certificate that the bounds are sound. We could incorporate such an analysis into an adaptation of Theorem 7 as follows:

$$\forall \mathsf{S} \in \mathsf{Source}, \forall \varepsilon \in \mathbb{Q}, \mathsf{approx}(\mathsf{S}, \varepsilon) \implies$$
$$\forall \mathbf{t'} \in \mathbf{Trace}^{\infty}, [\![\mathsf{S}]\!]_C \rightsquigarrow \mathbf{t'} \implies \exists \mathsf{t} \in \mathsf{Trace}^{\infty}, \mathsf{S} \rightsquigarrow \mathsf{t} \wedge \mathsf{t} \approx_{\mathcal{T}} \mathbf{t'}$$

where $\mathsf{approx}(\mathsf{S}, \varepsilon)$ holds if and only if $\varepsilon$ is a sound approximation to the imprecision that can accumulate in $[\![\mathsf{S}]\!]_C$, and $\mathsf{t} \approx_{\mathcal{T}} \mathbf{t'}$ holds if and only if $\mathsf{t}$ and $\mathbf{t'}$ are the same up to $\varepsilon$ floating-point imprecision between corresponding values. This design also contains elements of the **weakening the value relation** adaptation, but note that simply weakening the value relation

without a statically verified bound is impossible: for any $\varepsilon$, it should be possible to write a program that eventually accumulates at least $\varepsilon$ floating-point imprecision.

Additionally, note that adapting Theorem 7 may require that control flow and externally visible events do not depend on comparison of rationals. For instance, consider the program `if x > y then output("yes") else output("no")` (where `x` and `y` are rational variables). For values of `x` and `y` that are close together, acceptable floating-point imprecision might change which branch is taken, producing irreconcilable traces in the source and target semantics. This is another drawback to the per-program correctness theorem adaptation relative to the modified compilation strategy discussed earlier, which entails no such restriction.

### 3.2.4 Unrepresentable input

Programmers expect high-level languages to provide ergonomic abstractions for data types and data structures. Compilers for these languages must compile these abstractions to lower-level representations. When the source program takes a value of one of these types as input, the compiled code receives bytes that should conform to the chosen representation; a target-language error can occur if the input bytes do not match that representation. This issue can arise when reading from a command line or a file, or when using a foreign function interface to receive objects from another language.

As an example, consider a source language with boolean values true and false, compiled to a target language that has only 32-bit integer values. If we compile true to 1 and false to 0, the target value 10 would be unrepresentable as a boolean variable. The adaptation of **modifying the source language** would mean defining the source-level input() primitive to read

and return only 32-bit integers, requiring the programmer to choose how to map integers to boolean values. This is suitable for this simple example, but for more complex data types this would require the programmer to know and implement details of the memory representations used by the compiler. Alternatively, we could choose a **modified compilation strategy** in which source and target values are structurally identical—in this case, allocating only one bit for each source boolean variable. This avoids the unrepresentable input issue entirely, but may not be feasible for complex data types.

We can also use the **per-execution** and **prefix-correct theorems** to account for cases of unrepresentable input. For every source $\mathsf{input}()$ call, the compiler could insert code to validate the received input against the desired representation and raise a target-language error in cases of invalid input. We could design such a compiler to satisfy straightforward instantiations of Theorems 5 and 6. A **per-program correctness theorem** is impossible here, since inputs cannot be analyzed at compile time.

An alternative adaptation to the correctness theorem is to **weaken the value relation** by allowing multiple values at the target level to map to the same source value, instantiating Theorem 8. For the example of a boolean source and 32-bit target, suppose the compiler treats $0$ as $\mathsf{false}$ and all nonzero values as $\mathsf{true}$. Define the following relation on source and target values:

$$\approx_{\mathcal{V}} \triangleq \{(\mathsf{false}, 0)\} \cup \{(\mathsf{true}, \mathbf{n}') \mid \mathbf{n}' \neq 0\} \cup \{(\mathsf{n}, \mathbf{n}')\}$$

where $\mathsf{n}$, $\mathbf{n}'$ range over 32-bit integers. Then, define $\approx_{\mathcal{T}}$ in Theorem 8 by lifting $\approx_{\mathcal{V}}$, such that $\mathsf{t} \approx_{\mathcal{T}} \mathbf{t}'$ holds if and only if for every pair of corresponding values $\mathsf{v}$, $\mathbf{v}'$ that appear in $\mathsf{t}$ and $\mathbf{t}'$,

we have $v \approx_{\mathcal{V}} v'$. We have effectively defined the problem of unrepresentable input away, so the target program does not have to crash upon unrepresentable input. However, this design may not always be possible: for complex data types, there may be no good way to map bit sequences that do not satisfy the representation to particular instances of the data type.

### 3.2.5 I/O failure

Programs interact with their environment by means of potentially unreliable input-output constructs, including files, the command line, shared memory, a graphical device, or the network, each of which may fail. However, high-level language designers may not want to expose the details of handling I/O, instead hiding those details inside the compiler behind simple `input()` and `output()` primitives. With this design, I/O failure becomes a target-language error which is unrepresentable in the source, posing issues for verified compilation.

A simple adaptation is to **modify the source language** such that `input()` and `output()` can fail. Binary success or failure can obscure the low-level details that the source language is supposed to abstract over, while avoiding the need for the compiler to try to hide the underlying failure in some way. This is the approach most high-level languages take, and is arguably the most "principled" approach as well.

We consider other designs for the sake of completeness. It is straightforward to adapt the **per-execution** and **prefix-correct theorems**: compilers satisfying the resulting correctness theorems do not force the source programmer to explicitly handle I/O failure, instead producing programs that crash if an I/O call fails. Note, however, that it is impossible to adapt the **per-program correctness theorem** to this setting, since I/O errors cannot be analyzed

statically. Similarly, since the error does not involve the values of variables, we would not be able to **weaken the value relation** appropriately. Finally, **modifying the compilation strategy** to avoid the issue could entail retrying the I/O operation until it succeeds, which (while potentially of some utility in practice) would fail to provide any universal guarantees without additional assumptions.

### 3.2.6 Dynamic linking failure

Dynamic linking is present in many modern programming languages, allowing programs to link to precompiled binaries or bytecode at runtime. This can produce a target-language error, however, if the linked file is incorrectly formatted or not found. Since the source language usually hides the details of linking from the programmer, this error is unrepresentable at the source level. Superficially, this error seems like a case of I/O failure, since it results from failure to read a particular file. However, dynamic linking errors can occur anywhere in the program, without requiring any explicit I/O—they may be generated when a library is loaded, or only when code from a linked library is actually called.

Although there are important differences with I/O failure, the adaptations for verified compilers are similar. A straightforward approach is to **modify the source language** to allow dynamic linking to fail, as exemplified by Java: the Java 16 language specification [18] states that if an error occurs during class loading, a Java error of class `LinkageError` will be thrown when the running code attempts to use the class that failed to load. This error can be caught at the source level, meaning that Java programs could in principle try to recover gracefully from a dynamic linking error. Even if the error is fatal (rather than catchable), its

existence at the source level allows us to prove a standard semantic preservation theorem such as Theorem 2.

Alternatively, we could straightforwardly adapt the **per-execution** and **prefix-correct theorems** to account for dynamic linking errors. We would not need to allow linking to fail at the source level; rather, the theorem would implicitly require dynamic linking errors to cause the program to crash. On the other hand, we could not use the **per-program correctness theorem**, nor could we **weaken the value relation**, for the same reasons as discussed in Section 3.2.5. Attempting to discern a **modified compilation strategy** that could account for this error leads us to impractical designs such as attempting to download the missing linked library from some network repository.

### 3.2.7 Process interruption

The code emitted by a compiler may be executed on a variety of platforms, but in most cases it will run in a user-privileged process under some operating system. An accurate target semantics, then, must also account for behaviors at the target level caused by the operating system. These may include various system calls, such as for device communication, resource management, and process control. We could imagine augmenting the target semantics with a set of built-in system calls that have the appropriate effects on the modeled machine state.

Furthermore, operating systems also have behaviors and effects that are not directly invoked by a running process. For example, other processes or the operating system may force the process to terminate by a mechanism such as POSIX signals [21]. Verified compilers may or may not need to model signals and other such behaviors explicitly, but doing so increases re-

alism and reduces the gap between the model used for verification and the actual system that will run the compiled code. Furthermore, as discussed in Chapter 1, users of verified compilers who compose the compiler correctness theorem with source-level verification results obtain an end-to-end theorem that more accurately represents the properties of the resulting executable.

However, some process interruptions via signal (such as `SIGKILL`) are target-language errors that cannot be usefully represented in the source. This presents issues for verified compilers: clearly, a target program that terminates prematurely due to a signal or interruption does not have the same behavior as its source.

We note that **modifying the source language** is not useful: doing so would require us to add programmer-visible source constructs that handle interrupts, but—by definition—a process that has been interrupted cannot continue executing in order to process the interrupt.[¶] Similarly, **modifying the compilation strategy** seems to require the compiler to insert code to ignore interrupts into all programs; again, this cannot generally be done.

We thus consider adaptations to the compiler correctness theorem. Most generally, a process may be interrupted and terminated at any time. We could therefore instantiate the **prefix-correct theorem**, giving the following adaptation of Theorem 6:

**Theorem 9** (Prefix-correct theorem for process interruption)**.**

$$\forall \mathsf{S} \in \mathsf{Source}, \forall t \in \mathit{Trace}^{\infty},\ [\![\mathsf{S}]\!]_C \rightsquigarrow t \implies$$
$$\left(\mathsf{S} \rightsquigarrow t \vee (\exists m \in \mathit{Trace}, \exists t' \in \mathit{Trace}^{\infty}, t = m +\!\!+ [\mathbf{INT}] \wedge \mathsf{S} \rightsquigarrow m +\!\!+ t')\right)$$

[¶]Note that POSIX does allow processes to define signal handlers for some signals, but `SIGKILL` and similar signals cannot be caught or ignored [21].

Our Coq implementation, described in Appendix A, uses Theorem 9 as its compiler correctness theorem.

Note that we cannot use the **per-program correctness theorem** here, since there is no statically provable predicate over source programs that guarantees that a process cannot be interrupted. Likewise, we cannot **weaken the value relation**, since interruption is not a property of values in the languages.

In a sense, process interruption is the most pathological case for target-language error: it is triggered by parts of the machine state totally external to the target model, and can occur at any point during execution. We therefore cannot reason about it more carefully than Theorem 9 in the context of semantic preservation. It may be possible to make weaker guarantees about processes that are interrupted or induced to terminate prematurely; we discuss these guarantees in Chapter 4.

## 3.3 Proof technique for nondeterminism

In Section 2.3.1, we noted that a standard proof technique for backward simulation compiler correctness theorems involves proving both forward simulation for the compiler and determinism for the target language. This technique is applicable for proving many of the theorems presented in Section 3.2. For example, the correctness theorems stated in Section 3.2.1 are provable with this technique: memory exhaustion occurs deterministically when the target machine has allocated all available memory, so we can modify the semantics of the target language to track the amount of memory allocated and deterministically issue the out-of-

memory error when necessary. Indeed, this is the technique CakeML uses to prove the two versions of its correctness theorem described in Section 3.2.1 [43, 17].

However, some of the errors we discuss occur nondeterministically. In particular, process interruption can occur at any point during execution of the program and can be triggered by machine state not visible to the program. Thus, adding process interruption makes our target semantics nondeterministic, precluding the unmodified use of the proof technique described in Section 2.3.1.

In general, proving Theorem 6 for nondeterministic target languages may necessitate back-translation. However, the nondeterminism introduced by process interruption is of a particularly simple form: either the machine executes the next instruction, or it is interrupted and terminates immediately. More generally, nondeterministic target-language errors may cause the machine to execute a short code segment to report the error, then terminate.

The simplicity of this nondeterminism suggests that it might be possible to prove adaptations of Theorem 2 using adaptations of the standard proof technique. In particular, for every $\mathbf{T} \in \mathbf{Target}$, we ought to be able to define some "normal" trace $t_n$ that is generated when $\mathbf{T}$ executes without ever being interrupted. Then, we should be able to prove that either $\mathbf{T}$ admits $t_n$, or $\mathbf{T}$ admits some finite prefix of $t_n$ followed by the error. This is reminiscent of the structure of Theorem 6: either the target trace is a source trace, or it is a finite prefix of a source trace followed by the error. Indeed, if $\mathbf{T} = [\![\mathsf{S}]\!]_C$ for some $\mathsf{S}$, it appears that all we would need to do to obtain Theorem 6 is show that $\mathsf{S}$ admits $t_n$.

We formalize this intuition as follows. To formalize the notion of executing a target program without encountering interruption, we separate the small-step relation of the target

language into two layers: a deterministic relation $\longrightarrow_{\text{det}}$ containing all the "normal" machine transitions, and the full semantics $\longrightarrow$. We thus have the following inference rules for $\longrightarrow$:

**Definition 3** (Inference rules for $\longrightarrow$ relation in terms of $\longrightarrow_{\text{det}}$)**.** Letting metavariable $\sigma$ range over states of **Target** and **e** range over events:

$$\textsc{Normal}\ \frac{\sigma \xrightarrow{e}_{\text{det}} \sigma'}{\sigma \xrightarrow{e} \sigma'} \qquad\qquad \textsc{Err}\ \frac{}{\sigma \xrightarrow{\text{Error}} \sigma_{\text{err}}}$$

where $\sigma_{\text{err}}$ is an irreducible error state.

We then define $\rightsquigarrow_{\text{det}}$ and $\rightsquigarrow$ by substituting $\longrightarrow_{\text{det}}$ and $\longrightarrow$, respectively, into Definition 1.

The proof of Theorem 6 consists of proving two lemmas, corresponding to the two components of the informal proof outline given above.

**Lemma 1** (Backward simulation under deterministic semantics)**.**

$$\forall \mathsf{S} \in \mathsf{Source}, \forall t \in \mathit{Trace}^{\infty}, [\![\mathsf{S}]\!]_C \rightsquigarrow_{\text{det}} t \implies \mathsf{S} \rightsquigarrow t$$

Lemma 1 is proved using the standard technique described in Section 2.3.1: we first prove a forward simulation from Source to **Target** executed under the deterministic semantics, then use determinism of $\longrightarrow_{\text{det}}$ to obtain the backward simulation result.

**Lemma 2** (Modified backward simulation from full semantics to deterministic semantics)**.**

$$\begin{aligned}&\forall \mathbf{T} \in \mathbf{Target}, \forall t \in \mathit{Trace}^{\infty}, \mathbf{T} \rightsquigarrow t \implies \\ &\quad (\mathbf{T} \rightsquigarrow_{\text{det}} t \vee \exists m \in \mathit{Trace}, t' \in \mathit{Trace}^{\infty}, t = m +\!\!+ [\mathbf{Error}] \wedge \mathbf{T} \rightsquigarrow_{\text{det}} (m +\!\!+ t'))\end{aligned}$$

Lemma 2 can be thought of as a modified backward simulation between target programs under the deterministic semantics and the full semantics. Since the full semantics are nondeterministic, back-translation is required. However, the back-translation is trivial: target states are identical and execute in lock-step under either semantics (unless the error occurs), so we can use an identity back-translation. This theorem can thus be proven straightforwardly by induction on the derivations of $\longrightarrow$ for each step in the derivation of $\mathbf{T} \rightsquigarrow t$.

Finally, we can use Lemmas 1 and 2 to prove our top-level compiler correctness theorem, which is equivalent to Theorem 6:

**Theorem 1** (Prefix-correct theorem for target with nondeterministic error)

*If Lemmas 1 and 2 hold, then we have:*

$$\forall \mathsf{S} \in \mathsf{Source}, \forall t \in \mathit{Trace}^{\infty}, [\![\mathsf{S}]\!]_C \rightsquigarrow t \implies$$
$$(\mathsf{S} \rightsquigarrow t \vee \exists m \in \mathit{Trace}, t' \in \mathit{Trace}^{\infty}, t = m +\!\!+ [\mathbf{Error}] \wedge \mathsf{S} \rightsquigarrow (m +\!\!+ t'))$$

*Proof.* Let $\mathsf{S} \in \mathsf{Source}$ and $t \in \mathit{Trace}^{\infty}$ such that $[\![\mathsf{S}]\!]_C \rightsquigarrow t$.

By Lemma 2, we have:

$([\![\mathsf{S}]\!]_C \rightsquigarrow_{\mathbf{det}} t \vee \exists m \in \mathit{Trace}, t' \in \mathit{Trace}^{\infty}, t = m +\!\!+ [\mathbf{Error}] \wedge [\![\mathsf{S}]\!]_C \rightsquigarrow_{\mathbf{det}} (m +\!\!+ t'))$.

But by Lemma 1, we have $[\![\mathsf{S}]\!]_C \rightsquigarrow_{\mathbf{det}} t \implies \mathsf{S} \rightsquigarrow t$ and

$[\![\mathsf{S}]\!]_C \rightsquigarrow_{\mathbf{det}} m +\!\!+ t' \implies \mathsf{S} \rightsquigarrow m +\!\!+ t'$.

We can apply these implications to obtain:

$(\mathsf{S} \rightsquigarrow t \vee \exists m \in \mathit{Trace}, t' \in \mathit{Trace}^{\infty}, t = m +\!\!+ [\mathbf{Error}] \wedge \mathsf{S} \rightsquigarrow (m +\!\!+ t'))$. □

In summary, our adapted proof technique to prove theorems of the form of Theorem 6 for target languages with this kind of limited nondeterminism consists of the following three steps:

1. Separate the semantics of the target language into $\longrightarrow_{\text{det}}$ and $\longrightarrow$.

2. Prove Lemma 1: backward simulation between the source language and the target language executed under $\longrightarrow_{\text{det}}$.

3. Prove Lemma 2: modified backward simulation between the target language executed under $\longrightarrow_{\text{det}}$ and under $\longrightarrow$.

Separating the target semantics into deterministic and nondeterministic layers lets us obtain standard results under the deterministic semantics. Furthermore, we avoid having to reason about nondeterminism in the target in the context of compiled source programs, sidestepping the back-translation issue mentioned in Section 2.3.1.

We expect that this proof technique will straightforwardly generalize to any nondeterministic target-language error whose effect is to execute some small code segment, then terminate. However, undetectable errors that do not immediately terminate the target program would preclude the use of this proof technique. In particular, the proof of Lemma 2 would require a complex back-translation to map states in executions with errors to states in executions without errors, and it is unclear how to construct such a back-translation in general.

We have developed in Coq an implementation of a verified compiler for a target language featuring nondeterministic process interruption. Our implementation uses the technique we describe above to prove that the compiler satisfies Theorem 9. In addition to demonstrating

the practical utility of this proof technique, our implementation provides an example of how a verified compiler designer might instantiate one of our template correctness theorems for a particular source-target language pair. More details about our Coq development are given in Appendix A.

# 4

# Weaker guarantees in the absence of correctness

In this chapter, we discuss guarantees weaker than semantic preservation that can be proven to hold over all target executions. These can give verified compiler users confidence that compiled programs that do encounter errors will not exhibit arbitrarily bad behavior, potentially making the compiler more useful in practice.

In Section 3.1, we stated Theorems 5 and 6 which exclude executions that encounter target-language errors from the semantic preservation guarantee. Users of verified compilers may

still want some guarantees about the compiled code that hold in *all* executions, not just those that do not encounter errors. As we have discussed, such guarantees must be weaker than semantic preservation. However, they can guarantee that in cases where the program does encounter a target-language error, it does not execute arbitrary bad behaviors such as disclosing memory or jumping to arbitrary code. We seek to capture these cases in our formalism.

We extend our model of execution to capture additional information $\tau$, which consists of the machine state at every program point in the execution of the program. To do so, we redefine the reflexive transitive closure relation as $\longrightarrow^* \ : S_L \times \mathit{Trace} \times S_L^* \times S_L$, where $S_L^*$ denotes a finite list of states from $S_L$, using the following three rules:

$$\frac{}{\sigma \xrightarrow{[],[]}{}^* \sigma} \qquad \frac{\sigma \longrightarrow \sigma' \quad \sigma' \xrightarrow{t,\tau}{}^* \sigma''}{\sigma \xrightarrow{t,\sigma' :: \tau}{}^* \sigma''} \qquad \frac{\sigma \xrightarrow{e} \sigma' \quad \sigma' \xrightarrow{t,\tau}{}^* \sigma''}{\sigma \xrightarrow{e :: t,\sigma' :: \tau}{}^* \sigma''}$$

Likewise, we redefine the infinite transitive closure relation as $\longrightarrow^\infty \ : S_L \times \mathit{Trace}^\infty \times S_L^\infty$, where $S_L^\infty$ denotes an infinite list of states from $S_L$, using the following two rules:

$$\frac{\sigma \longrightarrow \sigma' \quad \sigma' \xrightarrow{t,\tau}{}^\infty}{\sigma \xrightarrow{t,\sigma' :: \tau}{}^\infty} \qquad \frac{\sigma \xrightarrow{e} \sigma' \quad \sigma' \xrightarrow{t,\tau}{}^\infty}{\sigma \xrightarrow{e :: t,\sigma' :: \tau}{}^\infty}$$

Now, let $P \overset{\tau}{\rightsquigarrow} t$ be the predicate generated by using these modified definitions of $\longrightarrow^*$ and $\longrightarrow^\infty$ in Definition 1. Intuitively, $P \overset{\tau}{\rightsquigarrow} t$ denotes that program $P$ admits trace $t$ with machine states $\tau$ during execution. The machine configurations appearing in $\tau$ contain all program state used by the semantics of the language of $P$. These configurations could include register state, stack and heap memory, a continuation, or local variables, depending on the semantics

of the language of $P$. Such machine state is not externally visible, so top-level compiler correctness theorems generally do not mention it. However, if the program terminates prematurely due to a target-language error aspects of this state may become visible, meaning that we would want to give guarantees over those aspects as well as the externally visible events discussed in the rest of this thesis. Furthermore, some of the weaker guarantees of interest involve compiler-enforced representation invariants, which are captured only in $\tau$.

We can then give the following template theorem statement for stating guarantees that hold over all executions in the target language:

**Theorem 10** (Guarantee over all target executions)**.**

$$\forall \mathsf{S} \in \mathsf{Source}, \forall t \in \mathit{Trace}^{\infty}, \; [\![\mathsf{S}]\!]_C \overset{\tau}{\rightsquigarrow} t \implies \mathbf{Q}(\boldsymbol{\tau})$$

where $\mathbf{Q}$ is a predicate enforcing the safety property of interest over $\boldsymbol{\tau}$.

It may not be immediately obvious that Theorem 10 can be proved without running into the issue of the target-language error being unrepresentable in the source language. After all, the theorem quantifies over source programs in Source. Note, however, that Theorem 10 does not mention *execution* of a source program. Instead, we effectively quantify over executions of all possible outputs of compiler $C$. Furthermore, since $\mathbf{Q}$ is not given the source program as an argument, we cannot choose a predicate $\mathbf{Q}$ that makes Theorem 10 equivalent to Theorem 2 or any other semantic preservation result. Thus, proofs of Theorem 10 do not need to use the source semantics in which errors are unrepresentable. Consequently, any predicate $\mathbf{Q}$ that satisfies Theorem 10 will hold for all executions of compiled code.

### 4.1 Example predicates

We informally describe examples of predicates Q that may be of interest in various settings.

Control-flow integrity. Theorem 5, the per-execution correctness theorem, allows target programs to have any behavior that terminates with a target-language error. In practice, we may want to prevent execution of arbitrary code in executions that encounter errors. For an assembly-like target, this could be expressed by a safety property over the instruction pointer: in all executions, the instruction pointer should remain within the code segments of the currently running program. The proof that such a guarantee holds would likely require proving that the code generated by the compiler jumps only to appropriate targets. More sophisticated models of the source program's control flow could be used to more precisely restrict all target executions, with correspondingly greater proof effort.

Memory safety. We may wish to prove that target-language code does not violate memory safety even in cases of error. For a source language with modules, objects, or other forms of encapsulation, we may wish to enforce a stronger guarantee: for instance, that only code from module M can directly manipulate state owned by M. Such an encapsulation guarantee might help contain the effects of a target-language error to just the module that encountered the error, even though the semantics of the target-language execution may differ significantly from any possible source-language execution.

Graceful exit. We may wish to prove that upon premature exit due to an error, the compiled code leaves the machine in a clean or safe state. What precisely constitutes a clean state

will depend on the particular system, but could include properties such as ensuring that files are closed, that network connections are appropriately terminated, or that secure information is not left in memory locations that could become externally visible. These requirements can be expressed as safety properties over the last state in $\tau$, and proven to hold by reasoning about the code that handles errors and terminates the program. Note that for some errors—such as process interruption and memory exhaustion—the error condition may not allow any code to execute after it arises, making graceful exit impossible.

# 5

# Related work

In this chapter, we discuss key contributions in verified compilation and related fields in order to situate our work within the literature.

## 5.1 Complex target models

There is a significant body of work examining ways of extending compiler correctness theorems for complex target languages and machine models. This work encompasses both theoretical frameworks that account for target-language errors and implementations of verified compilers for specific language pairs.

On the theoretical side, Abate et al. [1] present a general definition of compiler correctness* parameterized on a relation between traces, equivalent to our Theorem 8. They show that by defining the trace relation appropriately, Theorem 8 can account for a variety of differences in the externally visible behaviors of the source and target languages, including some target-language errors. In particular, defining $\approx_{\mathcal{T}}$ as

$$\{(\mathsf{t}, \mathbf{t}') \mid (\mathsf{t} = \mathbf{t}') \ \vee \ (\exists \mathbf{m}' \in \mathbf{Trace}, \mathbf{t}' = \mathbf{m}' +\!\!+ [\mathbf{err}])\}$$

in Theorem 8 produces a theorem equivalent to Theorem 5. Similarly, defining $\approx_{\mathcal{T}}$ as

$$\{(\mathsf{t}, \mathbf{t}') \mid (\mathsf{t} = \mathbf{t}') \ \vee \ (\exists \mathsf{m} \in \mathsf{Trace}, \mathbf{m}' \in \mathbf{Trace}, \mathsf{m} = \mathbf{m}' \wedge \mathsf{m} < \mathsf{t} \wedge \mathbf{t}' = \mathbf{m}' +\!\!+ [\mathbf{err}])\}$$

in Theorem 8 produces a theorem equivalent to Theorem 6. The authors describe how to choose trace relations that model the top-level correctness theorems of CompCert and the original version of CakeML, as well as a system similar to our example for unrepresentable input in Section 3.2.4. Note that since their relation is parameterized over traces, it cannot be used to encode a per-program correctness theorem analogous to our Theorem 7. Since their focus is purely on compiler correctness theorems, they also do not discuss the first three adaptations we described in Section 3.1: modifying the source, modifying the target, and modifying the compilation strategy.

On the implementation side, past work has built verified compilers that account for some of the specific errors we describe in Section 3.2. Férée et al. [15] build a verified implementation of file I/O for CakeML, accounting for nondeterminism in the file system. They specify

*Definition 1.2 in Abate et al. [1].

low-level, nondeterministic implementations of `read()` and `write()`, high-level versions that hide the nondeterminism behind repeated calls to the low-level versions, and the liveness property that "the file system eventually reads/writes at least one character." Their design combines elements of our modify-the-source and modify-the-compilation-strategy adaptations.

Boldo et al. [9] implement and verify compilation of floating-point arithmetic in CompCert, proving that the compiler emits code that respects the semantics of floating-point operations in C. This requires them to formalize the IEEE-754 floating-point standard in Coq. C has floating-point numbers, as opposed to rationals, so this work falls under our modify-the-source adaptation.

## 5.2 Separate compilation

One of the key recent directions in verified compilation research has been verification of separate compilation: compiling partial programs such that their compositions obey correctness guarantees. This work is largely orthogonal to our focus on target-language errors, but separate compilation may introduce new errors that can be addressed with the adaptations we describe: for example, our approaches to dynamic linking failure may be appropriate when constructing the top-level correctness theorem for the composition of separately compiled programs. Furthermore, the considerations we identified are also important when evaluating separate compilation designs.

Patterson and Ahmed [36] give a general compositional compiler correctness theorem[†] whose parameters can be instantiated to give a variety of theorems used by existing verified separate compilation work. Their theorem requires definitions of various components of the linking system, including the language in which linking is done, the set of all programs that can be linked, and lifting functions from the target to the linking medium (to produce "source-like" components whose behavior the target components should refine). This theorem gives easily understandable guarantees once parameters are chosen, but depending on the setting, proving it may require difficult back-translations from target to source.

SepCompCert [23] extends CompCert with separate compilation, albeit with the restriction that only components compiled by SepCompCert can be linked. Its theorem and formalization are easy to understand and relatively straightforward to prove, but its set of linkable components is somewhat constrained. Compositional CompCert [42] takes a different approach, allowing for linking components from any language and compiler using CompCert's memory model. To support this larger set of linkable programs, its theorem statement is more complex than SepCompCert's and its proof is significantly more difficult. CompCertM [41] bridges the gap between SepCompCert's and Compositional CompCert's approaches: it allows linking with components written in other languages (in its case, handwritten assembly), but it develops a novel verification technique that allows it to avoid some of the proof overhead incurred by Compositional CompCert. Its top-level correctness theorem and formalism are still somewhat complex, however, since they must account for linking with components in different languages.

[†]Theorem 4.1 in Patterson and Ahmed [36].

## 5.3 Secure compilation

Secure compilation is an increasingly important and related area that studies how to construct compilers that emit code satisfying the same security properties as the input source. For example, compilers for source languages with public and private annotations for local variables need to store private variables so that their values cannot be accessed by foreign code [35]. Similar to our work on target-language errors, proving preservation of security properties requires reasoning about behaviors of the target language that cannot be represented at the source level. In this setting, adversaries are typically represented by *contexts* in which source or compiled code is executed; the context can be thought of as a program with a hole into which the source or compiled code is inserted. While the semantics of high-level languages might restrict the context to entering the source program only via normal function calls, an assembly-like target context might be able to jump to arbitrary locations in the compiled code, creating security vulnerabilities not present in the source program.

Patrignani et al. [35] survey a variety of approaches to building provably secure compilers. They focus on the criterion of full abstraction, which requires that programs that are indistinguishable by adversarial contexts at the source level must also be indistinguishable by adversarial contexts at the target level. This criterion is attractive since it means that source programmers can understand the security properties of their code without reasoning about the target semantics. Patrignani et al. [35] discuss ways that secure compiler writers might achieve full abstraction, ranging from developing sophisticated type systems for target languages to inserting runtime checks or cryptographic operations during compilation.

Full abstraction is a very strong security property that may be difficult to prove, and it may be impossible in the presence of side-channels [2]. Abate et al. [2] explore a broad range of security guarantees characterized in terms of execution traces of programs similar to the trace-based compiler correctness theorems discussed in our work. Their approach is also similar to ours in that they seek to systematize a large space of possible designs. In their case, the designs correspond to classes of execution trace properties of varying expressive power, representing attackers with varying capabilities.

Evaluation of the security implications of our adaptations is beyond the scope of this thesis. However, the target-language errors we examine could be important sources of security vulnerabilities. We hope that future work will examine our adaptations from a security point of view—in particular, investigating what security properties hold for a compiler that does not satisfy full semantic preservation.

# 6

# Conclusion

In this thesis, we explored and systematized the design space faced by writers of verified compilers when adapting their systems to account for errors in the target language. We categorized these adaptations according to the component of the verified compiler they modify and gave template theorem statements for adaptations that modify the compiler correctness theorem. We then applied each of these adaptations to seven representative types of target-language error, showing that approaches that are suitable or commonly used for one error may be inapplicable or even absurd for another. Noting that some errors make the target language nondeterministic, we developed an adaptation to the standard compiler correctness proof tech-

nique that accounts for this limited nondeterminism with relatively minimal proof overhead. Given that errors prevent us from proving correctness for every target execution, we consider guarantees that do hold over all executions and can prevent *specific* unreasonable behaviors. Our comparison and evaluation of these designs has been guided by four key considerations: precision, simplicity, realism, and provability. Future builders of verified compilers can use our systematization when designing their own systems, choosing appropriate adaptations for their particular target language and setting. Users of those compilers can, in turn, compose our theorems with guarantees they produce in other stages of the code creation pipeline in order to understand exactly what properties their executables ultimately satisfy.

There are a few possible directions for future work based on this project. First, many of the errors we discussed have not been explicitly included in the target-language models of existing verified compilers. We hope that future verified compiler writers will find our systematization of the design space of adaptations useful when developing their own systems. Second, we hope that this work will inspire the research community to develop more adaptations for verified compilers to account for target-language errors, and to evaluate these adaptations in light of our systematization. Third, we hope that future work will examine the security implications of the errors we model and the adaptations we propose. In particular, given that target-language errors prevent us from proving full semantic preservation, it is important to understand what security properties hold for a compiler that does not satisfy full semantic preservation.

# A

# A verified compiler for a nondeterministic target

We summarize our implementation in the Coq proof assistant of a verified compiler satisfying Theorem 9 and using the proof technique we described in Section 3.3. The full source code of our Coq development is available at https://github.com/pratapsingh1729/adapting-verified-compilation.

Our verified compiler implementation is adapted from Xavier Leroy's tutorial "Proving the correctness of a compiler," most recently presented at the EUTypes Summer School in

2019 [29]. Leroy presents a stylized verified compiler from a simple imperative source language to a stack-machine target, as well as some verified optimization passes at the source level. We choose this implementation as our base because it is rich enough to capture some of the complex control-flow constructs that can make verifying a compiler difficult, but also simple enough that adding new features does not require prohibitive proof effort (unlike a "production-quality" system such as CompCert). Past work, including Abate et al. [1], has also adapted Leroy's compiler to demonstrate new formalisms and features for the same reasons.

The two main features we require that are not present in Leroy's compiler are explicit traces of events and nondeterminism via process interruption in the target language. For simplicity, we assume that both the source and target languages have the same externally visible events:

- $IN(z)$ means take $z$ as input (where $z \in \mathbb{Z}$).
- $OUT(z)$ means give $z$ as output (where $z \in \mathbb{Z}$).

Note that to avoid reasoning about fixed-width integer overflow, we use mathematical integers in our language. A more realistic verified compiler might combine this design with one of the adaptations described in Section 3.2.2.

Additionally, we need to add a model of the state of the external world with which the executing programs interact: this must provide inputs to and receive outputs from the program currently executing. The external world is generally nondeterministic, but since this nondeterminism is external to the program under execution, we can abstract it out of our source and target semantics [27]. Let $G$ denote an external world state, and define the following two functions:

- *take*$(G)$ maps world state $G$ to integer $z$ (given as input) and new world state $G'$.
- *give*$(G, z)$ maps world state $G$ and integer $z$ (received as output) to new world state $G'$.

Our compiler correctness theorem requires that both the source and target programs begin executing in the same world state $G$.

SOURCE LANGUAGE: IMP The source language is IMP, a simple imperative language featuring arithmetic and boolean expressions, conditionals, and `while` loops. The syntax of IMP is given in Figure A.1.

| | |
|---|---|
| arithmetic expressions | $\mathsf{a} ::= \mathsf{x} \mid \mathsf{z} \mid \mathsf{a} + \mathsf{a} \mid \mathsf{a} - \mathsf{a}$ |
| boolean expressions | $\mathsf{b} ::= \textbf{TRUE} \mid \textbf{FALSE} \mid \mathsf{a} = \mathsf{a} \mid \mathsf{a} \leq \mathsf{a} \mid \textbf{NOT}\ \mathsf{b} \mid \mathsf{b}\ \textbf{AND}\ \mathsf{b}$ |
| commands | $\mathsf{c} ::= \textbf{SKIP} \mid \mathsf{x} := \mathsf{a} \mid \mathsf{c}; \mathsf{c} \mid \textbf{INPUT}\ \mathsf{x} \mid \textbf{OUTPUT}\ \mathsf{a}$ |
| | $\mid \textbf{IF}\ \mathsf{b}\ \textbf{THEN}\ \mathsf{c}\ \textbf{ELSE}\ \mathsf{c} \mid \textbf{WHILE}\ \mathsf{b}\ \textbf{DO}\ \mathsf{c}$ |
| continuations | $\mathsf{k} ::= \mathbf{K_{stop}} \mid \mathsf{c} ;; \mathsf{k}$ |
| stores | $\mathsf{s} ::= \mathsf{Var} \mapsto \mathbb{Z}$ |

**Figure A.1:** Syntax of IMP. Metavariable x ranges over variables in Var, and metavariable z ranges over integers. We assume that stores s are total and unambiguous, i.e., there is exactly one mapping in the store for each program variable.

The operational semantics of IMP are given in Figure A.2. We give a labeled transition semantics in terms of continuations and with an explicit store containing variables. IMP states thus consist of a command under execution, a continuation, a store, and a world state.

$$\frac{}{(x, s) \longrightarrow_a s(a)} \qquad \frac{(a_1, s) \longrightarrow_a z_1 \quad (a_2, s) \longrightarrow_a z_2}{(a_1 + a_2, s) \longrightarrow_a z} z = z_1 + z_2$$

$$\frac{}{(z, s) \longrightarrow_a z} \qquad \frac{(a_1, s) \longrightarrow_a z_1 \quad (a_2, s) \longrightarrow_a z_2}{(a_1 - a_2, s) \longrightarrow_a z} z = z_1 - z_2$$

$$\frac{}{(b, s) \longrightarrow_b b} b \in \{\textsf{TRUE}, \textsf{FALSE}\} \qquad \frac{(b, s) \longrightarrow_b \textsf{TRUE}}{(\textsf{NOT } b, s) \longrightarrow_b \textsf{FALSE}} \qquad \frac{(b, s) \longrightarrow_b \textsf{FALSE}}{(\textsf{NOT } b, s) \longrightarrow_b \textsf{TRUE}}$$

$$\frac{(a_1, s) \longrightarrow_a z_1 \quad (a_2, s) \longrightarrow_a z_2}{(a_1 = a_2, s) \longrightarrow_b \textsf{TRUE}} \text{ if } z_1 = z_2 \qquad \frac{(a_1, s) \longrightarrow_a z_1 \quad (a_2, s) \longrightarrow_a z_2}{(a_1 = a_2, s) \longrightarrow_b \textsf{FALSE}} \text{ if } z_1 \neq z_2$$

$$\frac{(a_1, s) \longrightarrow_a z_1 \quad (a_2, s) \longrightarrow_a z_2}{(a_1 \leq a_2, s) \longrightarrow_b \textsf{TRUE}} \text{ if } z_1 \leq z_2 \qquad \frac{(a_1, s) \longrightarrow_a z_1 \quad (a_2, s) \longrightarrow_a z_2}{(a_1 \leq a_2, s) \longrightarrow_b \textsf{FALSE}} \text{ if } z_1 > z_2$$

$$\frac{(b_1, s) \longrightarrow_b b_1' \quad (b_2, s) \longrightarrow_b b_2'}{(b_1 \textsf{ AND } b_2, s) \longrightarrow_b \textsf{TRUE}} \text{ if } b_1' = b_2' = \textsf{TRUE}$$

$$\frac{(b_1, s) \longrightarrow_b b_1' \quad (b_2, s) \longrightarrow_b b_2'}{(b_1 \textsf{ AND } b_2, s) \longrightarrow_b \textsf{FALSE}} \text{ if } b_1' = \textsf{FALSE} \text{ or } b_2' = \textsf{FALSE}$$

$$\frac{(a, s) \longrightarrow_a z}{\langle x := a, k, s, G\rangle \longrightarrow \langle \textsf{SKIP}, k, [x := z]s, G\rangle} \qquad \frac{}{\langle c_1; c_2, k, s, G\rangle \longrightarrow \langle c_1, c_2 ;; k, s, G\rangle}$$

$$\frac{(b, s) \longrightarrow_b \textsf{TRUE}}{\langle \textsf{IF } b \textsf{ THEN } c_1 \textsf{ ELSE } c_2, k, s, G\rangle \longrightarrow \langle c_1, k, s, G\rangle}$$

$$\frac{(b, s) \longrightarrow_b \textsf{FALSE}}{\langle \textsf{IF } b \textsf{ THEN } c_1 \textsf{ ELSE } c_2, k, s, G\rangle \longrightarrow \langle c_2, k, s, G\rangle}$$

$$\frac{(b, s) \longrightarrow_b \textsf{FALSE}}{\langle \textsf{WHILE } b \textsf{ DO } c, k, s, G\rangle \longrightarrow \langle \textsf{SKIP}, k, s, G\rangle}$$

$$\frac{(b, s) \longrightarrow_b \textsf{TRUE}}{\langle \textsf{WHILE } b \textsf{ DO } c, k, s, G\rangle \longrightarrow \langle c, \textsf{WHILE } b \textsf{ DO } c ;; k, s, G\rangle}$$

$$\frac{}{\langle \textsf{SKIP}, c ;; k, s, G\rangle \longrightarrow \langle c, k, s, G\rangle} \qquad \frac{take(G) = (z, G')}{\langle \textsf{INPUT } x, k, s, G\rangle \xrightarrow{IN(z)} \langle \textsf{SKIP}, k, [x := z]s, G'\rangle}$$

$$\frac{(a, s) \longrightarrow_a z \quad give(G, z) = G'}{\langle \textsf{OUTPUT } a, k, s, G\rangle \xrightarrow{OUT(z)} \langle \textsf{SKIP}, k, s, G'\rangle}$$

**Figure A.2:** Small-step operational semantics of IMP.

TARGET LANGUAGE: **MACH** The target language is **MACH**, a simple stack machine featuring conditional and unconditional jumps, arithmetic instructions, and an explicit variable store. The syntax of **MACH** is given in Figure A.3.

| | | |
|---|---|---|
| instructions | **i** ::= | **Iconst z** \| **Ivar x** \| **Isetvar x** \| **Iadd** \| **Iopp** \| **Ibranch δ** \| **Ibeq** $(\delta, \delta)$ \| **Ible** $(\delta, \delta)$ \| **Iinput** \| **Ioutput** \| **Ihalt** |
| programs | **C** ::= | list of instructions **i** |
| stacks | **t** ::= | $[\,] \mid \mathbf{z} :: \mathbf{t}$ |
| stores | **s** ::= | $\mathbf{Var} \mapsto \mathbb{Z}$ |

**Figure A.3:** Syntax of **MACH**. Metavariable **x** ranges over variables in **Var**, metavariable **z** ranges over integers, and metavariable **δ** ranges over program offsets. We assume that stores **s** are total and unambiguous, i.e., there is exactly one mapping in the store for each program variable.

The operational semantics of **MACH** are given in Figure A.4. **MACH** states consist of a program, a program counter, a stack, a store, and a world state. We model interruption as a single-step transition, emitting the interruption event **INT**, to a special configuration $\langle \mathbf{INT} \rangle$ from which no further steps are possible. A more realistic semantics could model program termination as taking multiple steps, but as long as the program cannot be interrupted more than once, this would not meaningfully change the proof. Rule INTR is the instantiation of Rule ERR specified in Definition 3.

COMPILATION SCHEME The compilation scheme is essentially unmodified from Leroy's tutorial [29]: we add only cases for the compilation of **INPUT** x and **OUTPUT** x. We refer readers to our Coq development for details of the compilation scheme.

$$\frac{\mathbf{C}[\mathbf{pc}] = \mathbf{Iconst\ z}}{\langle \mathbf{C}, \mathbf{pc}, \mathbf{t}, \mathbf{s}, G\rangle \longrightarrow \langle \mathbf{C}, \mathbf{pc}+1, \mathbf{z} :: \mathbf{t}, \mathbf{s}, G\rangle}$$

$$\frac{\mathbf{C}[\mathbf{pc}] = \mathbf{Ivar\ x}}{\langle \mathbf{C}, \mathbf{pc}, \mathbf{t}, \mathbf{s}, G\rangle \longrightarrow \langle \mathbf{C}, \mathbf{pc}+1, \mathbf{s}(\mathbf{x}) :: \mathbf{t}, \mathbf{s}, G\rangle}$$

$$\frac{\mathbf{C}[\mathbf{pc}] = \mathbf{Isetvar\ x}}{\langle \mathbf{C}, \mathbf{pc}, \mathbf{z} :: \mathbf{t}, \mathbf{s}, G\rangle \longrightarrow \langle \mathbf{C}, \mathbf{pc}+1, \mathbf{t}, [\mathbf{x} := \mathbf{z}]\mathbf{s}, G\rangle}$$

$$\frac{\mathbf{C}[\mathbf{pc}] = \mathbf{Iadd}}{\langle \mathbf{C}, \mathbf{pc}, \mathbf{z}_2 :: \mathbf{z}_1 :: \mathbf{t}, \mathbf{s}, G\rangle \longrightarrow \langle \mathbf{C}, \mathbf{pc}+1, \mathbf{z} :: \mathbf{t}, \mathbf{s}, G\rangle}\ z = z_1 + z_2$$

$$\frac{\mathbf{C}[\mathbf{pc}] = \mathbf{Iopp}}{\langle \mathbf{C}, \mathbf{pc}, \mathbf{z} :: \mathbf{t}, \mathbf{s}, G\rangle \longrightarrow \langle \mathbf{C}, \mathbf{pc}+1, \mathbf{z}' :: \mathbf{t}, \mathbf{s}, G\rangle}\ z' = -z$$

$$\frac{\mathbf{C}[\mathbf{pc}] = \mathbf{Ibranch}\ \boldsymbol{\delta}}{\langle \mathbf{C}, \mathbf{pc}, \mathbf{t}, \mathbf{s}, G\rangle \longrightarrow \langle \mathbf{C}, \mathbf{pc}+1+\boldsymbol{\delta}, \mathbf{t}, \mathbf{s}, G\rangle}$$

$$\frac{\mathbf{C}[\mathbf{pc}] = \mathbf{Ibeq}\ (\boldsymbol{\delta}_1, \boldsymbol{\delta}_2)}{\langle \mathbf{C}, \mathbf{pc}, \mathbf{z}_2 :: \mathbf{z}_1 :: \mathbf{t}, \mathbf{s}, G\rangle \longrightarrow \langle \mathbf{C}, \mathbf{pc}+1+\boldsymbol{\delta}_1, \mathbf{t}, \mathbf{s}, G\rangle}\ z_1 = z_2$$

$$\frac{\mathbf{C}[\mathbf{pc}] = \mathbf{Ibeq}\ (\boldsymbol{\delta}_1, \boldsymbol{\delta}_2)}{\langle \mathbf{C}, \mathbf{pc}, \mathbf{z}_2 :: \mathbf{z}_1 :: \mathbf{t}, \mathbf{s}, G\rangle \longrightarrow \langle \mathbf{C}, \mathbf{pc}+1+\boldsymbol{\delta}_2, \mathbf{t}, \mathbf{s}, G\rangle}\ z_1 \neq z_2$$

$$\frac{\mathbf{C}[\mathbf{pc}] = \mathbf{Ible}\ (\boldsymbol{\delta}_1, \boldsymbol{\delta}_2)}{\langle \mathbf{C}, \mathbf{pc}, \mathbf{z}_2 :: \mathbf{z}_1 :: \mathbf{t}, \mathbf{s}, G\rangle \longrightarrow \langle \mathbf{C}, \mathbf{pc}+1+\boldsymbol{\delta}_1, \mathbf{t}, \mathbf{s}, G\rangle}\ z_1 \leq z_2$$

$$\frac{\mathbf{C}[\mathbf{pc}] = \mathbf{Ible}\ (\boldsymbol{\delta}_1, \boldsymbol{\delta}_2)}{\langle \mathbf{C}, \mathbf{pc}, \mathbf{z}_2 :: \mathbf{z}_1 :: \mathbf{t}, \mathbf{s}, G\rangle \longrightarrow \langle \mathbf{C}, \mathbf{pc}+1+\boldsymbol{\delta}_2, \mathbf{t}, \mathbf{s}, G\rangle}\ z_1 > z_2$$

$$\frac{\mathbf{C}[\mathbf{pc}] = \mathbf{Iinput} \quad \mathit{take}(G) = (z, G')}{\langle \mathbf{C}, \mathbf{pc}, \mathbf{t}, \mathbf{s}, G\rangle \xrightarrow{IN(z)} \langle \mathbf{C}, \mathbf{pc}+1, \mathbf{z} :: \mathbf{t}, \mathbf{s}, G'\rangle}$$

$$\frac{\mathbf{C}[\mathbf{pc}] = \mathbf{Ioutput} \quad \mathit{give}(G, z) = (G')}{\langle \mathbf{C}, \mathbf{pc}, \mathbf{z} :: \mathbf{t}, \mathbf{s}, G\rangle \xrightarrow{OUT(z)} \langle \mathbf{C}, \mathbf{pc}+1, \mathbf{t}, \mathbf{s}, G'\rangle}$$

$$\textsc{Intr}\ \frac{}{\langle \mathbf{C}, \mathbf{pc}, \mathbf{t}, \mathbf{s}, G\rangle \xrightarrow{\mathbf{INT}} \langle \mathbf{INT}\rangle}$$

**Figure A.4:** Small-step operational semantics of MACH.

## A.1 Correctness theorem and proof structure

Our top-level correctness theorem is Theorem 9. We present here the Coq definitions of our top-level correctness theorem and of the two key lemmas in the proof for comparison with those described in Section 3.3. The names of predicates and theorems were chosen to be evocative; we refer readers to the Coq development for full definitions of the predicates mentioned in these theorems.

A small difference between our implementation and our formalism (as described in Section 2.1) is that in Coq, we give separate definitions of the predicate $P \rightsquigarrow t$ for the three kinds of termination behavior: termination, silent divergence, or divergence with infinitely many events. This allows us to avoid mixing induction and coinduction in our correctness proofs, which can pose difficulties due to Coq's presently limited support for coinduction. Our proof requires a lemma stating that every execution either terminates, diverges silently, or diverges with infinitely many events. This cannot be proved in Coq's constructive logic [29]; we therefore assume the excluded middle axiom and give a classical proof of this lemma.

### Top-level correctness theorems

```
Theorem compile_program_correct_terminating:
    forall c s g tr,
        machine_terminates (compile_program c) s g tr ->
        (exists s' g', imp_terminates c s g tr s' g') \/
        (exists m, tr = m ++ [ev_intr] /\ imp_admits_finite c s g m).

Theorem compile_program_correct_diverging_silently:
    forall c s g tr,
        machine_diverges_silently (compile_program c) s g tr ->
        imp_diverges_silently c s g tr.

Theorem compile_program_correct_diverging_with_inftrace:
    forall c s g itr itr',
        machine_diverges_with_inftrace (compile_program c) s g itr ->
        imp_diverges_with_inftrace c s g itr' -> EqSt itr itr'.
```

These theorems instantiate Theorem 9 for this compiler. Note that only the first theorem, `compile_program_correct_terminating` for the terminating case, allows for interruption. This is because if the machine diverges, it cannot have been interrupted. An additional wrinkle is the predicate `EqSt`, which defines equality on coinductive infinite traces.

### Backward simulation for deterministic semantics

```
Module Determ.
...
Theorem compile_program_correct_terminating_backward:
  forall c s g tr s' g',
    machine_terminates (compile_program c) s g tr s' g' ->
    imp_terminates c s g tr s' g'.

Theorem compile_program_correct_diverging_silently_backward:
  forall c s g tr,
    machine_diverges_silently (compile_program c) s g tr ->
    imp_diverges_silently c s g tr.

Theorem compile_program_correct_diverging_with_inftrace_backward:
  forall c s g itr itr',
    machine_diverges_with_inftrace (compile_program c) s g itr ->
    imp_diverges_with_inftrace c s g itr' -> EqSt itr itr'.
...
End Determ.
```

These theorems instantiate Lemma 1 for this compiler.

### Modified backward simulation from full to deterministic semantics

```
Theorem prefix_correct_full_to_determ_semantics:
    forall C s g tr,
        machine_terminates C s g tr ->
        (exists s' g', Determ.machine_terminates C s g tr s' g')
        \/ (exists m, tr = m ++ [ev_intr]
            /\ Determ.machine_admits_finite C s g m)
        \/ (Determ.machine_goes_wrong C s g tr).

Lemma divergence_implies_determ_divergence_silent:
    forall C s g tr,
        machine_diverges_silently C s g tr ->
        Determ.machine_diverges_silently C s g tr.

Lemma divergence_implies_determ_divergence_inftrace:
    forall C s g itr,
        machine_diverges_with_inftrace C s g itr ->
        Determ.machine_diverges_with_inftrace C s g itr.
```

These theorems instantiate Lemma 2 for this compiler.

# References


[1] Abate, C., Blanco, R., Ciobâcă, Ş., Durier, A., Garg, D., Hriţcu, C., Patrignani, M., Tanter, É., & Thibault, J. (2020). Trace-Relating Compiler Correctness and Secure Compilation. In P. Müller (Ed.), *Programming Languages and Systems*: Springer.

[2] Abate, C., Blanco, R., Garg, D., Hritcu, C., Patrignani, M., & Thibault, J. (2019). Journey Beyond Full Abstraction: Exploring Robust Property Preservation for Secure Compilation. In *IEEE 32nd Computer Security Foundations Symposium*.

[3] Agda Development Team (2021). *The Agda Reference Manual, version 2.6.2*.

[4] Anand, A., Appel, A. W., Morrisett, G., Paraskevopoulou, Z., Pollack, R., Bélanger, O. S., Sozeau, M., & Weaver, M. Z. (2016). CertiCoq : A verified compiler for Coq.

[5] Appel, A. W. (2015). Verification of a cryptographic primitive: SHA-256. *ACM Trans. Program. Lang. Syst.*, 37(2).

[6] Barthe, G., Blazy, S., Grégoire, B., Hutin, R., Laporte, V., Pichardie, D., & Trieu, A. (2019). Formal Verification of a Constant-Time Preserving C Compiler. *Proc. ACM Program. Lang.*, 4(POPL).

[7] Beringer, L., Petcher, A., Ye, K. Q., & Appel, A. W. (2015). Verified correctness and security of OpenSSL HMAC. In *24th USENIX Security Symposium (USENIX Security 15)* (pp. 207–221). Washington, D.C.: USENIX Association.

[8] Beringer, L., Stewart, G., Dockins, R., & Appel, A. W. (2014). Verified Compilation for Shared-Memory C. In Z. Shao (Ed.), *Programming Languages and Systems* (pp. 107–127). Berlin, Heidelberg: Springer Berlin Heidelberg.

[9] Boldo, S., Jourdan, J.-H., Leroy, X., & Melquiond, G. (2015). Verified Compilation of Floating-Point Computations. *J. Autom. Reason.*, 54(2), 135–163.

[10] Chen, H., Ziegler, D., Chajed, T., Chlipala, A., Kaashoek, M. F., & Zeldovich, N. (2016). Using Crash Hoare Logic for certifying the FSCQ file system. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)* Denver, CO: USENIX Association.

[11] Chlipala, A. (2013). *Certified Programming with Dependent Types: A Pragmatic Introduction to the Coq Proof Assistant*. The MIT Press. Cambridge: The MIT Press.

[12] Coq Development Team (2020). *The Coq Reference Manual, version 8.13*.

[13] Di Franco, A., Guo, H., & Rubio-González, C. (2017). A comprehensive study of real-world numerical bug characteristics. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)* (pp. 509–519).

[14] Erbsen, A., Philipoom, J., Gross, J., Sloan, R., & Chlipala, A. (2020). Simple high-level code for cryptographic arithmetic: With proofs, without compromises. *SIGOPS Oper. Syst. Rev.*, 54(1), 23–30.

[15] Férée, H., Åman Pohjola, J., Kumar, R., Owens, S., Myreen, M. O., & Ho, S. (2018). Program Verification in the Presence of I/O. In R. Piskac & P. Rümmer (Eds.), *Verified Software. Theories, Tools, and Experiments* (pp. 88–111).: Springer Intl. Pub.

[16] GNU Project (2021). *GCC, the GNU Compiler Collection*.

[17] Gómez-Londoño, A., Åman Pohjola, J., Syeda, H. T., Myreen, M. O., & Tan, Y. K. (2020). Do You Have Space for Dessert? A Verified Space Cost Semantics for CakeML Programs. *Proc. ACM Program. Lang.*, 4(OOPSLA).

[18] Gosling, J., Joy, B., Steele, G., Bracha, G., Buckley, A., Smith, D., & Bierman, G. (2021). The Java Language Specification: Java SE 16 Edition.

[19] Gu, R., Shao, Z., Chen, H., Wu, X. N., Kim, J., Sjöberg, V., & Costanzo, D. (2016). Certikos: An extensible architecture for building certified concurrent OS kernels. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)* (pp. 653–669). Savannah, GA: USENIX Association.

[20] IEEE (2019). IEEE Standard for Floating-Point Arithmetic. *IEEE Std 754-2019 (Revision of IEEE 754-2008)*, (pp. 1–84).

[21] IEEE Std 1003.1 2018 Edition (2018). The Open Group Base Specifications Issue 2. System Interfaces.

[22] Isabelle/HOL Development Team (2021). *The Isabelle/HOL Reference Manual, version Isabelle2021*.

[23] Kang, J., Kim, Y., Hur, C.-K., Dreyer, D., & Vafeiadis, V. (2016). Lightweight Verification of Separate Compilation. *SIGPLAN Not.*, 51(1), 178–190.

[24] Kästner, D., Barrho, J., Wünsche, U., Schlickling, M., Schommer, B., Schmidt, M., Ferdinand, C., Leroy, X., & Blazy, S. (2018). CompCert: Practical Experience on Integrating and Qualifying a Formally Verified Optimizing Compiler. In *9th European Congress Embedded Real-Time Software and Systems* (pp. 1–9).

[25] Kumar, R., Myreen, M. O., Norrish, M., & Owens, S. (2014). CakeML: A verified implementation of ML. In *Principles of Programming Languages (POPL)* (pp. 179–191).: ACM Press.

[26] Lean Development Team (2021). *The Lean Reference Manual, version 4*.

[27] Leroy, X. (2009a). A Formally Verified Compiler Back-End. *J. Autom. Reason.*, 43(4), 363–446.

[28] Leroy, X. (2009b). Formal verification of a realistic compiler. *Communications of the ACM*, 52(7), 107–115.

[29] Leroy, X. (2019). The formal verification of compilers (2019 EUTypes Summer School).

[30] Leroy, X., Appel, A. W., Blazy, S., & Stewart, G. (2012). *The CompCert Memory Model, Version 2*. Research report RR-7987, INRIA.

[31] McCarthy, J. & Painter, J. (1967). Correctness of a compiler for arithmetic expressions.

[32] Milner, R. & Weyhrauch, R. (1972). Proving compiler correctness in a mechanised logic. *Machine Intelligence*, 7, 51–73.

[33] MITRE Corporation (2020). *Common Weakness Enumeration: Top 25 Most Dangerous Softwaree Weaknesses*. Technical report.

[34] Muller, S. & Hoffman, J. (2020). Combining Source and Target Level Cost Analyses for OCaml Programs. *Working paper*.

[35] Patrignani, M., Ahmed, A., & Clarke, D. (2019). Formal approaches to secure compilation: A survey of fully abstract compilation and related work. *ACM Comput. Surv.*, 51(6).

[36] Patterson, D. & Ahmed, A. (2019). The next 700 Compiler Correctness Theorems (Functional Pearl). *Proc. ACM Program. Lang.*, 3(ICFP).

[37] Pierce, B. C. (2019). *Types and Programming Languages*. The MIT Press. The MIT Press.

[38] Pierce, B. C., Azevedo de Amorim, A., Casinghino, C., Gaboardi, M., Greenberg, M., Hriţcu, C., Sjöberg, V., & Yorgey, B. (2018). *Logical Foundations*. Software Foundations series, volume 1. Electronic textbook.

[39] Python Software Foundation (2021). *Python 3.9.5 Documentation*.

[40] Sangiorgi, D. (2011). *Introduction to Bisimulation and Coinduction*. Cambridge: Cambridge University Press.

[41] Song, Y., Cho, M., Kim, D., Kim, Y., Kang, J., & Hur, C.-K. (2019). CompCertM: CompCert with C-Assembly Linking and Lightweight Modular Verification. *Proc. ACM Program. Lang.*, 4(POPL).

[42] Stewart, G., Beringer, L., Cuellar, S., & Appel, A. W. (2015). Compositional CompCert. *SIGPLAN Not.*, 50(1), 275–287.

[43] Tan, Y. K., Myreen, M. O., Kumar, R., Fox, A., Owens, S., & Norrish, M. (2016). A New Verified Compiler Backend for CakeML. *SIGPLAN Not.*, 51(9), 60–73.

[44] Titolo, L., Feliú, M. A., Moscato, M., & Muñoz, C. A. (2018). An Abstract Interpretation Framework for the Round-Off Error Analysis of Floating-Point Programs. In I. Dillig & J. Palsberg (Eds.), *Verification, Model Checking, and Abstract Interpretation* (pp. 516–537).: Springer International Publishing.

[45] Yang, X., Chen, Y., Eide, E., & Regehr, J. (2011). Finding and Understanding Bugs in C Compilers. *SIGPLAN Not.*, 46(6), 283–294.

This thesis was typeset using LaTeX, originally developed by Leslie Lamport and based on Donald Knuth's TeX. The body text is set in 11 point Egenolff-Berner Garamond, a revival of Claude Garamont's humanist typeface. A template that can be used to format a PhD dissertation with this look and feel has been released under the permissive AGPL license, and can be found online at github.com/suchow/Dissertate or from its lead author, Jordan Suchow, at suchow@post.harvard.edu.