Syntactic Proofs of Compositional Compiler Correctness

Adam Chlipala

Harvard University
Cambridge, Massachusetts, USA

Abstract. Semantic preservation by compilers for higher-order languages can be verified using simple syntactic methods. At the heart of classic techniques are relations between source-level and target-level values. Unfortunately, these relations are specific to particular compilers, leading to correctness theorems that have nothing to say about linking programs with functions compiled by other compilers or written by hand in the target language. Theorems based on logical relations manage to avoid this problem, but at a cost: standard logical relations do not apply directly to programs with non-termination or impurity, and extensions to handle those features are relatively complicated, compared to the classical compiler verification literature.

In this paper, we present a new approach to "open" compiler correctness theorems that is "syntactic" in the sense that the core relations do not refer to semantics. Though the technique is much more elementary than previous proposals, it scales up nicely to realistic languages. In particular, untyped and impure programs may be handled simply, while previous work has addressed neither in this context.

Our approach is based on the observation that it is an unnecessary handicap to consider proofs as black boxes. We identify some theorem-specific proof skeletons, such that we can define an algebra of nondeterministic compilations and their proofs, and we can compose any two compilations to produce a correct-by-construction result. We have prototyped these ideas with a Coq implementation of multiple CPS translations for an untyped Mini-ML source language with recursive functions, sums, products, mutable references, and exceptions.

1 Introduction

Compiler verification is a long-standing area of interest within computer science. Because it is relatively straightforward to give precise semantics to programming languages, we can state clear theorems characterizing when a compiler is correct. Such a compiler preserves the observable behavior of programs. Verification holds the promise of decreasing the cost of building compilers that operate correctly most of the time.

Mechanized compiler verification, where a computer checks the proofs, was first studied seriously by Moore (1989), who used the Boyer-Moore prover to

verify an entire programming language stack, culminating in a first-order language called Piton. However, only in the last 10 years has the research area really taken off, probably due to a mixture of increases in hardware capacity and improvements to theorem-proving algorithms and implementations. The CompCert verified C compiler project (Leroy 2006) is one of the best-known among the recent projects.

There have also been many studies of compiler verification for functional languages (Minamide and Okuma 2003; Tian 2006; Chlipala 2007; Dargaye and Leroy 2007; Chlipala 2008; Benton and Hur 2009; Chlipala 2010). At the core of each of these proofs is some relation between intermediate values occurring in program execution. By ensuring that source and compiled programs maintain related values throughout execution, one can prove semantic preservation inductively. The different proofs may be be divided usefully into the broad categories of "syntactic" and "semantic" proofs.

Syntactic proofs use value relations that do not mention the semantics of programs. Instead, a source function f is said to be related to a target function g if g is the result of compiling f with this particular compiler. The syntactic approach leads to some very straightforward proofs that, for simple object languages, are tractable to write out in full detail as natural deduction proof trees. A serious downside, however, is that a correctness theorem of this kind has nothing to say about linking with target language functions produced by a different compiler or written by hand. Essentially no real programs run only code produced by a single compiler, so this downside is more than just theoretical.

Semantic proofs use variants of the classic technique of logical relations (Plotkin 1973), which defines binary relations by recursion on object language type structure. Some of our past work for strongly-normalizing languages (Chlipala 2007, 2008) applied this standard technology. Benton and Hur (2009) consider more ambitious verification problems, applying step-indexed logical relations (Ahmed 2006) and $\top\top$-closure (Pitts and Stark 1998) to treat an object language with general-recursive functions.

The approach of Benton and Hur adds significant complexity beyond that of syntactic techniques. Step-indexed logical relations break cycles in relation definitions by introducing explicit accounting of how many steps programs run for, and $\top\top$-closure involves explicit reasoning about evaluation contexts. Classic syntactic proofs avoid both dimensions of complexity by using simple induction over evaluation derivations; details that semantic proofs incorporate in their logical relations are instead present implicitly in the inductive structure of proofs.

The popularity of the syntactic approach to type soundness (Wright and Felleisen 1994) provides evidence that most languages researchers prefer the "syntactic" style over the "semantic" style. Can this preference be reconciled with the need for compositional theorems? The history of step-indexed logical relations provides a hint at the answer. The idea originated in the work on foundational proof-carrying code (Appel and McAllester 2001), where it formed a semantic counterpart to a parallel line of work on syntactic techniques (Hamid

et al. 2003). The foundations of these syntactic techniques can be adapted naturally to the problem of compiler verification.

In this paper, we propose a new technique that combines some of the benefits of syntactic and semantic proofs. The technique handles compilation of untyped and impure languages, which had not previously been verified in compositional ways. We can verify individual compilers in isolation and later use their correctness theorems abstractly to reason about the results of linking their output programs. We can hand-craft target-level programs and reason abstractly about linking them with the outputs of any verified compilers.

In the next section, we review classical syntactic techniques for verifying CPS translations for functional languages, finishing with an explanation of why these techniques are not sufficiently compositional. The next section introduces a relational framework for more compositional proofs. We show how to prove properties of programs built with multiple compilers and with some hand-crafted code, relying only on a common set of requirements on translations. Our first framework is not sufficient for reasoning about compiling most higher-order functions, which motivates an extension, presented in the following section. We then describe how this framework can be adapted to a language with most of the key dynamic features of core ML, and we summarize how we implemented the framework and the example theorems in Coq.

As our theorems are mechanized in Coq, we will focus in this paper on presenting the main ideas in an understandable way. We ignore issues of term well-formedness and omit some side conditions, and the proofs we give are not very detailed. The Coq development can be consulted for the rigorous details.

2 Syntactic Proofs of Semantic Preservation

Throughout this article, our running example will be translation to continuation-passing style (CPS), which consumes normal programs in lambda calculi and produces programs where functions never return. Instead, functions are rewritten to take explicit "return pointers" as new arguments. We further restrict the target languages to enforce that computations are broken down into sequences of primitive operations.

We start with the basic untyped lambda calculus with boolean constants. Booleans form a simple, non-trivial domain of values whose representation can be preserved by CPS translation, allowing us to phrase our final correctness theorem in terms of preservation of boolean results.

$$\begin{aligned} \text{Boolean constants } b &::= \top \mid \bot \\ \text{Variables } x & \\ \text{Expressions } e &::= b \mid x \mid e\ e \mid \lambda x.\ e \end{aligned}$$

A simple call-by-value, big-step operational semantics explains the meanings of closed expressions. We choose an environment semantics to simplify the proofs that will follow. Judgments have the form $\sigma; e \Downarrow v$, indicating that expression e

evaluates to result v in environment σ. An environment is a partial map from variables to values, and we write $\sigma\{x \mapsto v\}$ for the extension of σ with a mapping from x to v.

$$\text{Values } v ::= b \mid \langle \sigma, \lambda x.\ e \rangle$$

$$\frac{}{\sigma; x \Downarrow \sigma(x)} \quad \frac{}{\sigma; b \Downarrow b} \quad \frac{}{\sigma; \lambda x.\ e \Downarrow \langle \sigma, \lambda x.\ e \rangle}$$

$$\frac{\sigma; e_1 \Downarrow \langle \sigma', \lambda x.\ e \rangle \quad \sigma; e_2 \Downarrow v \quad \sigma'\{x \mapsto v\}; e \Downarrow v'}{\sigma; e_1\ e_2 \Downarrow v'}$$

The CPS version of this language forces explicit sequencing and forces all function calls to appear in tail positions. We add pairs to the language, because they are useful in implementing CPS translations.

$$\begin{array}{c} \text{Primops } p ::= b \mid \lambda x.\ e \mid (x, x) \mid x.1 \mid x.2 \\ \text{Expressions } e ::= \mathsf{halt}(x) \mid x\ x \mid \mathsf{let}\ x = p\ \mathsf{in}\ e \\ \text{Values } v ::= b \mid \langle \sigma, \lambda x.\ e \rangle \mid (v, v) \end{array}$$

The target language has its own big-step semantics.

$$\frac{}{\sigma; b \Downarrow b} \quad \frac{}{\sigma; \lambda x.\ e \Downarrow \langle \sigma, \lambda x.\ e \rangle} \quad \frac{}{\sigma; (x_1, x_2) \Downarrow (\sigma(x_1), \sigma(x_2))} \quad \frac{\sigma(x) = (v_1, v_2)}{\sigma; x.1 \Downarrow v_1}$$

$$\frac{\sigma(x) = (v_1, v_2)}{\sigma; x.2 \Downarrow v_2} \quad \frac{}{\sigma; \mathsf{halt}(x) \Downarrow \sigma(x)}$$

$$\frac{\sigma(x_1) = \langle \sigma', \lambda x.\ e \rangle \quad \sigma'\{x \mapsto \sigma(x_2)\}; e \Downarrow v}{\sigma; x_1\ x_2 \Downarrow v} \quad \frac{\sigma; p \Downarrow v \quad \sigma\{x \mapsto v\}; e \Downarrow v'}{\sigma; \mathsf{let}\ x = p\ \mathsf{in}\ e \Downarrow v'}$$

We use a higher-order, one-pass CPS translation, based on that of Danvy and Filinski (1992). Where var is the type of variables, sexp the type of source expressions, and cexp the type of target expressions, the translation $\lfloor \cdot \rfloor$ has type $\mathsf{sexp} \to (\mathsf{var} \to \mathsf{cexp}) \to \mathsf{cexp}$. The second argument is a "continuation" that describes what to do with the result of executing the first argument's compilation; this result should be placed in a variable, and that variable should be passed as the continuation's argument. We will write $\hat{\lambda}$ for the meta language's function abstraction, and we write meta language function application of f to x as $f(x)$, to distinguish from an object language application $f\ x$.

$$\begin{aligned} \lfloor b \rfloor &= \hat{\lambda} k.\ \mathsf{let}\ x = b\ \mathsf{in}\ k(x) \\ \lfloor x \rfloor &= \hat{\lambda} k.\ k(x) \\ \lfloor \lambda x.\ e \rfloor &= \hat{\lambda} k.\ \mathsf{let}\ f = (\lambda p.\ \mathsf{let}\ x = p.1\ \mathsf{in}\ \mathsf{let}\ k' = p.2\ \mathsf{in}\ \lfloor e \rfloor\ (\hat{\lambda} r.\ k'\ r))\ \mathsf{in}\ k(f) \\ \lfloor e_1\ e_2 \rfloor &= \hat{\lambda} k.\ \lfloor e_1 \rfloor\ (\hat{\lambda} f.\ \lfloor e_2 \rfloor\ (\hat{\lambda} x.\ \mathsf{let}\ k' = (\lambda r.\ k(r))\ \mathsf{in}\ \mathsf{let}\ p = (x, k')\ \mathsf{in}\ f\ p)) \end{aligned}$$

Every function is transformed to a take a pair as an argument. The first field of the pair is the original argument, while the second field is the return continuation. In the translation of a source-level λ, the function body is passed a continuation built by lifting an object language continuation variable k' into a

meta language continuation $\hat{\lambda} r.\ k'\ r$. At each call site, the current continuation k is reified into an object language function k' by wrapping it in an object language abstraction $\lambda r.\ k(r)$.

Finally, we can define the overall program translation $\lfloor \cdot \rfloor^{\mathcal{P}}$ by choosing an appropriate initial continuation. In particular, we define $\lfloor e \rfloor^{\mathcal{P}} = \lfloor e \rfloor\ (\hat{\lambda} x.\ \mathsf{halt}(x))$.

We would like to prove that this translation preserves the observable behavior of programs. For simplicity, let us consider terminating programs only.

Theorem 1 (Semantic preservation). $\forall e, b.\ \cdot; e \Downarrow b \Rightarrow \cdot; \lfloor e \rfloor^{\mathcal{P}} \Downarrow b$

To prove Theorem 1, we will need to come up with a stronger induction hypothesis. To do so, it turns out to be useful to define a CPS translation for values. We overload the notation $\lfloor \cdot \rfloor$ for this translation, since context will make it clear when we are referring to the expression or value translation.

$$\lfloor b \rfloor = b$$
$$\lfloor \langle \sigma, \lambda x.\ e \rangle \rfloor = \left\langle \lfloor \sigma \rfloor, \lambda p.\ \mathsf{let}\ x = p.1\ \mathsf{in}\ \mathsf{let}\ k' = p.2\ \mathsf{in}\ \lfloor e \rfloor\ (\hat{\lambda} r.\ k'\ r) \right\rangle$$

We write $\lfloor \sigma \rfloor$ for the pointwise application of value compilation to a source-level substitution.

Now we can state and prove the main inductive lemma.

Lemma 1. $\forall \sigma, e, v.\ \sigma; e \Downarrow v \Rightarrow \forall k, v'.\ (\forall x, \sigma' \supseteq \lfloor \sigma \rfloor\ .\ \sigma'(x) = \lfloor v \rfloor \Rightarrow \sigma'; k(x) \Downarrow v') \Rightarrow \lfloor \sigma \rfloor; \lfloor e \rfloor\ k \Downarrow v'$

Proof. By induction on the derivation of $\sigma; e \Downarrow v$. □

Theorem 1 follows as an easy corollary. We take k to be $\hat{\lambda} x.\ \mathsf{halt}(x)$, which makes $\sigma'; k(x) \Downarrow v'$ equivalent to $v' = \lfloor v \rfloor$.

2.1 Optimizing Tail Calls

This simple CPS translation will sometimes build a new continuation when an old continuation could have been reused. This happens in tail positions, where the last thing that a function does is call another function. We have a continuation variable k in scope that could be passed in the final function call, but we instead define a new function $\lambda r.\ k\ r$, the eta-expansion of k.

A simple modification to the algorithm suffices to optimize tail calls. We define a type cont of continuations, whose values are variables $\mathsf{Var}(x)$ and meta-level functions $\mathsf{Fun}(f)$, with f of type $\mathsf{var} \to \mathsf{cexp}$. A binary operator $\diamond$ applies a continuation to a variable.

$$\mathsf{Var}(k) \diamond x = k\ x$$
$$\mathsf{Fun}(f) \diamond x = f(x)$$

We also define a notation $\mathsf{let}^{\mathsf{K}}\ x = K\ \mathsf{in}\ e$ for local binding of a continuation.

$$(\mathsf{let}^{\mathsf{K}}\ x = \mathsf{Var}(k)\ \mathsf{in}\ e) = e\{x \mapsto k\}$$
$$(\mathsf{let}^{\mathsf{K}}\ x = \mathsf{Fun}(f)\ \mathsf{in}\ e) = \mathsf{let}\ x = (\lambda y.\ f(y))\ \mathsf{in}\ e$$

Our revised translation $\lfloor\cdot\rfloor_{\mathcal{T}}$ has type $\mathsf{sexp} \to \mathsf{cont} \to \mathsf{cexp}$. The basic logic is the same as before, but we use the new continuation type to detect tail calls.

$$\lfloor b\rfloor_{\mathcal{T}} = \hat{\lambda}k.\ \mathsf{let}\ x = b\ \mathsf{in}\ k \diamond x$$
$$\lfloor x\rfloor_{\mathcal{T}} = \hat{\lambda}k.\ k \diamond x$$
$$\lfloor \lambda x.\ e\rfloor_{\mathcal{T}} = \hat{\lambda}k.\ \mathsf{let}\ f = (\lambda p.\ \mathsf{let}\ x = p.1\ \mathsf{in}\ \mathsf{let}\ k' = p.2\ \mathsf{in}\ \lfloor e\rfloor_{\mathcal{T}}\ (\mathsf{Var}(k')))\ \mathsf{in}\ k \diamond f$$
$$\lfloor e_1\ e_2\rfloor_{\mathcal{T}} = \hat{\lambda}k.\ \lfloor e_1\rfloor_{\mathcal{T}}\ (\mathsf{Func}(\hat{\lambda}f.\ \lfloor e_2\rfloor_{\mathcal{T}}\ (\mathsf{Func}(\hat{\lambda}x.\ \mathsf{let}^{\mathsf{K}}\ k' = k\ \mathsf{in}\ \mathsf{let}\ p = (x, k')\ \mathsf{in}\ f\ p))))$$

The overall program translation is $\lfloor e\rfloor_{\mathcal{T}}^{\mathcal{P}} = \lfloor e\rfloor_{\mathcal{T}}\ (\mathsf{Fun}(\hat{\lambda}x.\ \mathsf{halt}(x)))$.

We can prove a correctness theorem for the new translation.

Theorem 2 (Semantic preservation). $\forall e, b.\ \cdot; e \Downarrow b \Rightarrow \cdot; \lfloor e\rfloor_{\mathcal{T}}^{\mathcal{P}} \Downarrow b$

Again we need a lemma expressing a stronger induction hypothesis, and again we define value compilation to allow us to state the lemma.

$$\lfloor b\rfloor_{\mathcal{T}} = b$$
$$\lfloor\langle \sigma, \lambda x.\ e\rangle\rfloor_{\mathcal{T}} = \langle \lfloor\sigma\rfloor_{\mathcal{T}}, \lambda p.\ \mathsf{let}\ x = p.1\ \mathsf{in}\ \mathsf{let}\ k' = p.2\ \mathsf{in}\ \lfloor e\rfloor_{\mathcal{T}}\ (\mathsf{Var}(k'))\rangle$$

Lemma 2. $\forall \sigma, e, v.\ \sigma; e \Downarrow v \Rightarrow \forall k, v'.\ (\forall x, \sigma' \supseteq \lfloor\sigma\rfloor_{\mathcal{T}}.\ \sigma'(x) = \lfloor v\rfloor_{\mathcal{T}} \Rightarrow \sigma'; k \diamond x \Downarrow v') \Rightarrow \lfloor\sigma\rfloor_{\mathcal{T}}; \lfloor e\rfloor_{\mathcal{T}}\ k \Downarrow v'$

Proof. By induction on the derivation of $\sigma; e \Downarrow v$. □

2.2 The Problem

Our two translations make different optimization choices, but they follow the same calling convention. Thus, it is reasonable to expect that we can pick and choose parts of a source program to compile with each translation, and the resulting program will still behave correctly. The following theorem embodies one case of our expectations. We use the source-level abbreviation $\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2$ for $(\lambda x.\ e_2)\ e_1$.

Theorem 3 (Linking).

$$\forall e_1, e_2, b.\ \cdot; (\mathsf{let}\ f = \lambda x.\ e_1\ \mathsf{in}\ e_2) \Downarrow b \Rightarrow \cdot; (\mathsf{let}\ f = \lfloor \lambda x.\ e_1\rfloor\ \mathsf{in}\ \lfloor e_2\rfloor_{\mathcal{T}}^{\mathcal{P}}) \Downarrow b$$

We have a program e_2 that uses a function with body e_1. Perhaps e_1 is from a library that was compiled using the old, non-optimizing compiler, and perhaps we do not want to have to recompile that library. Nonetheless, we want to compile e_2 using the new compiler that optimizes tail calls. Theorem 3 asserts that this kind of linking is always sound, up to boolean results.

Theorem 3 is true, and we will prove it in a later section. We might be tempted to prove the theorem with a method specialized to the combination of our two translations, but that approach does not scale well. What happens when the optimizing compiler evolves still further, and we still want to link with the same legacy code? What happens if we want to link with legacy code produced by multiple old compiler versions, or if we want to link with a library that contains hand-written CPS code? We must prove a linking theorem for each combination. The process is very unmodular; the programmer assembling code that depends on a set of compilers must understand all of the compilers and their interactions, if he is to prove his linking theorem.

We could restate our semantic preservation theorems using step-indexed logical relations. This would let us apply those theorems *abstractly* to prove linking theorems without revisiting any of the details of our compilation strategies. In the introduction, we explained why such semantic methods introduce a nontrivial layer of new complexity. The remainder of this paper is dedicated to presenting a new technique that lets us follow the proof outlines from this section while gaining the chance to compose these theorems abstractly. We can write straightforward proofs of compiler correctness, such that our compiler's outputs can later be composed soundly with outputs of other compilers that we knew nothing about.

3 Compilation Relations

The idea of step-indexed logical relations (Appel and McAllester 2001) originated in the context of foundational proof-carrying code (FPCC) (Appel 2001). In that setting, the challenge was to give denotational semantics to a type system expressive enough to apply to compiled ML programs. Step indices helped break troubling cycles in the definitions of the meanings of types.

Concurrently to the development of the semantic approach to FPCC, a competing syntactic approach (Hamid et al. 2003) was also in progress. While the semantic approach compiled types into relations over machine values, the syntactic approach followed the more common style of applying syntactic typing rules to assembly programs. Both approaches make it possible to believe that an assembly program satisfies a safety policy, without needing to trust in the soundness of any type system.

The two lines of work have their benefits and drawbacks. The semantic approach made it easier to integrate new types into an existing proof, since, in that setting, a type is just a particular kind of relation over machine values. On the other hand, in the syntactic setting, coming up with a proof in the first place was arguably easier by a significant margin, since proofs proceeded by elementary techniques familiar to scholars of type systems.

More recently, the over-specificity of syntactic FPCC implementations has been addressed using the family of program logics called Certified Assembly Programming (CAP) (Hamid and Shao 2004). Several different Hoare logics applicable to assembly programs have been generalized into OCAP (Feng et al.

2007), a logic parameterized by sets of specification languages, such that derivations apply safely to broader classes of specifications than those they were implemented with.

The work we will present here was born from pondering the analogies between FPCC and compiler verification. Both have evolved distinct syntactic and semantic approaches, with similar pros and cons. Ideas from semantic FPCC have previously been adapted towards enabling compositional compiler verification. What follows is the result of our attempt to adapt the ideas of syntactic FPCC and CAP to the same setting.

In FPCC and general verification of low-level programs, the main objects of interest are specifications which are relations over single values, machine states, blocks of code, etc.. In compiler verification, we care about relations over pairs of values, states, programs, etc.. Such relations connect source and compiled versions of programs.

Each CAP derivation is parameterized by a set of specifications for a subset of the possible code labels. Rules of the logics make it possible to link together compatible specification sets and their derivations. Similarly, the theorems to follow will be phrased in terms of particular compilers, represented as relations between programs. We will prove that it is sound to combine these results via unions of compilation relations.

3.1 Compilers as Relations

Our approach must be specialized to particular translation tasks. For the rest of the paper, we stick with the example of CPS translation, though the techniques generalize to other cases.

What follows is the first cut at a characterization of a compiler that is correct in a composable way.

Definition 1. *A* compilation *is a pair* $(R_{\mathsf{E}}, R_{\mathsf{F}})$ *such that:*

- R_{E} *is a relation over* $\mathsf{sexp} \times (\mathsf{var} \to \mathsf{cexp}) \times \mathsf{cexp}$.
- R_{F} *is a relation over* $\mathsf{sexp} \times \mathsf{cexp}$*, where both expressions are considered as function bodies with argument variable* x*. (We alpha-vary expressions implicitly as needed to maintain this property.)*
- *For every* $(f_1, f_2) \in R_{\mathsf{F}}$*, there exist expression* f_2' *and variables* p *and* k *such that:*
  - $(f_1, (\hat{\lambda} x.\ k\ x), f_2') \in R_{\mathsf{E}}$
  - $\forall \sigma, v, F, r.\ (\forall \sigma' \supseteq \sigma\{x \mapsto v\}\{k \mapsto F\}.\ \sigma'; f_2' \Downarrow r) \Rightarrow \sigma\{p \mapsto (v, F)\}; f_2 \Downarrow r$*, where we implicitly alpha-convert* f_2 *to call its argument variable* p

For compilation $(R_{\mathsf{E}}, R_{\mathsf{F}})$, the relation R_{E} is a nondeterministic compiler for expressions, while R_{F} is a nondeterministic compiler for function bodies.

The third condition of Definition 1 expresses the *calling convention*. The condition essentially says that, if f_1 can be compiled to f_2, then there exists some f_2' that represents the part of f_2 after a function preamble. We need to

know that composing a standard preamble with f_2' leads to code that behaves operationally like an appropriate call to f_2, following the calling convention of passing a pair of the primary argument and a continuation.

Note that both relations within a compilation quantify over potentially *open* expressions. The free variables of such expressions may end up being filled with values produced by different compilers. Separating expressions from substitutions σ helps us orchestrate this sharing of responsibility.

Our example CPS translations can be phrased as compilations.

Definition 2. *Define* $\mathcal{C} = (\mathcal{C}_\mathsf{E}, \mathcal{C}_\mathsf{F})$ *as:*

- $\mathcal{C}_\mathsf{E} = \{(e_1, k, e_2) \mid e_2 = \lfloor e_1 \rfloor\, k\}$
- $\mathcal{C}_\mathsf{F} = \{(f_1, f_2) \mid (\lambda x.\ f_2) = \lfloor \lambda x.\ f_1 \rfloor\}$

From the operational semantics of the CPS language, it is easy to verify that $\mathcal{C}$ satisfies the conditions for a compilation.

The tail-call-optimizing compiler can also be expressed as a compilation.

Definition 3. *Define* $\mathcal{C}^\mathcal{T} = (\mathcal{C}^\mathcal{T}_\mathsf{E}, \mathcal{C}^\mathcal{T}_\mathsf{F})$ *as:*

- $\mathcal{C}^\mathcal{T}_\mathsf{E} = \{(e_1, k, e_2) \mid \exists k'.\ e_2 = \lfloor e_1 \rfloor_\mathcal{T}\, k' \wedge k = (\hat{\lambda} x.\ k' \diamond x)\}$
- $\mathcal{C}^\mathcal{T}_\mathsf{F} = \{(f_1, f_2) \mid (\lambda x.\ f_2) = \lfloor \lambda x.\ f_1 \rfloor_\mathcal{T}\}$

3.2 Compilation Correctness

We can define a relation between source- and target-level values, parameterized by a compilation.

$$\frac{}{C \vdash b \sim b} \qquad \frac{(f_1, f_2) \in C_\mathsf{F} \quad \forall x \in \mathsf{dom}(\sigma_1).\ C \vdash \sigma_1(x) \sim \sigma_2(x)}{C \vdash \langle \sigma_1, \lambda x.\ f_1 \rangle \sim \langle \sigma_2, \lambda x.\ f_2 \rangle}$$

A boolean value must be compiled to itself, but our relation gives some leeway in the compilation of functions. A function $\lambda x.\ f_1$ must be compiled to some function $\lambda x.\ f_2$. However, we permit any translation included in C_F, the compilation's relation for function bodies. This involves a choice not only of f_2 but also of a target-level substitution σ_2, which in effect expresses how each free variable of f_1 has been compiled. We check the compatibility of σ_1 and σ_2 by requiring that their mappings respect the $\sim$ relation. With this relation, we can give a quite general definition of what it means for a compilation to handle a specific program correctly.

Definition 4 (local compilation correctness). *For compilations A (mnemonic for "all compilers used to build a particular final program") and S (mnemonic for "self"), and for source-level substitution σ, expression e, and value v, define $A \vdash \sigma; e \Downarrow v \mathbin{@} S$ to mean that the following condition holds for every substitution σ', continuation k, and target-level expression e':*

- ***If**:*

  - $(e, k, e') \in S_{\mathsf{E}}$
  - $\forall x \in \mathsf{dom}(\sigma).\ A \vdash \sigma(x) \sim \sigma'(x)$
- ***Then*** *there exists target-level value* v' *such that:*
  - $A \vdash v \sim v'$
  - $\forall r.\ (\forall x, \sigma'' \supseteq \sigma'.\sigma''(x) = v' \Rightarrow \sigma''; k(x) \Downarrow r) \Rightarrow \sigma'; e' \Downarrow r$

Informally, the desired correctness condition for compilation S is that, for any suitable compilation A and source expression e evaluating to v in σ, we have $A \vdash \sigma; e \Downarrow v \mathbin{@} S$. We can give a formal characterization of when A is "suitable."

Definition 5 (compilation inclusion). *Say* $C \subseteq C'$ *if and only if* $C_{\mathsf{E}} \subseteq C'_{\mathsf{E}}$ *and* $C_{\mathsf{F}} \subseteq C'_{\mathsf{F}}$.

We are almost ready to state this section's final correctness property for compilations. To do so in a way that enables later composition, we choose a skeleton for proofs about CPS translations. This skeleton should be general enough to apply to translation strategies that we have not yet thought up. In this section, we fix a skeleton that enforces proof by simple structural induction over source-level evaluation derivations. Later sections consider alternate choices.

Definition 6 (compilation correctness). *A compilation* S *is* correct *if and only if the following conditions hold for any* A *such that* $S \subseteq A$*:*

- $\forall \sigma, x \in \mathsf{dom}(\sigma).\ A \vdash \sigma; x \Downarrow \sigma(x) \mathbin{@} S$
- $\forall \sigma, b.\ A \vdash \sigma; b \Downarrow b \mathbin{@} S$
- $\forall \sigma, e.\ A \vdash \sigma; \lambda x.\ e \Downarrow \langle \sigma, \lambda x.\ e \rangle \mathbin{@} S$
- $\forall \sigma, e_1, e_2, \sigma', e, v, v'.\ \sigma; e_1 \Downarrow \langle \sigma', \lambda x.\ e \rangle \wedge \sigma; e_2 \Downarrow v \wedge \sigma'\{x \mapsto v\}; e \Downarrow v'$
  $\wedge\ A \vdash \sigma; e_1 \Downarrow \langle \sigma', \lambda x.\ e \rangle \mathbin{@} A \wedge A \vdash \sigma; e_2 \Downarrow v \mathbin{@} A \wedge A \vdash \sigma'\{x \mapsto v\}; e \Downarrow v' \mathbin{@} A$
  $\Rightarrow A \vdash \sigma; e_1\ e_2 \Downarrow v' \mathbin{@} S$

The form of these conditions is determined directly by the induction principle that we have chosen, structural induction over evaluation derivations. The conditions are essentially the base and inductive cases required in the proofs of Lemmas 1 and 2. It is like we are manually stating the obligations to prove $\forall \sigma, e, v.\ \sigma; e \Downarrow v \Rightarrow A \vdash \sigma; e \Downarrow v \mathbin{@} S$, by induction on the derivation of $\sigma; e \Downarrow v$. Each case comes from a rule of the operational semantics, and each case has as premises not only the subderivations of the rule, but also an inductive hypothesis for each subderivation.

The final condition does not quite match the corresponding obligation within this kind of induction. The conclusion is of the form $A \vdash \cdot \mathbin{@} S$, indicating that we are only *obligated* to establish the condition relative to the behavior of the current compiler; while the inductive hypotheses are of the form $A \vdash \cdot \mathbin{@} A$, indicating that we are *permitted* to assume the theorem we are proving for *all* compilers, including the current compiler. Intuitively, each node of the big-step evaluation proof tree is tagged with the compiler responsible for that step. A step's subderivations may be the responsibilities of other compilers.

Theorem 4 (soundness). *For any correct compilation* C*, we have* $\forall \sigma, e, v.\ \sigma; e \Downarrow v \Rightarrow C \vdash \sigma; e \Downarrow v \mathbin{@} C$.

Proof. By induction on the derivation of $\sigma; e \Downarrow v$. The correctness obligations of C are precisely what we need to establish each inductive case, since we can show $C \subseteq C$ trivially. □

Theorem 5. *Compilations $\mathcal{C}$ and $\mathcal{C}^{\mathcal{T}}$ are correct.*

Proof. The proofs are nearly identical to those of Lemmas 1 and 2, respectively. □

In the previous section, we proved Theorems 1 and 2, characterizing the correctness of our two translations. These theorems now follow as easy corollaries of Theorems 4 and 5.

3.3 Linking

The next step in our agenda is to provide tools to reason about compilers cooperating to produce a single output program.

Definition 7 (compilation union). *For compilations C and C', define $C \cup C' = (C_{\mathsf{E}} \cup C'_{\mathsf{E}}, C_{\mathsf{F}} \cup C'_{\mathsf{F}})$. It is easy to verify that compilation unions are, in fact, compilations.*

Critically, compilation union preserves our correctness property.

Theorem 6. *For correct compilations C and C', their union $C \cup C'$ is also correct.*

Proof. We prove each obligation of Definition 6 with the same procedure. For each, we have that some $(e_1, k, e_2) \in (C \cup C')_{\mathsf{E}}$. By construction, this means $(e_1, k, e_2) \in C_{\mathsf{E}}$ or $(e_1, k, e_2) \in C'_{\mathsf{E}}$. Each of the cases follows by the corresponding correctness obligation of C or C'. □

Recall Theorem 3, which we put off proving until now:

$$\forall e_1, e_2, b.\ \cdot; (\mathsf{let}\ f = \lambda x.\ e_1\ \mathsf{in}\ e_2) \Downarrow b \Rightarrow \cdot; (\mathsf{let}\ f = \lfloor \lambda x.\ e_1 \rfloor\ \mathsf{in}\ \lfloor e_2 \rfloor^{\mathcal{P}}_{\mathcal{T}}) \Downarrow b$$

The machinery we have built up makes it possible to prove this theorem without revisiting the details of either CPS translation.

Proof. By the semantics of the source and target languages, the body of the theorem is equivalent to:

$$\cdot\{f \mapsto \langle \cdot, \lambda x.\ e_1 \rangle\}; e_2 \Downarrow b \Rightarrow \cdot\{f \mapsto \langle \cdot, \lfloor \lambda x.\ e_1 \rfloor \rangle\}; \lfloor e_2 \rfloor_{\mathcal{T}} (\mathsf{Fun}(\hat{\lambda} x.\ \mathsf{halt}(x))) \Downarrow b$$

By Theorem 6, we can deduce that $\mathcal{C} \cup \mathcal{C}^{\mathcal{T}}$ is correct, establishing the premise by Theorem 5. Thus, by Theorem 4, the hypothesis of our current theorem implies $\mathcal{C} \cup \mathcal{C}^{\mathcal{T}} \vdash \cdot\{f \mapsto \langle \cdot, \lambda x.\ e_1 \rangle\}; e_2 \Downarrow b\ @\ \mathcal{C} \cup \mathcal{C}^{\mathcal{T}}$.

We would like to use Definition 4 to deduce our final conclusion from this local correctness fact. Definition 4 includes two conditions that we must prove.

First, we need $(e_2, \hat{\lambda}x.\ \mathsf{halt}(x), \lfloor e_2 \rfloor_{\mathcal{T}}\,(\mathsf{Fun}(\hat{\lambda}x.\ \mathsf{halt}(x)))) \in (\mathcal{C} \cup \mathcal{C}^{\mathcal{T}})_{\mathsf{E}}$. This tuple is in $\mathcal{C}^{\mathcal{T}}_{\mathsf{E}}$ by construction, and it follows trivially that it must belong to the union as well.

Next, we need $\forall x \in \mathsf{dom}(\sigma_1).\ \mathcal{C} \cup \mathcal{C}^{\mathcal{T}} \vdash \sigma_1(x) \sim \sigma_2(x)$, for $\sigma_1 = \cdot\{f \mapsto \langle \cdot, \lambda x.\ e_1 \rangle\}$ and $\sigma_2 = \cdot\{f \mapsto \langle \cdot, \lfloor \lambda x.\ e_1 \rfloor \rangle\}$. This obligation reduces to $\mathcal{C} \cup \mathcal{C}^{\mathcal{T}} \vdash \langle \cdot, \lambda x.\ e_1 \rangle \sim \langle \cdot, \lfloor \lambda x.\ e_1 \rfloor \rangle$. The inference rule for compatibility of function values gives us two more obligations.

The first of these is $(e_1, e_1') \in (\mathcal{C} \cup \mathcal{C}^{\mathcal{T}})_{\mathsf{F}}$, where e_1' is the body of $\lfloor \lambda x.\ e_1 \rfloor$. The tuple belongs to $\mathcal{C}_{\mathsf{F}}$ by construction, and so it belongs to the union as well.

The second of the obligations is $\forall x \in \mathsf{dom}(\sigma_1).\ \mathcal{C} \cup \mathcal{C}^{\mathcal{T}} \vdash \sigma_1(x) \sim \sigma_2(x)$, where $\sigma_1 = \sigma_2 = \cdot$, so the obligation holds vacuously.

Thus, we have succeeded in applying Definition 4, so we know that there exists some v such that $\mathcal{C} \cup \mathcal{C}^{\mathcal{T}} \vdash b \sim v$ and

$$\begin{array}{l} \forall r, y \neq f.\ \cdot\{f \mapsto \langle \cdot, \lfloor \lambda x.\ e_1 \rfloor \rangle\}\{y \mapsto v\}; (\mathsf{Fun}(\hat{\lambda}x.\ \mathsf{halt}(x))) \diamond y \Downarrow r \\ \quad \Rightarrow\ \cdot\{f \mapsto \langle \cdot, \lfloor \lambda x.\ e_1 \rfloor \rangle\}; \lfloor e_2 \rfloor_{\mathcal{T}}\,(\mathsf{Fun}(\hat{\lambda}x.\ \mathsf{halt}(x))) \Downarrow r \end{array}$$

The antecedent of this implication reduces to $r = v$, so, by instantiating the quantifier with that equation, we get the final evaluation fact that we are looking for, but with v instead of b. We conclude by inversion on $\mathcal{C} \cup \mathcal{C}^{\mathcal{T}} \vdash b \sim v$, which gives us $b = v$. □

3.4 Linking Handwritten Code

Our framework of compilations also supports another version of the linking theorem, which could be relevant to linking compiled programs with hand-optimized versions of some functions. We state the theorem in terms of our simplest CPS translation, but it can be adapted easily to apply to any correct compilation. We also specialize this theorem to functions from booleans to booleans; more complicated statements admit broader policies.

Theorem 7. *Let e_1 be a source-level function body and e_2 be a source-level expression intended to be the body of a "let" that binds $\lambda x.\ e_1$, where e_2 only passes booleans to the function defined by e_1. Let e_1' be the body of an arbitrary realization of $\lambda x.\ e_1$ in the CPS language. Assume e_1' is a correct implementation, in the sense that, if the following hold for all σ, b_1, F, b_2, and r:*

- $\sigma\{x \mapsto b_1\}; e_1 \Downarrow b_2$
- $\forall y, k, \sigma' \supseteq \lfloor \sigma \rfloor\,\{x \mapsto (b_1, F)\}.\ \sigma'(y) = b_2 \wedge \sigma'(k) = F \Rightarrow \sigma'; k\ y \Downarrow r$

...then we have:

- $\lfloor \sigma \rfloor\,\{x \mapsto (b_1, F)\}; e_1' \Downarrow r$

If this obligation is satisfied, then:

$$\forall b.\ \cdot; (\mathsf{let}\ f = \lambda x.\ e_1\ \mathsf{in}\ e_2) \Downarrow b \Rightarrow \cdot; (\mathsf{let}\ f = (\lambda p.\ \mathsf{let}\ x = p.1\ \mathsf{in}\ \mathsf{let}\ k = p.2\ \mathsf{in}\ e_1')\ \mathsf{in}\ \lfloor e_2 \rfloor^{\mathcal{P}}) \Downarrow b$$

Proof. First, we define a compilation $\mathcal{C}^{\mathcal{H}}$ embodying just the representation of e_1 as e'_1.

- $\mathcal{C}^{\mathcal{H}}_{\mathsf{E}} = \{(e_1, (\hat{\lambda}x.\ k\ x), e'_1)\}$
- $\mathcal{C}^{\mathcal{H}}_{\mathsf{F}} = \{(e_1, \mathsf{let}\ x = p.1\ \mathsf{in}\ \mathsf{let}\ k = p.2\ \mathsf{in}\ e'_1)\}$

It is easy to show that this is a compilation, since the definition has been crafted to satisfy the relevant property. Establishing compilation correctness is not much harder. Definition 6 allows us to prove correctness inductively, but we do not need any inductive hypotheses, since we deal only with a single expression. Whatever the top-level AST constructor of e_1 is, we prove that case of correctness without referring to any of the evaluation subderivations or inductive hypotheses. The remaining correctness cases are contradictory, since they deal with compilation of expressions that cannot possibly equal e_1. The rest of the proof follows the outline of Theorem 3's proof from the end of the previous subsection. □

The technique embodied in this theorem is quite general. Whenever one can give a standalone proof that a CPS-level function body acts like some source-level counterpart, the former is a sound substitute for the latter. For first-order functions, this obligation tends not to be problematic. However, one cannot in general prove the correctness of a higher-order function in this way. A proof needs to know the CPS-level behavior of functional arguments when called at particular parameters. The relation $\sim$ only makes it possible to conclude that a function and its compiled version belong to the current compilation. What is missing is a way to structure a well-founded induction so that it is legal to assume that any such call to a functional argument behaves correctly.

The next section addresses this problem by weakening the definition of compilation correctness.

4 Composable Proofs via Strong Induction

The discussion at the end of the last section was not quite accurate. Some hand-written higher-order functions can be verified as correct compilations. Specifically, if the source-level function body immediately calls a functional argument, then Definition 6 includes the correctness of the CPS-level version of that function evaluation as an inductive hypothesis. We only run into trouble for any calls appearing later in the evaluation process.

This problem would show up in verifying a compiler that applies high-level algebraic rewrite rules. The correctness of a rule might depend on a function call that appears deeply nested within the source program's evaluation tree. Since Definition 6 is based on standard rule induction, inductive hypotheses are only made available for immediate subderivations.

All this suggests that it would be profitable to construct a new definition based on *strong rule induction*, where we have an inductive hypothesis for every recursive child of the current proof tree, not just the immediate children. To

accomplish this, it is helpful to define an auxiliary judgment that characterizes which additional expressions will be evaluated in the course of evaluating a particular starting expression. We write $\sigma_1; e_1 \sqsubset \sigma_2; e_2$ for the fact that evaluating e_2 in σ_2 will trigger evaluation of e_1 in σ_1.

$$\frac{}{\sigma; e_1 \sqsubset \sigma; e_1\ e_2} \quad \frac{}{\sigma; e_2 \sqsubset \sigma; e_1\ e_2} \quad \frac{\sigma; e_1 \Downarrow \langle \sigma', \lambda x.\ e \rangle \quad \sigma; e_2 \Downarrow v}{\sigma'\{x \mapsto v\}; e \sqsubset \sigma; e_1\ e_2}$$

$$\frac{\sigma_1; e_1 \sqsubset \sigma_2; e_2 \quad \sigma_2; e_2 \sqsubset \sigma_3; e_3}{\sigma_1; e_1 \sqsubset \sigma_3; e_3}$$

Definition 8 (compilation strong-correctness). *A compilation S is* strong-correct *if and only if the following condition holds for any A such that $S \subseteq A$.*

For all σ, e, and v, if the following are true:

- $\sigma; e \Downarrow v$
- $\forall \sigma', e', v'.\ \sigma'; e' \Downarrow v' \wedge \sigma'; e' \sqsubset \sigma; e \Rightarrow A \vdash \sigma'; e' \Downarrow v'\ @\ A$

...then:

- $A \vdash \sigma; e \Downarrow v\ @\ S$

Theorem 8 (soundness). *For any strong-correct compilation C, we have $\forall \sigma, e, v.\ \sigma; e \Downarrow v \Rightarrow C \vdash \sigma; e \Downarrow v\ @\ C$.*

Proof. By strong induction on the derivation of $\sigma; e \Downarrow v$. As in the proof of Theorem 4, the correctness obligation of C is precisely what we need to establish each inductive case. □

Theorem 9. *Every correct compilation is also strong-correct.*

Proof. Each definition's correctness conditions make explicit a particular induction scheme. The inclusion we want is a simple consequence of the fact that normal induction is a special case of strong induction. □

Theorem 10. *For strong-correct compilations C and C', their union $C \cup C'$ is also strong-correct.*

Proof. Following the same outline as Theorem 6. □

4.1 An Example of Linking a Handwritten Higher-Order Function

Assume that we have added pairs to the source language and extended the semantics and compilation machinery in the obvious way to support the extension. Consider the following program.

$$\begin{aligned} \mathsf{swap} &= \lambda p.\ (p.2, p.1) \\ \mathsf{map} &= \lambda p.\ (p.1\ p.2.1, p.1\ p.2.2) \\ \mathsf{swapMap} &= \lambda p.\ \mathsf{swap}\ (\mathsf{map}\ p) \end{aligned}$$

The function swap switches the order of the components of a pair, and the function map takes as input a pair of a function and another pair, building a new pair by applying the function componentwise to the old pair. The function swapMap is the composition of swap and map.

We can implement a specialized CPS-level version of swapMap, where we manually inline the calls to swap and map.

$$\begin{aligned}\mathsf{swapMap}' = \lambda p.\ &\mathsf{let}\ p' = p.1\ \mathsf{in}\ \mathsf{let}\ k = p.2\ \mathsf{in}\ \mathsf{let}\ f = p'.1\ \mathsf{in}\ \mathsf{let}\ d = p'.2\ \mathsf{in}\\ &\mathsf{let}\ k_1 = (\lambda r_1.\\ &\quad \mathsf{let}\ k_2 = (\lambda r_2.\mathsf{let}\ y = (r_2, r_1)\ \mathsf{in}\ k\ y)\ \mathsf{in}\\ &\quad \mathsf{let}\ d_2 = d.2\ \mathsf{in}\ \mathsf{let}\ z = (d_2, k_2)\ \mathsf{in}\ f\ z)\ \mathsf{in}\\ &\mathsf{let}\ d_1 = d.1\ \mathsf{in}\ \mathsf{let}\ z = (d_1, k_1)\ \mathsf{in}\ f\ z\end{aligned}$$

We would like to reason about linking swapMap′ with compiled programs. A good first step is a lemma characterizing the behavior of swapMap. We write swapMapB for the body of swapApp, without the initial λ.

Lemma 3. *If $\sigma; \mathsf{swapMapB} \Downarrow v$, then there exist σ', f, v_1, v_2, v_1', and v_2' such that:*

- $\sigma(p) = (\langle \sigma', \lambda y.\ f \rangle, (v_1, v_2))$
- $\sigma'\{y \mapsto v_1\}; f \Downarrow v_1'$
- $\sigma'\{y \mapsto v_2\}; f \Downarrow v_2'$
- $v = (v_2', v_1')$
- $\sigma'\{y \mapsto v_1\}; f \sqsubset \sigma; \mathsf{swapMapB}$
- $\sigma'\{y \mapsto v_2\}; f \sqsubset \sigma; \mathsf{swapMapB}$

Proof. By repeated inversion on the evaluation derivation for swapMapB. □

Theorem 11. *For any source-level expression e,*

$$\forall b.\ \cdot; (\mathsf{let}\ f = \mathsf{swapMap}\ \mathsf{in}\ e) \Downarrow b \Rightarrow \cdot; (\mathsf{let}\ f = \mathsf{swapMap}'\ \mathsf{in}\ \lfloor e \rfloor^{\mathcal{P}}) \Downarrow b$$

Proof. As in Theorem 7, we start by defining a compilation $\mathcal{C}^{\mathcal{S}}$ embodying just the representation of swapMap as swapMap′. We write swapMapB′ for the body of swapMap′ and write swapMapB″ for swapMapB′ without its initial two let bindings.

- $\mathcal{C}^{\mathcal{S}}_{\mathsf{E}} = \{(\mathsf{swapMapB}, (\hat{\lambda} x.\ k\ x), \mathsf{swapMapB}'')\}$
- $\mathcal{C}^{\mathcal{S}}_{\mathsf{F}} = \{(\mathsf{swapMapB}, \mathsf{swapMapB}')\}$

It follows easily that $\mathcal{C}^{\mathcal{S}}$ is a compilation. We want to establish that it is strong-correct, so we must show, for any $A \supseteq \mathcal{C}^{\mathcal{S}}$, that $A \vdash \sigma; e \Downarrow v \mathbin{@} \mathcal{C}^{\mathcal{S}}$ for any σ, e, and v satisfying certain conditions. The definition of strong-correctness allows us to assume local correctness of any environment-expression pair that is evaluated in the course of evaluating e at σ, with the exception of the initial pair $\sigma; e$.

From Definition 4, we can read off what we must prove. We know $\sigma; e \Downarrow v$. We assume $(e, k', e') \in \mathcal{C}^{\mathcal{S}}_{\mathsf{E}}$, so $e = \mathsf{swapMapB}$, $k' = (\hat{\lambda} x.\ k\ x)$, and $e' = \mathsf{swapMapB}''$; and we assume $\forall x \in \mathsf{dom}(\sigma).\ A \vdash \sigma(x) \sim \sigma'(x)$. We must find v' such that

- $A \vdash v \sim v'$
- $\forall r.\ (\forall x, \sigma'' \supseteq \sigma'.\sigma''(x) = v' \Rightarrow \sigma''; k\ x \Downarrow r) \Rightarrow \sigma'; \mathsf{swapMapB}'' \Downarrow r$

Our evaluation hypothesis matches the premise of Lemma 3 exactly, so there exist σ'', f, v_1, v_2, v_1', and v_2' such that:

- $\sigma(p) = (\langle \sigma'', \lambda y.\ f \rangle, (v_1, v_2))$
- $\sigma''\{y \mapsto v_1\}; f \Downarrow v_1'$
- $\sigma''\{y \mapsto v_2\}; f \Downarrow v_2'$
- $v = (v_2', v_1')$
- $\sigma''\{y \mapsto v_1\}; f \sqsubset \sigma; \mathsf{swapMapB}$
- $\sigma''\{y \mapsto v_2\}; f \sqsubset \sigma; \mathsf{swapMapB}$

The first and fifth items on this list give us exactly what we need to instantiate our inductive hypothesis. The same holds for the second and sixth items. Each instantiation gives us the existence of a CPS-level value compatible with the result from the source level. The rest of the proof of strong-correctness follows by the operational semantics of $\mathsf{swapMap}'$.

This establishes that $\mathcal{C}^{\mathcal{S}}$ is strong-correct. By Theorems 5 and 9, $\mathcal{C}$ is also strong-correct. By Theorem 10, $\mathcal{C} \cup \mathcal{C}^{\mathcal{S}}$ is strong-correct. The rest of the proof follows the outline of Theorem 7. □

5 Scaling Up to More Expressive Object Languages

We have extended the basic techniques of the previous sections to apply to a significant untyped Mini-ML language. This object language contains recursive functions, pairs, sums, mutable references, and exceptions. Its syntax is as follows.

$$e ::= x \mid e\ e \mid \mathsf{fix}\ f(x).\ e \mid \mathsf{let}\ x = e\ \mathsf{in}\ e \mid () \mid (e, e) \mid \mathsf{fst}(e) \mid \mathsf{snd}(e) \mid \mathsf{inl}(e) \mid \mathsf{inr}(e)$$
$$\mid (\mathsf{case}\ e\ \mathsf{of}\ \mathsf{inl}(x) \Rightarrow e \mid \mathsf{inr}(x) \Rightarrow e) \mid \mathsf{ref}(e) \mid !e \mid e := e \mid \mathsf{raise}(e) \mid e\ \mathsf{handle}\ x \Rightarrow e$$

Step-indexed logical relations have been used to handle most of these features. Some cleverness is needed to figure out how to adapt that technique to each new feature. In contrast, our syntactic approach can be applied more or less mechanically to any family of traditional, non-compositional syntactic proofs.

We have proved all of the main theorems from the paper, with respect to this Mini-ML language. It was straightforward to adapt our proof of CPS translation from earlier work on a verified Mini-ML compiler (Chlipala 2010). We added some new lines of boilerplate code, explicitly stating each of the new correctness conditions, where before they had been generated automatically by the induction tactic. Besides that boilerplate, we had to add or modify under 10 lines of proof, since we had written the original proof in a highly automated style.

Theorem 11, which considers a manual implementation of $\mathsf{swapMap}$, differs the most for the expanded language, compared to the version we sketched earlier. Evaluating a call to this higher-order function involves two calls to a functional argument, each of which may mutate the reference heap, and each of which may terminate the original call early by raising an exception. The extended version of Lemma 3 accounts for both dimensions.

6 Implementation

We have checked all of the proofs mechanically in Coq. The theorems of this paper are included in the `examples/Compose` directory of the latest distribution of our Lambda Tamer library, available from:

`http://ltamer.sourceforge.net/`

We encode syntax with parametric higher-order abstract syntax (Chlipala 2008) and encode semantics with substitution-free operational semantics (Chlipala 2010). In this style, explicit heaps of closures appear in evaluation judgments. These closure heaps show up sprinkled throughout the implemented versions of the definitions from the paper.

We represent compilations as modules in Coq's module system, which is influenced by ML module systems (MacQueen 1984). Each variety of correctness condition corresponds to a different module signature. The soundness theorem for each of these is realized as a functor from a compilation to a set of useful theorems. Compilation union is a two-argument functor, and correct compilations are translated into strong-correct compilations by another functor.

The whole development comes to about 3200 lines of code. The correctness proof for naive CPS translation is under 400 lines long, and the proof for the tail-call-optimizing version is also under 400 lines. A majority of the lines in each are boilerplate implied by the framework.

7 Discussion

The strong induction formulation of compilation correctness seems to be very permissive, but we do not mean to argue that this is precisely the formulation that should be used in practice. Further study is needed to determine which (sound) correctness conditions best enable the range of important compilation strategies. We only mean to recommend the general recipe of identifying proof skeletons common to many traditional syntactic proofs. Such a skeleton can be adapted without much new cleverness to yield a compositional framework. These derived frameworks are more elementary than the semantic frameworks developed to address similar concerns, and so they are more accessible to most of the community of people who might want to verify compilers.

Our implementations to date have only dealt with CPS translation, the first phase in the verified compiler that we built previously (Chlipala 2010). We are hopeful that the ideas generalize uneventfully to compilation of higher-order languages in general. We see no obstacle to handling translations that use the results of program analyses, since compilation relations may be defined via existential quantification over sound analysis results.

The formalism in this paper has only addressed translating facts about terminating source-level executions into facts about target-level execution. We believe that our technique is quite compatible with syntactic proofs based on coinductive big-step operational semantics (Leroy and Grall 2009), which facilitates

reasoning about nontermination in a style more suitable than that of small-step semantics to compiler verification.

An important dimension requiring further study is flexible support for compiling programs with mutable state. Our implementation fixes the convention that compiler outputs allocate new mutable references in lock-step with the corresponding input programs. This precludes, for instance, automatic memoization, where the compiler allocates new target-level references that have no source-level counterparts. We could generalize the framework to support such scenarios by parameterizing the $\sim$ relation by a map from source-level references to target-level references. To support practical compilation strategies, it may also be necessary to add further degrees of freedom to the definition of compilations, such that custom relations between source and compiled program states can be specified.

One important metric for evaluating verification approaches has not been very visible in this paper: the cost of dealing with details in a mechanical proof assistant. Compared to semantic approaches, we avoid reasoning about individual execution steps and about contexts, by using big-step semantics; and we avoid calculating step indices, by inducting over proof tree depths instead. Our relation $\sqsubset$ is fulfilling much the same role as step indices; we could replace our strong rule induction by strong induction over depths of evaluation derivations, which are very close to step indices. We save time by avoiding any such complex reasoning, where possible, by proving compilers correct according to the simpler correctness conditions, and then using Theorem 9 to lift these results into results about strong induction. This is the case for both of our CPS translations. In contrast, the semantic, compositional compiler proofs that we are aware of reason directly using step indices. Logical step-indexed logical relations (Dreyer et al. 2009) also address this weakness, via a modal logic that hides step indices; a modality must still be used explicitly in relation definitions and dealt with explicitly in proofs.

Compositional compiler correctness has a close connection to fully abstract compilation. Ahmed and Blume (2008) prove that typed closure conversion is fully abstract, in the sense that two source programs are observationally equivalent if and only if their compilations are. Our syntactic proofs of compiler correctness can be used to provide similar guarantees in a "proof-carrying code" setting: Our theorems do not tell us anything about general contextual equivalence of compiled programs, but we *do* get strong guarantees about any "contexts" shipped as programs that come with proofs of equivalence to source programs, relative to arbitrary correct compilations.

In summary, we have shown how elementary syntactic proof techniques can be adapted to verification of compilers for higher-order languages, in a way that supports sound linking with code produced in unanticipated ways. The wide range of interesting program properties guarantees that there will always be opportunity to develop useful new proof techniques, both "syntactic" and "semantic," and we have argued that the need for compositionality does not force us to leave syntactic methods behind.

Bibliography

Amal Ahmed. Step-indexed syntactic logical relations for recursive and quantified types. In *Proc. ESOP*, pages 69–83, 2006.

Amal Ahmed and Matthias Blume. Typed closure conversion preserves observational equivalence. In *Proc. ICFP*, pages 157–168, 2008.

Andrew W. Appel. Foundational proof-carrying code. In *Proc. LICS*, pages 247–256, 2001.

Andrew W. Appel and David McAllester. An indexed model of recursive types for foundational proof-carrying code. *TOPLAS*, 23(5):657–683, 2001.

Nick Benton and Chung-Kil Hur. Biorthogonality, step-indexing and compiler correctness. In *Proc. ICFP*, pages 97–108, 2009.

Adam Chlipala. A certified type-preserving compiler from lambda calculus to assembly language. In *Proc. PLDI*, pages 54–65, 2007.

Adam Chlipala. A verified compiler for an impure functional language. In *Proc. POPL*, 2010. Draft available at `http://adam.chlipala.net/papers/ImpurePOPL10/`.

Adam Chlipala. Parametric higher-order abstract syntax for mechanized semantics. In *Proc. ICFP*, pages 143–156, 2008.

Olivier Danvy and Andrzej Filinski. Representing control: A study of the CPS transformation. *Mathematical Structures in Computer Science*, 2(4):361–391, 1992.

Zaynah Dargaye and Xavier Leroy. Mechanized verification of CPS transformations. In *Proc. LPAR*, pages 211–225, 2007.

Derek Dreyer, Amal Ahmed, and Lars Birkedal. Logical step-indexed logical relations. In *Proc. LICS*, pages 71–80, 2009.

Xinyu Feng, Zhaozhong Ni, Zhong Shao, and Yu Guo. An open framework for foundational proof-carrying code. In *Proc. TLDI*, pages 67–78, 2007.

Nadeem A. Hamid, Zhong Shao, Valery Trifonov, Stefan Monnier, and Zhaozhong Ni. A syntactic approach to foundational proof-carrying code. *J. Automated Reasoning*, 31(3-4):191–229, 2003.

Nadeem Abdul Hamid and Zhong Shao. Interfacing Hoare Logic and type systems for foundational proof-carrying code. In *Proc. TPHOLs*, pages 118–135, 2004.

Xavier Leroy. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In *Proc. POPL*, pages 42–54, 2006.

Xavier Leroy and Hervé Grall. Coinductive big-step operational semantics. *Inf. Comput.*, 207(2):284–304, 2009.

David MacQueen. Modules for Standard ML. In *Proc. LFP*, pages 198–207, 1984.

Yasuhiko Minamide and Koji Okuma. Verifying CPS transformations in Isabelle/HOL. In *Proc. MERLIN*, pages 1–8, 2003.

J Strother Moore. A mechanically verified language implementation. *J. Automated Reasoning*, 5(4):461–492, 1989.

A. M. Pitts and I. D. B. Stark. Operational reasoning for functions with local state. *Higher Order Operational Techniques in Semantics*, 1998.

G. D. Plotkin. Lambda-definability and logical relations. Memorandum SAI-RM-4, University of Edinburgh, 1973.

Ye Henry Tian. Mechanically verifying correctness of CPS compilation. In *Proc. CATS*, pages 41–51, 2006.

Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Inf. Comput.*, 115(1):38–94, 1994.