Under Control: Compositional Correctness of Closure Conversion with Mutable State

Phillip Mates
Northeastern University

Jamie Perconti
Northeastern University

Amal Ahmed
Northeastern University

ABSTRACT
Compositional compiler verification aims to ensure correct compilation of components, not just whole programs. Perconti and Ahmed [2014] propose a methodology for compositional compiler correctness that supports linking with code of arbitrary provenance. In particular, they allow compiled components to be linked with code whose functionality cannot even be expressed in the compiler’s own source language. The essence of their approach is to define a multi-language system that formalizes interoperability between the source and target languages so that compiler correctness can be stated as contextual equivalence in the multi-language. They illustrate this methodology on a two-pass type-preserving compiler for a polymorphic language with recursive types.

We show how to extend this multi-language compiler-verification approach to a source language with ML-style mutable references. We present the first compositional correctness proof of typed closure conversion for a language with mutable state. More importantly, we show we can extend our target language with first-class control (call/cc) yielding a compiler correctness theorem that allows components compiled from the source language (without call/cc) to be linked with target-language components (with call/cc) whose extensional behavior cannot be expressed in the source. A non-trivial technical contribution is the design of the multi-language logical relation used to carry out the proof of compiler correctness. This is semantically challenging due to the mix of parametric polymorphism and mutable state in both interoperating languages.

CCS CONCEPTS
Software and its engineering → Correctness; Functional languages; Compilers; Interoperability.

KEYWORDS
Compiler correctness, typed closure conversion, multi-language semantics, mutable state, first-class continuations, logical relations

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ACM Reference Format:

ACM ISBN 978-1-4503-7249-7/19/10...

INTRODUCTION
Compositional compiler verification aims to formally verify correct compilation of components, not just whole programs. It has been the focus of much recent work, with researchers proposing several different approaches to specifying and proving such theorems [Perconti and Ahmed 2014; Stewart et al. 2015; Neis et al. 2015; Kang et al. 2016; Wang et al. 2014, 2019]. Still, it remains a difficult problem, and harder still if we want to accommodate linking with components compiled from a different language.

In 2014, Perconti and Ahmed proposed a methodology for compositional compiler correctness that supports linking with code of arbitrary provenance, (e.g., compiled from a different source language). To date, theirs is the only approach that (1) allows compiled components to be linked with code whose functionality cannot even be expressed in the compiler’s own source language; and (2) does not require that the source, target, and intermediate languages of the compiler have the same memory model. The essence of their approach is to define a multi-language system that formalizes interoperability between the source and target languages so that compiler correctness can be stated as contextual equivalence in the multi-language. They illustrate this methodology on a two-pass type-preserving compiler for a polymorphic language with recursive types, performing closure conversion and allocation to make data layout explicit. To demonstrate that their framework supports linking with code that cannot be expressed in the source, they add mutable references only to their target language and show an example of linking with code that internally uses a mutable reference as a counter. In comparison, SepCompCert [Kang et al. 2016], Pilsnér [Neis et al. 2015], and CompCertX [Wang et al. 2019] don’t support linking with code inexpressible in the compiler’s source language and Compositional CompCert [Stewart et al. 2015] only allows linking with code that satisfies the CompCert memory model. Meanwhile, SepCompCert and Compositional CompCert also rely on a uniform memory model across all the compiler’s languages.

In this paper, we show how to extend the Perconti-Ahmed multi-language approach to a source language with ML-style mutable references. We present the first compositional correctness proof of typed closure conversion for a language with mutable state. More significantly, we show we can extend our target language with first-class control (call/cc) yielding a compiler correctness theorem that allows components compiled from the source language (without call/cc) to be linked with target-language components (with call/cc) whose extensional behavior cannot be expressed in the source. Our source language M is essentially an idealized ML with polymorphism and mutable state while our closure-conversion target C adds call/cc and the restriction that functions must not have any free type or term variables.
signature THREAD = sig
  type thread
  val fork : (unit -> unit) -> unit
  val yield : unit -> unit
  val exit : unit -> unit
end

functor Thread () : THREAD = struct
  type thread = unit cont
  val readyQueue : thread Queue.queue = Queue.mkQueue ();
  fun dispatch () = let val t = Queue.dequeue readyQueue
     in throw t ()
     end
  fun enqueue t = Queue.enqueue (readyQueue, t)
fun fork f = callcc (fn parent =>
    (enqueue parent; f (); exit ()))
fun yield () = callcc (fn parent =>
    (enqueue parent; dispatch ()))
fun exit () = dispatch ()
end

Figure 1: A Simple Threads Library

The goal of our work is to allow linking of a component compiled from M with code that uses first-class control in a way that fruitfully disrupts the control flow of the M component. Consider, for example, a scenario where we write a component \( P \) in M and want to link it with a green threads library (which can be implemented in language C). Figure 1 presents such a simple threads library.

As another example, consider a continuation-based web server that provides \( \text{call/cc} \)-based primitives that a web programmer can use for creating client-server interaction points [Krishnamurthi et al. 2007; Queinnec 2003]. The programmer, meanwhile, might use our language \( M \) (without first-class control) to develop her web application, making use of the aforementioned primitives which allow her to write her program in a more readable direct style. In both of these situations, the programmer wishes to link with code compiled from a language with \( \text{call/cc} \) and benefit from \( \text{call/cc} \) disrupting the control flow of the compiled M component. Our verified compilers should be able to formally support such linking scenarios.

We develop a multi-language semantics between our source and target languages that is the first to support both mutable references as well as polymorphism in both interoperating languages. We then use contextual equivalence in the multi-language to state our compositional compiler correctness theorem. A nontrivial technical contribution of our work is the design of the multi-language logical relation used to carry out the proof of compiler correctness. This is semantically challenging due to the mix of parametric polymorphism and mutable state in both of the interoperating languages.

\section{The Source and Target Languages}

The source and target languages are both call-by-value. They are both in monadic normal form—a choice common for compiler intermediate languages—which means that constructors and eliminators are only applied to syntactic values [Benton et al. 1998]. Note that we will often refer to the \textit{terms} of each language as \textit{components}: this is meant to emphasize that individual terms are the level of granularity at which we ensure correct compilation of source components and linking with target components.

\begin{align*}
\tau & ::= \alpha \mid \text{unit} \mid \text{int} \mid \text{\ForAll}\tau ([\varpi].(\tau) \rightarrow \tau) \mid \exists \alpha.\tau \mid \mu\alpha.\tau \mid \text{ref } \tau \mid \langle \tau \rangle \\
p & ::= + \mid - \mid * \\
v & ::= x \mid () \mid n \mid \lambda\tau.[\varpi]e \mid \text{pack } (\tau, v) \text{ as } \exists \alpha.\tau \mid \text{fold}_{\mu\alpha.\tau} v \mid \ell \mid \langle \tau \rangle \\
e & ::= v \mid v \cdot v \mid \text{if} v \cdot e \mid v \cdot [\varpi][\psi] \mid \text{unpack } (\alpha, x) \cdot v \text{ in } e \mid \text{unfold} v \\
\text{new } v & ::= v \mid !v \mid \pi(v) \mid \text{let } x = e \text{ in } e \\
E & ::= [] \mid \text{let } x = E \text{ in } e \\
H & ::= H \mid H, \ell \Rightarrow v
\end{align*}

\begin{align*}
\langle H | e \rangle & \rightarrow \langle H | e' \rangle \\
(H | [\psi]E(v)) & \rightarrow \langle H | [\psi][\varpi]E(v) \rangle \quad \ell \notin H \\
(H | [\psi]E(\ell)) & \rightarrow \langle H | [\psi][\varpi]E(\ell) \rangle \quad \ell \in H
\end{align*}

\text{dom}(\cdot) \cap \text{dom}(\cdot') = \emptyset

\begin{align*}
\text{r} & ::= \cdot \mid \cdot, \ell : r \text{ and } \alpha & ::= \cdot | \cdot, \alpha \text{ and } \alpha' & ::= \cdot | \cdot, \delta : \alpha
\end{align*}

\begin{align*}
x : r & ::= \cdot \mid \cdot, \ell : r \\
r & ::= r \mid r \cap \ell : r \\
\ell & ::= \emptyset | \hat{\alpha} | \hat{\psi} | \hat{\tau} | \cdot + e : r'
\end{align*}
mapping from locations $\ell$ to their contents $v$. We define a small-step operational semantics as a relation on program configurations, using evaluation contexts $E$ to lift the primitive reductions to a standard left-to-right call-by-value semantics. The reduction rules are standard; we only show the application rule and the rules for creating a new initialized reference (new $v$), assignment ($v := v'$), and dereferencing ($v$).

The typing judgment for M has the form $\vdash \cdot : \tau$. The heap type $\tau$ tracks the types of heap locations $\ell$ in scope, where $\tau$ must be a closed type. The type environment $\tau$ tracks the type variables $\alpha$ in scope. The value environment $\tau$ tracks the term variables $x$ in scope along with their types $\tau$, which must be well formed under $\vdash$ (written $\vdash \cdot \tau$ and defined as $\text{ftv}(\tau) \subseteq \cdot$). The typing rules are standard and we omit most of them; they appear later as part of the extended judgment for type-directed closure conversion presented in Figure 5.

The typing judgment for heap fragments has the form $\vdash H : H'$, which says that the heap fragment $H$ is assigned the heap type $H'$ under the assumption that there is some (external) heap type $\tau$ with locations that may be referenced by values stored in $H$. Here $\tau$ must provide types for exactly the locations in $H$ and the values stored in $H$ must typecheck under the disjoint union of the two heap types $\tau$.

The reader may wonder why we need to typecheck heap fragments instead of whole heaps. In fact, in the language M, a judgment for typechecking whole heaps is sufficient for proving type soundness since we only appeal to this judgment when typechecking the (whole) heap in a program configuration. However, we adopt the more general judgment for typing heap fragments because we will need this ability in §4 when we combine our source and target variables.

Target Language: C. Our closure-conversion target language C is shown in Figure 3. The non-shaded parts of the figure show the language without first-class control (call/cc), which we will refer to as the base language below. This subset of C is nearly identical to M, with two exceptions. First, since this language is the target of closure conversion, functions are not allowed to contain free type or term variables. This is enforced by the C function typing rule as shown in Figure 3. Second, we allow the partial application of a function to a type. Hence, M terms include the value form $v[\tau]$.

The shaded parts of Figure 3 extend the base language with first-class control. First-class representations are contained by $\text{cont}_E$, a value form that injects evaluation contexts $E$ (with a hole of type $\tau$) into the term language. The type $\text{cont}_\tau$ is the type ascribed to first-class continuations $\text{cont}_E$ that expect a value of type $\tau$. Evaluation contexts $E$ are a subset of general contexts $C$, which are terms with a single hole. The typing judgment for contexts has the form $\vdash C : (\cdot ; \vdash \tau) \leadsto (\cdot ; \vdash \tau')$. Context typing ensures that, given any term $e$ that satisfies the type of the hole, $\cdot ; \vdash e : \tau$, we have that $\cdot ; \vdash C[e] : \tau'$.

Figure 3: Target Language C: Syntax & Semantics (excerpt)

with a continuation typing judgment $\vdash \cdot ; \vdash e : \tau$ that says that $E$ is an evaluation context with a hole of type $\tau$. We need the latter to type check first-class continuations $\text{cont}_E$.

We also add call/cc and throw with the standard operational semantics and typing rules. As usual, call/cc $E$ captures its current continuation $E$, binds it to a, and continues with $E[\text{cont}_E/a]$. Meanwhile, throw $v$ to $E$ throws away its current continuation $E$ and continues with $E[v]$.

3 TYPED CLOSURE CONVERSION

Closure conversion is a standard compiler pass that transforms functions with references to free variables—i.e., variables from the local environment—into closed functions in which all variable references are bound by the functions’ parameters. The transformation collects a function’s free term variables in a tuple called the closure.
environment and modifies the function itself to take the environment as an additional input. The closed function is paired with its environment to create a closure.

Typing the result of closure conversion poses a problem in that two source types with the same type but different free variables may end up with differently typed closure environments. As an example, consider two functions \( \lambda x. y + 1 \) and \( \lambda x. x \) of type \( \text{int} \to \text{int} \), where \( y \) is a free variable of type \( \text{int} \). The function part of their translations would have types \( (\text{int}, \text{int}) \to \text{int} \) and \( (\text{int}, \text{int}) \to \text{int} \), respectively, where the first argument in each case is the function’s environment. The solution to this problem was proposed by Minamide et al. [1996], who used an existential type to abstract the type of the environment, thus hiding the fact that the closures’ environments may have different types.

Function definitions in \( M \) may also have references to free type variables, which means we must also transform functions to take their free type variables as additional arguments. However, instead of collecting these types in a type environment as Minamide et al. do, we follow Morrisett et al. [1999] and directly substitute the types into the function. Like Morrisett et al. and Perconti and Ahmed [2014], we adopt a type-erase interpretation, which means that since all types are erased at run time the substitution of types into functions has no run-time effect.

Our typed closure-conversion pass compiles \( M \) terms of type \( \tau \) to \( C \) terms of type \( \tau^C \). Figure 4 presents the type translation \( \tau^C \). The only interesting case is that for function types which are transformed into an existential type where \( \beta \) represents the (abstract) type of the closure environment. Figure 4 also presents the translation of environments \( \cdot^C \), \( \cdot \), and \( \cdot^{C^C} \), which we use to typecheck a closure-converted term.

Figure 5 presents the term translation, which is defined by induction on typing derivations. We translate an \( M \) variable \( x : \tau \) into a \( C \) variable \( x : \tau^C \), and similarly, translate an \( M \) location \( f : \text{ref} \tau \) into a \( C \) location \( f : \text{ref} \tau^C \). Since this is closure conversion, the interesting cases of the translation are those that involve functions and application. The omitted rules are defined by structural recursion on terms. Note that even though our target language \( C \) contains \text{call/cc}, no types or terms pertaining to first-class continuations—shown shaded in Figure 3—appear in the output of our translation.

4 MULTI-LANGUAGE SEMANTICS

We define a multi-language semantics to specify interoperability between \( M \) and \( C \) components along the lines of Matthews and Findler [2007]; Ahmed and Blume [2011]; Perconti and Ahmed [2014]. Our \( M+C \) multi-language system embeds the languages \( M \) and \( C \) so that both languages have natural access to foreign values (i.e., values from the other language). For instance, they receive foreign integer values as native values, and can call foreign functions as native functions. We extend the original languages with new syntax, evaluation contexts, and reduction rules that define syntactic boundaries \( \langle M \rangle \) and \( \langle C \rangle \) to allow cross-language communication. The term \( \langle M \rangle e \) (\( C \) inside, \( M \) outside) allows a term \( e \) of target type \( \tau^C \) to be used as a term of source type \( \tau \), while the term \( \langle C \rangle e \) (\( M \) inside, \( C \) outside) allows a term \( e \) of source type \( \tau \) to be used as a target term of translation type \( \tau^C \).

Our semantics follows Perconti and Ahmed’s except that we add mutable references. The tricky parts, which involve abstract types, are inherited from Perconti and Ahmed so we summarize their work before discussing mutable references. A term \( \langle C \rangle e \) has type \( \tau^C \) if \( e \) has type \( \tau \) and to evaluate it we first reduce \( e \) to a value \( v \) (using \( M \) reduction rules). Then a type-directed meta-function called the value translation is applied to \( v \), yielding a value \( \tau^C \) in \( C \) of type \( \tau^C \) (written \( \langle C \rangle v = \tau^C \)). The value translation is only defined for closed values since it is used at run time: even if we write programs with free variables under boundaries, we will have substituted closed values for all variables since we only run closed programs.

Value translation is easy at base types, e.g., a value \( n \) of type \( \text{int} \) is converted to the same integer \( n \) in \( C \). Most of the other types are translated simply by structural recursion, but functions are the interesting case (shown below). To translate an \( M \) function of type \( \tau \to \tau' \), we construct a closure in \( C \). Since at runtime a source function will be closed by substitution, the closure has the empty environment. Additionally, the input of the function is translated to \( M \) so it can be passed to the original function whose output is then translated to \( C \) (as with higher-order contracts). To translate a \( C \) closure to \( M \), we produce an \( M \) function that unpacks the closure, translates the input, calls the \( C \) function with the translated input and its packed environment, and translates the output to \( M \).

\[
\text{CM}^\tau \to \tau^C (v) = \text{pack} \left( \text{unit}(\langle v \rangle) \right) \text{as } \beta.\left((\langle \beta \rangle \tau^C) \to \tau^C\right)\left(\beta\right),
\]

where \( v = \lambda(x: \text{unit}, \tau^C, \text{MC}^\tau) \). Naively value-translation a value \( v \) of this type, we get:

\[
\langle C \rangle v \to \alpha \text{MC}(v) = \lambda(x : \tau^C)\text{MC}^\tau(\text{unpack} \beta, y) = v \text{ in } \pi_1(y)\pi_2(y) \text{MC}^\tau(x)
\]

Next, consider the type \( \forall [\alpha] (\langle \alpha \rangle \to \alpha) \to \alpha \). Since \( \langle C \rangle \) has the translation of this type is \( \forall [\alpha] (\langle \alpha \rangle \to \alpha) \to \alpha \). Naively value-translation a value \( v \) of this type, we get:

\[
\forall [\alpha] (\langle \alpha \rangle \to \alpha) \to \alpha \text{MC}(v) = \lambda([\alpha][x : \alpha], \alpha \text{MC}^\tau(\text{unpack} \beta, y) = v \text{ in } \pi_1(y)\pi_2(y) \text{MC}^\tau(x)
\]

But note that \( \langle C \rangle \) can be bound by \( \alpha \), which means \( \alpha \) would be bound! Perconti and Ahmed point out that what we want instead is that when \( \alpha \) is instantiated with a concrete type \( \tau \), the positions inside language \( C \) where that type is needed receive \( \tau^C \). To this end, they add a type \( [\alpha] \) (\( \alpha \) suspended in \( C \)) that allows an \( M \) type variable to appear in a \( C \) type. The \( M \) type variable \( \alpha \) needs to be translated, but the translation is \( \text{delayed} \) until \( \alpha \) is instantiated with a concrete type. This is enforced in the definition of type substitution: \( [\alpha] \to M \langle\alpha\rangle = \tau^C \). In addition, they define a modified type translation—boundary type translation \( \tau^C \to \tau^C \) to turn \( M \) type variables into suspended type variables instead of \( C \) type variables. Formally, the rule for type variables in the compiler’s type translation is replaced...
produce a closed value. Moreover, the boundary terms in the body of $v$ expect to be annotated with a type that translates to $\alpha$. To fix this, Perconti and Ahmed introduce a lump type $L(\tau)$ that allows passing C values to M terms as opaque lumps. The introduction form for the lump type is the boundary term $\bar{\tau}(\ell)MCv$, and the elimination form is $\text{CM}^{\ell}(\tau)\alpha$. Opposite boundaries at lump type cancel to yield the underlying C value. Thus, our boundary type translation defines $L(\tau)(\alpha) = \tau$. Now, each $\alpha$ in the above example can be changed to $L(\alpha)$.

Consider translating a value $\ell$ to the other language. We can’t generate a new C location $\ell$—how would we keep these mutable references in sync? The only option is to require $\text{CM}^{\ell}(\tau)\alpha$ (and similarly, $\text{CM}^{\ell}(\tau)\alpha$) to be a value form that can be accessed (upated and dereferenced) from the other language. Figure 6 and 7 present the details, including the typing rules and complete value translations. Note that call/cc captures the entire continuation (not just the part that is red code, until the first boundary).

Our multi-language semantics for mutable references resembles the treatment of references in Dimoulas et al. [2012]’s imperative Contract PCF (CPCF). Our value forms $\text{CM}^{\ell}(\tau)\alpha$ and $\text{CM}^{\ell}(\tau)\alpha$ are analogous to theirs where a guard attaches itself permanently around locations when locations cross component boundaries.

Like Perconti-Ahmed, our goal is to define a multi-language for stating compositional compiler correctness. Thus, an important property our multi-language must satisfy is boundary cancellation, which says that wrapping a term in opposing boundaries is contextually equivalent to the underlying term (discussed further in §7).

## 5 COMPOSITIONAL CORRECTNESS OF CLOSURE CONVERSION

We specify correctness of our closure-translation pass from M to C in terms of contextual equivalence in the multi-language M+C. In this section, we formally define contextual equivalence for M+C components and present the compiler correctness theorem. Later, in §7, we will show to prove the compiler correctness theorem using a logical relation that gives us a sound and complete proof method for M+C contextual equivalence.

Multi-language Contextual Equivalence. As usual, a general context $C$ is an M+C expression with a single hole in it. However, there are two ways in which M+C contexts are not entirely standard. First, since this a multi-language, we can only plug a hole in the context with a term that is the rightmost outermost color—i.e., a term
Figure 7: M+C: Dynamic Semantics (extends Figs 2 and 3)

Figure 8: M+C General Contexts (excerpt)
e2 are such that the expressions can be legally plugged into the context’s hole. In other words, we assume that C[e1] and C[e2] are syntactically valid expressions such that $\Psi' : \vdash C[e1]: \tau'$.

**Compositional Compiler Correctness.** We can now state our main result, that closure conversion of M components into C components is semantics preserving.

**Theorem 5.2 (Closure Conversion is Semantics Preserving).** If $I : \tau; \bar{x} : \tau^\alpha \vdash e : \tau \rightsquigarrow e$, then

$$I : \tau; \bar{x} : \tau^\alpha \vdash e \equiv_{\text{M} \rightarrow \text{C}} \tau MC(e)\text{CM}^{\text{ref}}\tau' / \tau[\alpha/\alpha]\text{CM}^{\text{ref}}(\cdot/\cdot); \cdot : \tau \rightsquigarrow \tau'.$$

The formal theorem essentially says that if e compiles to e then $e \equiv_{\text{M} \rightarrow \text{C}} e'$, but we need to perform a substitution so that the free locations, term variables and type variables of the source component match those of the compiled component. The source component e may contain free locations $\ell : \tau$, free type variables $\alpha$, and free term variables $x : \tau'$, which the compiler translates into $I : \tau; \bar{x} : \tau^\alpha \vdash e : \tau$, and $x : \tau' \rightsquigarrow \tau'$, respectively, in the output term e. But the contextual equivalence in the above theorem uses the blue M environments, so the compiled component e may only refer to variables and locations from those environments. Since we are in M+C, we can get the free locations and variables of the two components back in sync by substituting wrapped locations $\text{CM}^{\text{ref}}\tau$ for translated locations, suspended type variables $[\alpha]$ for translated type variables, and the wrapped term $\text{CM}^{\text{ref}}(\cdot)$ for translated term variables. Note that since we are in a call-by-value language, we have to be careful to substitute value forms for term variables.

The above compiler correctness theorem can equivalently be stated using the translated C environments and with the substitution on the other side, as follows:

$$I : \tau; \bar{x} : \tau^\alpha \vdash e \equiv_{\text{M} \rightarrow \text{C}} \tau MC\ell / \ell[\alpha/\alpha]\text{CM}^{\text{ref}}\tau^\alpha / \tau[\alpha/\alpha]\text{CM}^{\text{ref}}(\cdot/\cdot); \cdot : \tau \rightsquigarrow \tau',$$

where $\bar{\tau} = \tau[\alpha/\alpha]$, and similarly, $\bar{\tau}' = \tau'[\alpha/\alpha]$ and $\bar{\tau}'' = \bar{\tau}'[\alpha/\alpha]$.

Moreover, it does not matter which side the boundary term is placed on, since boundary cancellation lemmas (discussed in §7) allow us to prove, for instance, the following as a corollary:

$$I : \tau; \bar{x} : \tau^\alpha \vdash \text{CM}^{\text{ref}}\ell / \ell[\alpha/\alpha]\text{CM}^{\text{ref}}\tau^\alpha / \tau[\alpha/\alpha]\text{CM}^{\text{ref}}(\cdot/\cdot); \cdot : \tau \rightsquigarrow \tau^C.$$

Note that this version of the theorem uses the compiled type translation $\tau^C$ for the result type (instead of the compiler’s type translation $\tau^C$) since we need to ensure that (blue) type variables in the environment remain connected to their free occurrences in the result type.

### 6 Examples: Linking With Target Code that Uses CALL/CC

In this section, we show that we can link components compiled from M with C components that make use of call/cc whose extensional behavior cannot be expressed in M. As discussed in §1, such linking is useful to programmers when they want to link with a library written in a different language that provides some functionality they cannot implement in their own source language. We also discuss when our compiler correctness theorem lets us use single-language reasoning instead of mixed-language reasoning.

**6.1 A Simple Example with Callbacks**

We have claimed that we can link with target code inexpresible in the source; we now make that claim precise. We will demonstrate two programs $e_1$ and $e_2$ that are contextually equivalent in the source language M and then show a target context C that can distinguish between the compiled versions of $e_1$ and $e_2$. This means that the context C has behavior that cannot be expressed by any well-typed source context C that $e_1$ and $e_2$ may be linked with.

Consider the following two M components of type $\tau$. (This is the “awkward” example taken from Ahmed et al. [2009]; Dreyer et al. [2012].)

\[
\begin{array}{ll}
\tau & = ((\text{unit}) \to \text{unit}) \to \text{int} \\
e_1 & = \lambda x.\text{new} 0 \text{ in } \lambda f_x.\text{let } x := 0; f_x(); x := 1; f_x(); \text{ex}
\end{array}
\]

The components $e_1$ and $e_2$ are contextually equivalent in the source language M (see Dreyer et al. [2012] for a proof); that is, there is no M context C that can distinguish between them. Intuitively, this is because the reference $x$ is kept private to the function returned by $e_1$ and cannot be modified by the callbacks to $f$, so every time this function is called, $x$ is set to 0 but will eventually be set to 1. Hence, the functions returned by both $e_1$ and $e_2$ will always return 1 when applied to any M function of the appropriate type. The terms $e_1$ and $e_2$ below, of translation type $\tau^C$ (i.e., $((\text{unit}) \to \text{unit}) \to \text{int}^C$), are the compiled versions of $e_1$ and $e_2$:

\[
\begin{array}{ll}
\tau^C & = 3\beta.(((\beta, 3\alpha.((\alpha, \text{unit}) \to \text{unit}), \alpha)) \to \text{int}), \beta \\
e_1' & = \lambda x.\text{new} 0 \text{ in } \lambda f_x.\text{pack}((\text{ref} \text{int}), (\lambda(x.\text{env} \cdot \text{ref} \text{int}), f_1(\text{unit}) \to \text{unit}^C), e_{i,2}'; (x))
\end{array}
\]

While $e_1$ and $e_2$ are contextually equivalent in the source language, their compiled versions $e_1$ and $e_2$ may be linked with the target-language context C that we present below, which is able to distinguish between them. First, we show a version of C written in a non-closure-converted (hence, more readable) form, as written, say, in an M-like language with call/cc:

\[
\begin{array}{ll}
C & = \text{let } g = [] \text{ in } \text{let } b = \text{new} 1 \text{ in } \\
& \text{let } f = \lambda (_).\text{let } b' = \text{new} 1 \text{ in } \\
& \\
& \text{if } 0 \text{ bv } (\text{call/cc} \text{unit}(k, g(\lambda (\_).\text{throw} \text{unit} (1) \to k))) \\
& (b := 0) \text{ in } \text{g } f
\end{array}
\]

Note that if we could link C with the source components $e_1$ and $e_2$, the program $C[e_1]$ would reduce to 0, while $C[e_2]$ would reduce to 1. The context C can distinguish between $e_1$ and $e_2$ by using call/cc to capture the continuation k of the second call to f, after which it sets x back to 0 and then throws control back to k.

The above context C can be expressed in our target language as C below, which has a hole of type $\tau^C$.
We can push the context plugging under the boundary to get:

Applying boundary cancellation, we have:

Thus, we are now essentially equating a purely

potential type that exports three functions: fork, yield, and exit. Such

a threads package can be closure converted to our target language

structure T = Thread()
fun some1 () =
  (< some1 work start >
  < some1 work continue >
  < some1 work finish >
  T.yield ();
  < some1 work continue >
  < some1 work finish >
  T.yield ();
  < some1 work finish >
  T.exit ()
fun some2 () =
  (< some2 work start >
  T.yield ();
  < some2 work continue >
  T.yield ();
  < some2 work finish >
  T.exit ()
T.fork (some1);
T.fork (some2);
T.yield ();
T.yield ();

Figure 9: Code that uses threads package from Figure 1 (left), and effect of running linked program (right)

C, yielding threads. (We do not show the closure-converted implementation of the threads package as it is simply a less readable version of the SML implementation in Figure 1.)

Now consider the SML code in Figure 9 (left), which uses fork, yield, and exit from the threads package to interleave the work being done by two functions (some1 and some2). This code can easily be written in our source language M, then compiled to C and linked with the aforementioned threads package threads. If we run the resulting C program we would have the interleaving effect shown on the right in Figure 9.

7 LOGICAL RELATION FOR M+C

The compiler correctness theorem we presented in §5 is stated in terms of M+C contextual equivalence, but it is well known that direct proofs of contextual equivalence are difficult. The problem is the quantification over all contexts C in the definition of contextual equivalence (≈ctx M+C) which makes direct proofs intractable. Rather than prove contextual equivalences directly, we devise a logical relation for our multi-language M+C, and use it to prove our compiler correctness theorem.

We design a step-indexed, biorthogonal, Kripke logical relation that inherits elements from the multi-language logical relation devised by Perconti and Ahmed [2014]. The latter extended the standard Kripke logical relations design by Dreyer et al. [2012] with the ability to handle multi-language type abstraction. Our main challenge was figuring out how to extend Perconti and Ahmed’s logical relation to accommodate wrapped mutable references since in our multi-language these take the form of wrapped locations ref/MC/l and CAMref r/l that are value forms. In a nutshell, we needed to devise a more general form of logical relation that not only allows us to relate two values from the same language but also values across languages, as we shall explain. This in turn led us to revisit the relational interpretations for types α that the logical relation is normally parametrized with: our relational interpretation allows values to be related across languages.
Below we start with an overview of the basic structure of the logical relation and then discuss novel aspects of the relation and the major steps required to prove the logical relation sound and complete for M+C contextual equivalence. We elide details of how we prove many basic properties of the construction and also elide proofs of the Fundamental Property of the logical relation, and the proof of compiler correctness. Complete definitions and proofs are given in the technical appendix [Mates et al. 2019].

Overview of the Logical Relation. The basic idea of logical relations is to define an equivalence relation on program terms by induction on the structure of their types. For instance, two functions are related at the type \( \tau_1 \rightarrow \tau_2 \) if, given related arguments at type \( \tau_1 \), the functions yield related results at type \( \tau_2 \). Two tuples of length \( n \) are related at type \( (\tau_1, \ldots, \tau_n) \) if their \( i \)-th components (for all \( 1 \leq i \leq n \)) are related at type \( \tau_i \).

In the presence of state, we have to use Kripke logical relations, which are relations indexed by possible worlds \( W \). Kripke logical relations are needed when relatedness only holds under certain conditions; possible worlds allow us to capture these conditions and specify constraints on how the conditions may evolve over time. Our worlds \( W \) will specify constraints on heaps. We write \( (H_1, H_2) : W \) when the heaps \( H_1 \) and \( H_2 \) satisfy \( \tau \). For instance, two locations \( l_1 \) and \( l_2 \) should only be related at type \( \text{ref } \tau \) if they actually exist in some heaps that satisfy the current world \( W \); if they contain heap values related at type \( \tau \); and if they continue to exist and contain related values in all future worlds \( W' \) that are accessible from \( W \) (written \( W' \sqsupseteq W \), where \( \sqsupseteq \) is pronounced "extends"). An important property of Kripke logical relations is monotonicity, which says that if two values are related in world \( W \), then they must be related in all future possible worlds \( W' \) that extend \( W \).

Finally, step-indexed logical relations allow one to easily deal with semantic features that lead to “circularities” in the construction of semantic models, e.g., recursive types [Ahmed 2006] and mutable references that store functions [Ahmed 2004; Ahmed et al. 2009]. While standard logical relations are usually defined by induction on the structure of types, in the presence of contravariant recursive types and higher-order store, this scheme is no longer well-founded. The idea behind step-indexing is to define the logical relation by outer induction on a natural number that, intuitively, corresponds to the number of steps of computation for which two programs behave in a related manner, and then nested induction on the structure of types. The reason why step-indexing helps break the aforementioned circularities is, intuitively, that unfolding a recursive type and dereferencing a location consumes a step, which justifies lowering the step index in appropriate places so we can give an inductively defined logical relation.

Figures 10 and 11 present the important pieces of our logical relation. The high-level idea is that we define a value relation \( V[\tau] \) that relates closed values at type \( \tau \), a continuation relation \( K[\tau] \) that relates closed continuations (evaluation contexts) that have a hole of type \( \tau \), and a term relation \( E[\tau] \) that relates closed terms at type \( \tau \). Each of these relations is indexed by a world \( W \). These relations are built from well-formed worlds (which we discuss below) and well-typed values, continuations, and terms, requirements captured by the TermAtom/ValAtom/ContAtom definitions at the top section of Figure 10. (We will explain the value/term/continuation relations in Figures 10 and 11 in detail a little later.) We then generalize the definition to open terms (written \( \Psi; \Delta; \Gamma \vdash e_1 \Rightarrow_{M+C} e_2 : \tau \)).

Worlds and World Extension. Our worlds \( W \) are 4-tuples of the form \( (k, \Psi_1, \Psi_2, \Theta) \), where \( k \) is the number of computation steps we have left, \( \Psi_1 \) and \( \Psi_2 \) are the heap types that any heaps \( H_1 \) and \( H_2 \), respectively, must have if they are to satisfy \( W \), and \( \Theta \) is a sequence of islands that specify invariants on disjoint parts of the heap. Figure 10 (top) presents the requirements on the structure of worlds and islands, which are entirely standard (so we refer the reader to Dreyer et al. [2012] for details). Briefly, islands can encode state-transition systems (STS), where \( s \) denotes the current state of the STS; \( S \) denotes the set of all states; \( \delta \) are all possible transitions; \( \pi \) is the subset of \( \delta \) that are marked as “public” transitions\(^1\); \( HR \) are heap relations that, given the current state \( s \) of the STS, tell us what heap fragments are related; and \( bij \) keeps track of the correspondence (bijection) between “global” locations in the island’s heaps for the two programs.

Figure 10 (top) also defines \( k \)-approximation \((\cdot)_k \) which drops information at level \( k \) and higher from the islands, heap relations, etc., as well as the “later” operator \((\triangleright)\) which moves us to a world with one fewer step—we use the later operator in parts of the definition where we must decrement the step index to ensure a well-founded relation. It also defines world extension \((\sqsupseteq)\), which essentially says that in a future world there may be fewer steps available, all current locations and their types must be unchanged, there may be additional locations, and the island’s STS may have made valid transitions to a future state. We omit the definition of \( \sqsupseteq_{\text{pub}} \), which is identical to \( \sqsupseteq \) except that \( \sqsupseteq_{\text{pub}} \) for islands requires \((s, s') \in \pi \) instead of \((s, s') \in \delta \).

Value Relation: Standard Parts. The standard practice is for each of the above relations, \( V[\tau] \), \( K[\tau] \), and \( E[\tau] \), to be parametrized by a mapping \( \rho \) that provides relational interpretations for the free type variables in \( \tau \). For now, assume that \( \rho \) maps type variables \( \alpha \) to triples \( \text{VR} := (\tau_1, \tau_2, R) \), where \( \tau_1 \) and \( \tau_2 \) are the types used to instantiate \( \alpha \) on the left and right sides, respectively, and \( R \) is a relation between values of those types, or more precisely some \( \varphi \subseteq \text{ValAtom}(\tau_1, \tau_2) \). We will explain shortly why this structure is not quite what we need—that is, why our \( R \) is not simply a \( \varphi \) and what \( R[M, M] \) and \( R[C, C] \) signify—but the above suffices to explain the general principles of the logical relation. We write \( \rho_1 \) for the substitution that instantiates each \( \alpha \in \text{dom}(\rho) \) with the corresponding \( \tau_1 \) and \( \tau_2 \) for the substitution that instantiates each \( \alpha \) with the corresponding \( R \). We use dot notation to extract the components of the triple \( \text{VR} = (\tau_1, \tau_2, R) \) via \( \text{VR}._{\tau_1}, \text{VR}._{\tau_2}, \) and \( \text{VR}.R \).

We briefly explain the M cases of the value relation, \( V[\tau] \), which are shown in the bottom section of Figure 10. (For now, ignore the middle section of Figure 10, which we return to when we discuss the value relation for mutable references.) Values of base type are related if they are the same value. Values are related at type \( \alpha \) if they are in the relational interpretation of \( \alpha \), namely \( \rho(\alpha).R \). (For now, ignore the \([M, M]\) that follows \( R \) in the figure.)

---

\(^1\)Dreyer et al. [2012] use private transitions to reason about well-bracketed control flow in the absence of call/cc. Note that the shared part in the definition of Island \((\pi = \delta)\), which we add in to our definition in the presence of call/cc: it says that public transitions are the same as all transitions, which means there are no private transitions in the presence of call/cc.
\[
\begin{align*}
\text{Heap}_{\text{atom}} & \overset{\text{def}}{=} \{(W, H_1, H_2) \mid W \in \text{World}_{\text{atom}} \} \\
\text{Heap}_{\text{rel}} & \overset{\text{def}}{=} \{\rho_H \subseteq \text{Heap}_{\text{atom}} \mid \forall (W, H_1, H_2) \in \rho_H. \forall W' \triangleright W, (W', H_1, H_2) \in \rho_H \} \\
\text{Term}_{\text{atom}}[r_1, r_2] & \overset{\text{def}}{=} \{(W, e_1, e_2) \mid W \in \text{World}_{\text{atom}} \wedge W.\Psi_1 : \cdots \vdash e_1 : r_1 \wedge W.\Psi_2 : \cdots \vdash e_2 : r_2 \} \\
\text{Val}_{\text{atom}}[r_1, r_2] & \overset{\text{def}}{=} \{(W, v_1, v_2) \mid \in \text{Term}_{\text{atom}}[r_1, r_2] \} \\
\text{Island}_{\text{atom}} & \overset{\text{def}}{=} \{\theta = (s, S, \delta, \pi, \text{HR, bij}) \mid s \in S \wedge S \in \text{Set} \wedge \delta \subseteq S \times S \wedge \pi \subseteq \delta \wedge [\pi = \delta = ] \} \\
\text{World}_{\text{atom}} & \overset{\text{def}}{=} \{W = (k, \Psi_1, \Psi_2, \Theta) \mid k < n \wedge \exists \mathfrak{m}, \Theta \in \text{Island}_{\text{atom}} \} \\
\text{Val}_{\text{atom}}[r] & \overset{\text{def}}{=} \text{Val}_{\text{atom}}[\rho_1(r), \rho_2(r)] \\
\text{Cont}_{\text{atom}}[r_1, r_2] & \overset{\text{def}}{=} [r_1', r_2'] \overset{\text{def}}{=} \{(W, E_1, E_2) \mid W \in \text{World} \wedge \exists \Psi_1, \Psi_2, \tau : E_1:(W.\Psi_1 : \cdots \vdash \tau_{r_1}) \wedge + E_2:(W.\Psi_2 : \cdots \vdash \tau_{r_2}) \to (\Psi_1 : \vdash \tau_{r_1}) \wedge (\Psi_2 : \vdash \tau_{r_2}) \} \\
\text{Cont}_{\text{atom}}[r_1, r_2] & \overset{\text{def}}{=} [r_1', r_2'] \overset{\text{def}}{=} \text{Cont}_{\text{atom}}[\rho_1(r_1), \rho_2(r_2)] \to \rho_1'(r_1'), \rho_2'(r_2')] \\
\text{Val}_{\text{atom}}[r] & \overset{\text{def}}{=} \text{Val}_{\text{atom}}[\rho_1(r), \rho_2(r)] \\
\text{Cont}_{\text{atom}}[r_1, r_2] & \overset{\text{def}}{=} [r_1', r_2'] \overset{\text{def}}{=} \{(W, E_1, E_2) \mid W \in \text{World} \wedge \exists \Psi_1, \Psi_2, \tau : E_1:(W.\Psi_1 : \cdots \vdash \tau_{r_1}) \wedge + E_2:(W.\Psi_2 : \cdots \vdash \tau_{r_2}) \to (\Psi_1 : \vdash \tau_{r_1}) \wedge (\Psi_2 : \vdash \tau_{r_2}) \} \\
\end{align*}
\]

Figure 10: M+C Logical Relation: Auxiliary Definitions (top); Value Relation (middle, bottom)
Normally, functions are related if, in any future world \( W \), applying them to related arguments yields related results. Our functions, however, combine type- and term-abstracted values, so we say that two functions are related at type \( \mathcal{V}[\mathcal{F}] : \mathcal{P} \rightarrow \tau \) if the following holds: given any admissible relation triple \( \mathcal{R} = \langle \tau_1, \tau_2, R \rangle \) for each \( a \) that the function abstracts over, applying the functions in some future world \( W \) to (1) the types \( \tau_1 \) and \( \tau_2 \), respectively, and (2) argument values related in \( W \) at types \( \tau \) under a mapping \( \rho \) extended to account for the new \( R \) for each \( a \), they yield related results. Here, the definition in the figure says the functions yields results related in the relation \( \mathcal{E}[\rho, \tau] \). We will explain this “two-dimensional” \( \mathcal{E} \) relation later (when we discuss mutable references). For now, it suffices to think of \( \mathcal{E}[\rho, \tau] \) simply as \( \mathcal{E}[[\tau]] \) which specifies when computations are related at the type \( \tau \).

Packages are related at existential type \( \exists a. \tau \) if there exists some interpretation \( \mathcal{V} \) of the abstract type under which their bodies are related. Values of recursive type \( \mu a. \tau \) are related if unfolding the folded value yields values that are related at the type \( \tau[a/\alpha] \) after expending a step (denoted by the later operator \( \triangleright \)). Tuples are related if all their components are related at the appropriate types. Finally, lumps are related if the underlying values are related at the underlying type.

Most \( C \) cases of the value relation, \( \mathcal{V}[\tau] \), are shown at the bottom of Figure 10. The value relations for \( C \)'s base, function, existential, recursive, and tuple types are analogous to the corresponding \( M \) cases. The only tricky case is the case for suspended type variables \( [a] \) which we discuss below.

Figure 11 presents the term and continuation relations. In the term relation \( \mathcal{E}[\rho, \tau] \), two terms are related if running them in related continuations gives related observations. Two continuations are related in \( \mathcal{K}[\rho] \) if whenever we are given related values in some future world (under a restricted notion of public future worlds; see Dreyer et al. [2012]), then running the continuations with those values gives us related observations. This technique of defining the term relation \( \mathcal{E} \) by appealing to a continuation relation \( \mathcal{K} \) is referred to as biorthogonality or \( \tau^+ \)-closure, and it yields a logical relation that is complete with respect to contextual equivalence.

Under the relation \( \mathcal{O} \), two closed terms give us related observations in world \( W \) if, when we run them in two heaps that satisfy \( W \), either they both terminate, or they are both still running after \( k \) steps, where \( k \) is the number of steps allowed by world \( W \).

Value Relation: Mutable References. In a single-language Kripke logical relation (such as Dreyer et al. [2012]), the value relation for reference types should say that two locations \( \ell_1 \) and \( \ell_2 \) are related at type \( \text{ref} \tau \) in world \( W \) if there exists an island in \( W \) (the 4th island in \( W \), written \( W(i) \)), such that in all future worlds \( W' \):

1. the island’s heap relation \( HR \) in the current STS state \( s \) requires that the locations’ contents must be related at type \( \tau \), and
2. the pair of locations \( (\ell_1, \ell_2) \) are included in the bijection on locations (bij) tracked by that island.4

In our multi-language setting, the above interpretation for \( \mathcal{V}[\text{ref} \tau] \) does not suffice. Since the multi-language treats wrapped locations such as \( \text{ref} \mu \mathcal{C} \mathcal{M} \ell_2 \) as a value form of type \( \text{ref} \tau \), our value relation needs to be able to specify relatedness of locations across languages. But when should a value \( \ell_1 \) (a location) be related to a value \( \text{ref} \mu \mathcal{C} \mathcal{M} \ell_2 \) (a wrapped location)? Our first instinct when trying to relate such values was to instantly pull out the concrete values stored at these locations from heaps \( H_1, H_2 \) (provided by the island heap relations) and ensure that they are related. But since the contents of these locations belong to different languages, we needed to convert them into the same language so that we could talk about their relatedness in a logical relation that only relates same-language terms (as in Perconti and Ahmed). For instance, let \( H_1(\ell_1) = v_1 \) and let \( H_2(\ell_2) = v_2 \). Then, we could state our requirement that the locations’ contents be related by requiring that the values \( v_1 \) and \( \mathcal{V}(v_2) \) be related at type \( \tau \).

Unfortunately, the above strategy of defining \( \mathcal{V}[\text{ref} \tau] \) (and similarly \( \mathcal{V}[\mu \mathcal{C} \mathcal{M} \ell_2] \)) so that they translate the contents of the two locations into the same language made it impossible to prove the bridge lemma (which we present below). We won’t show here how that proof breaks, but the intuition is that the multi-language delays the translation of the value that a wrapped location points to until dereference. A semantic model of the multi-language must follow suit, mimicking such delays.

Our solution is to generalize our value, term, and continuation relations so that they allow us to relate values across languages (such that one of the values has source type \( \tau \) and the other has type \( \tau^{(\ell)} \)). Thus, we have \( \mathcal{V}[\tau, \tau^{(\ell)}] \) and \( \mathcal{V}[\tau^{(\ell)}, \tau] \), which we can use to say that two locations from different languages are related if the values they point to are related across languages. Making the logical relation more general in this way enables us to prove the bridge lemma.

Technically, we now have an \( N \)-by-\( N \) matrix of logical relations where \( N \) is the number of languages embedded in the multi-language. For our multi-language \( M+C \), \( N = 2 \). On the diagonals, we have \( \mathcal{V}[\tau, \tau] \) and \( \mathcal{V}[\tau, \tau] \). Off-diagonal, we have the cross-language versions \( \mathcal{V}[\tau, \tau^{(\ell)}] \) and \( \mathcal{V}[\tau^{(\ell)}, \tau] \). Now, the astute reader may be worried that an approach that requires an \( N \)-by-\( N \) matrix of logical relations will not scale well. Fortunately, our actual logical relation definition isn’t much more involved than a standard one because we are able to define the off-diagonals in terms of the diagonals. This can be seen in the middle section of Figure 10, which gives the definition of \( \mathcal{V}[\tau_1, \tau_2] \), both the cross-language and single-language variants.

Value Relation: Suspended Type Variables, Admissible Relations. We now discuss what properties an interpretation \( \mathcal{V}[\tau] \) of a type variable \( \alpha \) must satisfy to be considered admissible. As usual, these requirements stem from lemmas that we need about \( \mathcal{V}[\tau][\rho[s]] \). Since \( \tau = \alpha \) is a base case, these properties need to hold for any interpretation of \( \alpha \). In our multi-language setting, the two properties we need are boundary cancellation and the bridge lemmas. We discussed boundary cancellation in §4; here we give a statement of it in terms of \( \mathcal{V}[\tau][\rho] \) (with cancellation on the right, see technical appendix for details) which we need to prove admissibility.

**Lemma 7.1 (MC/CM Boundary Cancellation).** If \( (W, v_1, v_2) \in \mathcal{V}[\tau][\rho] \) and \( \rho'([MC^{(\ell)}(v_2)]) = v_2 \), then \( (W, v_1, v_2) \in \mathcal{V}[\tau][\rho'] \).
Ahmed [2014], instead of trying to do all that work in
properties between
needed properties up-front. Specifically, we do this by changing the
into some

Intuitively, the interpretation of
their backward translations are related at the corresponding source

PPDP '19, October 7–9, 2019, Porto, Portugal
Phillip Mates, Jamie Perconti, and Amal Ahmed

Figure 11: M+C Logical Relation: Heap, Continuation, and Term Relations

ValRel\(\{r_1, r_2\} \overset{\text{def}}{=} \{ \varnothing \leq \text{ValAtom}(r_1, r_2) \mid (W, v_1, v_2) \in \varnothing \}
\)  
\(\forall W' \models W \cdot (W', v_1, v_2) \in \varnothing \}

CCValRel\(\{r_1, r_2\} \overset{\text{def}}{=} \{ \varnothing \leq \text{ValRel}(r_1, r_2) \mid \text{R}[M, C] \in \text{ValRel}(r_1, r_2) \land \text{R}[C, M] \in \text{ValRel}(r_1, r_2) \}
\)

Figure 12: Logical Relation: Admissible Value Relations

The lemma for cancellation on the left is similar. Then, we have two
more boundary cancellation lemmas (similarly, on the right and
the left) for the CM/MC boundaries.

The bridge lemmas state that if two values are related at a given
type, then their translations are related at translation type. Or, in
the other direction, if two values are related at translation type,
their backward translations are related at the corresponding source
type. These lemmas are needed to prove soundness of the logical
relation for contextual equivalence. The bridge lemma stated for
the term relation is as follows:

LEMMA 7.2 (BRIDGE LEMMA). Let \(\alpha \vdash \tau\) and \(\rho \in D[\alpha]\)

1. If \((W, e_1, e_2) \in E[\bar{\cdot}][C]\) then \((W, \rho(e_1), \rho(e_1)MC e_1, \rho(e_2)MC e_2) \in E[\tau][\rho]\)
2. If \((W, e_1, e_2) \in E[\bar{\cdot}][\rho]\) then \((W, CM(\rho(e_1), \rho(e_2))) \in E[\bar{\cdot}][C][\rho]\)

A further complication is that the admissibility criteria for value
relations and the choice of how to define \(V[\alpha]\) are intertwined.
Intuitively, the interpretation of \(\alpha\) requires that we be able to
take the relation VR for \(\alpha\) and be able to “translate” the relation
into some VR for \(\alpha\), all while VR.R and VR’.R both satisfy the
boundary cancellation and bridge properties. Like Perconti and
Ahmed [2014], instead of trying to do all that work in \(V[\alpha]\), we
take a different approach and define “translations” of VR.R with
the needed properties up-front. Specifically, we do this by changing
the structure of VR.R to include not just one relation \(\rho\) on values
in the language of the type variable whose interpretation is being
given, but rather an \(i \times 1\)-by-\(i\)-1 matrix of relations where \(i\) is the
number of languages below it. Hence, as shown in Figure 12, we have a
1-by-1 matrix for the interpretation of C type variables (CCValRel)
and a 2-by-2 matrix for the interpretation of M type variables (MMValRel).
For the latter, we require that the four relations satisfy properties
between each other: translating only all the (or the

\(D[\alpha]\)  \(= \{ 0 \}\)

\(D[\alpha, \alpha] = \{ \rho(\alpha) \rightarrow VR \mid \rho \in D[\alpha] \land VR \in \text{MMValRel} \}
\)

\(D[\alpha, \alpha] = \{ \rho(\alpha) \rightarrow VR \mid \rho \in D[\alpha] \land VR \in \text{CCValRel} \}
\)

\(G[\tau][\rho] = \{ (W, \emptyset) \mid W \in \text{World} \}
\)

\(G[\Gamma, x: \tau][\rho] = \{ (W, y \mapsto (v_1, v_2)) \mid (W, y) \in G[\Gamma][\rho] \land
\)

\(\{ (W, v_1, v_2) \in V[\tau][\rho] \}
\)

\(H_1[\cdot][\rho] = \text{World} \)

\(H_1[\cdot, \tau][\rho] = H_1[\cdot] \land \{ (W, \emptyset, f) \in V[\text{ref } \tau][\rho] \}
\)

\(H_1[\cdot][\rho] = \text{World} \)

\(H_1[\cdot, \tau][\rho] = H_1[\cdot] \land \{ (W, f, \tau) \in V[\text{ref } \tau][\rho] \}
\)

\(H_1[\cdot][\rho] = \text{World} \)

\(H_1[\cdot, \tau][\rho] = H_1[\cdot] \land \{ (W, f, \tau) \in V[\text{ref } \tau][\rho] \}
\)

\(\Psi; \Delta; \Gamma \vdash e_1 \overset{\text{def}}{=} \rho(\alpha) \in \text{MMValRel}; e_2 \overset{\text{def}}{=} \Psi; \Delta; \Gamma \vdash e_2 \in \text{MMValRel}; \Delta; \Gamma \vdash e_2 \in \text{MMValRel}
\)

We now present some of the main lemmas we prove about the
logical relation, building up to the Fundamental Property.

As mentioned above, the value relation must satisfy monotonicity,
which says that if two values are related in a world \(W\), then
they must be related in all future worlds accessible from \(W\).

LEMMA 7.3 (MONOTONICITY). If \(\rho \in D[\alpha], \Delta \vdash \tau, \Delta \vdash \tau\) and

\(W' \models W, \text{then} \)
The term relation $E[\tau]$ is closed under anti-reduction, which roughly says that if $e_1$ and $e_2$ reduce to $e'_1$ and $e'_2$, respectively, then if the latter are related in $E[\tau]$, then the former must be as well.

**Lemma 7.4 (Closed under Type-Preserving Anti-Reduction).** Let $(W, e_1, e_2) \in \text{TermAtom}[\tau, \rho]$ be closed. Given $W \supseteq W'$, if $W.k \leq W'.k + k_1$, $W.k \leq W'.k + k_2$, and

$(H_1 \mid e_1) \longrightarrow^k \langle H'_1 \mid e'_1 \rangle \quad \text{and} \quad (H_2 \mid e_2) \longrightarrow^k \langle H'_2 \mid e'_2 \rangle,$

then $(W', e'_1, e'_2) \in E[\tau][\rho] \implies (W, e_1, e_2) \in E[\tau][\rho]$.

We use the Monadic Bind lemma extensively: when proving boundary cancellation, bridge lemmas, and compatibility lemmas.

**Lemma 7.5 (Monadic Bind).** If $(W, e_1, e_2) \in E[\tau][\rho]$, $(W', e_1, e_2) \in \text{ContAtom}[\tau, \rho] \land \rho \rightarrow \langle \tau, \rho' \rangle$ and

$W \equiv \text{pub} W'.(W', v_1, v_2) \in V[\tau][\rho] \implies (W', E_1[v_1], E_2[v_2]) \in E[\tau'][\rho]$,

then $(W, E_1(e_1), E_2(e_2)) \in E[\tau][\rho]$.

We also prove the boundary cancellation and bridge lemmas stated earlier and then prove that the $\forall$ reductions are admissible.

**Lemma 7.6 (Admissibility of $\forall$).** Let $\rho \in D[\Delta]$.

1. If $\Delta \vdash \tau$, then $(\rho_1(\tau), \rho_2(\tau), R) \in \text{MMValRel}$, where $R = \langle V[\tau][\rho], V[\tau'][\rho], V[\tau'][\rho] \rangle$.

2. If $\Delta \vdash \tau$, then $(\rho_1(\tau), \rho_2(\tau), \rho[\tau]) \in \text{CCValRel}$.

The proof of admissibility follows easily from monotonicity (Lemma 7.3), boundary cancellation for $\forall$ (Lemma 7.1) and the bridge lemma (Lemma 7.2).

Next, we prove the compatibility lemmas, from which we can easily prove the Fundamental Property of the logical relation.

**Lemma 7.7 (Fundamental Property).** If $\Psi; \Delta; \Gamma \vdash e : \tau$, then $\Psi; \Delta; \Gamma \vdash e \approx^\log_{M+C} \tau$

**Soundness and Completeness with respect to Contextual Equivalence.** To show that the logical relation is sound for contextual equivalence, we first prove that the logical relation is a congruence (elided) and that it is adequate (see below). Then we show soundness with respect to contextual equivalence, i.e., that logical equivalence implies contextual equivalence.

**Lemma 7.8 (Adequacy).** If $\Psi; \forall \vdash e_1 \approx^\log_{M+C} e_2 : \tau$, then $\forall : \tau 

**Lemma 7.9 (Logical Relation Sound for Contextual Equivalence).** If $\Psi; \Delta; \Gamma \vdash e_1 \approx^\text{ctx}_{M+C} e_2 : \tau$, then $\forall : \tau$

Finally, we also prove that the logical relation is complete with respect to $M+C$ contextual equivalence, i.e., that contextual equivalence implies logical equivalence.

**Lemma 7.10 (Logical Relation Complete for Contextual Equivalence).** If $\Psi; \Delta; \Gamma \vdash e_1 \approx^\text{ctx}_{M+C} e_2 : \tau$, then $\forall : \tau$

## 7.2 Correctness of Closure Conversion

Having proved our logical relation respects contextual equivalence, we can use it to prove the correctness of closure conversion, Theorem 5.2, which is stated in terms of $M+C$ contextual equivalence.

First, we prove boundary cancellation for open terms, which follows easily from the lemma for closed terms. We then use that to prove a lemma about boundary cancellation in a general context.

**Lemma 7.11 (Context Boundary Cancellation).**

1. If the hole in $C$ is $[\ ]$, then $\Psi; \Delta; \Gamma \vdash e_1 \approx^\log_{M+C} C[e_2] : \tau$, then $\Psi; \Delta; \Gamma \vdash e_1 \approx^\log_{M+C} C[e_2] : \tau$.

2. If the hole in $C$ is $[\ ]$ and $\Psi; \Delta; \Gamma \vdash e_2 : \tau$, then $\Psi; \Delta; \Gamma \vdash e_1 \approx^\log_{M+C} C[e_2] : \tau$.

3. If the hole in $C$ is $[\ ]$, then $\Psi; \Delta; \Gamma \vdash e \approx^\log_{M+C} C[\tau] : \tau$.

4. If the hole in $C$ is $[\ ]$, then $\Psi; \Delta; \Gamma \vdash e \approx^\log_{M+C} C[\tau] : \tau$.

Next, we prove correctness of closure conversion.

**Theorem 7.12 (Closure Conversion is Semantics Preserving).**

1. If $\forall : \tau \vdash e \approx^\log_{M+C} \tau$.

The proof proceeds by induction on the compiler judgment (Figure 5). Most cases of the proof make use of Lemma 7.11 as well as anti-reduction (Lemma 7.4). The most involved proof cases are the ones for function and application (see technical appendix).

## 8 Related Work and Discussion

**Compositional Compiler Correctness.** There have been several recent compositional compiler correctness results, all based on very different approaches. Most of these—unlike Percanti-Ahmed and our result—restrict linking to target code that can be expressed in the compiler’s source language.

Benten and Hut [2009] and Hut and Dreyer [2011] use a cross-language logical relation to specify “semantic equivalence” between source and target terms. They show that if a source term $s$ compiles to a target term $t$, then $s$ and $t$ are related in the source-target logical relation. Benton and Hut prove correctness of a compiler from STLC with recursion to an SECD machine, while Hut and Dreyer do so for a compiler from an ML-like language to untyped assembly. The cross-language-logical-relation approach does not scale to multi-pass compilers, so Neis et al. [2015] devised parametric inter-language simulations (PILS) to prove correctness of Pilñer, a multi-pass compiler from an ML-like language to assembly [Neis et al. 2015; Neis et al. 2018]. However, all of these cross-language-inter-language-relations approaches have a significant drawback: they only allow linking with target components that are related to some source component by the cross-language relation or PILS. In practical terms then, code produced by a PILS-verified compiler can only be linked with target code produced by either the same compiler, or a different compiler from the same source language to the same target, verified using the same PILS specifications.

Stewart et al. [2015] use interaction semantics to provide an abstract specification of interoperability between source and target
components and use it to prove compositional correctness of the CompCert C compiler [Stewart 2015]. Compositional CompCert allows linking with any target component that respects restrictions imposed by CompCert’s memory model. It’s not clear how to extend this approach to compilers whose source and target languages have different memory models as in Perconti-Ahmed and our work.

Kang et al. [2016] developed SepCompCert, which only allows linking with other components produced by the same compiler—i.e., supports separate compilation. Their goal was to demonstrate that supporting verified separate compilation requires much less proof effort than verified compositional compilation which imposes fewer restrictions on linking.

Wang et al. [2019] use contextual refinement to compositionally verify Stack-Aware CompCertX, a compiler that extends the memory model of all the CompCert languages with an abstract finite stack with uniform stack access policy. CompCertX’s final pass compiles from CompCert’s block-based model to a flat memory model. CompCertX does not allow linking with target code that cannot be expressed in the source—in fact, they augment the source language with an abstract stack.

Patterson and Ahmed [2019] present CCC, a parametrized compositional interpreter correctness theorem that they propose is the desired correctness statement for past and future compositional compilation results. They show how to instantiate CCC with Perconti-Ahmed’s multi-language compiler-correctness result so that the latter’s correctness theorem implies CCC. The same instantiation applies for our result; thus our Theorem 5.2 implies CCC.

Fully Abstract Closure Conversion. A translation is said to be fully abstract if it preserves and reflects contextual equivalence—the former is difficult to prove when the target language is more expressive than the source, as the proof requires showing that every target context that we can link with can be represented in the source language (see Patrignani et al. [2019] for a survey).

Ahmed and Blume [2008] were the first to show that typed closure conversion is fully abstract. Their source and target languages were identical: System F with recursive and existential types. New et al. [2016] show full abstraction of typed closure conversion from STLC with recursive types to a target that also has existential types and exceptions (but neither language has references as we do here). The target language has a modal type system to distinguish code with control effects from code without, and they use it to ensure—by choosing the right type translation—that compiled code is typed so that it will never be linked with code that throws an unhandled exception. For the proof, they use universal embedding to embed target types (including those inexpressible in the source language) into a universal type (i.e., recursive sum type) and define interoperability between source code and the universal type.

We conjecture that if we remove call/cc from our target language C, our translation would be fully abstract. For the proof, we would need to extend Ahmed and Blume [2008]’s technique, based on wrapper functions $W^+$ and $W^-$, that translate components of type $\tau$ to $\tau^C$ and vice versa, to support mutable references: in this case, wrapping would create proxied references. To achieve full abstraction when the target has first-class control, we would need to restrict call/cc to delimited continuations and adopt a target type system for delimited continuations. We would need to ensure that compiled code is only linked with code that captures continuations up to the boundary—i.e., continuation-capturing cannot cross into compiled code. With these changes, we could then use universal embedding to prove full abstraction.

Finally, instead of achieving full abstraction via restrictions on linking with target-level control features, we could explore extending the source language with linking types [Patterson and Ahmed 2017], a feature that allows a programmer to annotate her programs to indicate where she wishes to link with features unavailable in the source. A fully abstract compiler would ensure that linking respects those annotations. This lets programmers control the precise source-level contextual equivalence they want the compiler to preserve.

Additional Compiler Passes. As future work, we wish to extend our compiler all the way down to assembly, performing heap allocation and then code generation. For the latter pass, we can leverage the work on FunTAL, a multi-language that mixes a high-level functional language with low-level assembly [Patterson et al. 2017] and solves the challenge of making assembly compositional. Extending the compiler with additional passes requires extending our multi-language with additional languages that sit below C in the compiler. As discussed in §7, when we have a multi-language with N-languages, although we seem to need an N-by-N matrix of logical relations, we are able to define the off-diagonals for the $\nu$ relation in terms of the diagonals so this approach scales to more languages without making the $\nu$ relation unmanageable. However, the part of the logical relation that does not scale well are the admissibility criteria (see Figure 12). As our discussion in §7 notes, we need an $i+1$-by-$i+1$ matrix for the interpretation of type variables from each language, where $i$ is the number of languages below it in the compiler. Nonetheless, the admissibility criteria are quite uniform, so we should be able to systematically derive and prove them even when $i$ is large.

An alternative is to investigate a different multi-language semantics that scales better. Specifically, the problem with the admissibility criteria stems from the interpretation of suspended type variables $[\alpha]$. Substituting $\tau$ for $\alpha$ in the suspension, requires translating the type to $\tau^C$. If we could devise a compiler multi-language that treats suspended types like the opaque lump types we employ in the other direction, we think that could lead to a logical relation with simple admissibility criteria (as in the work of Scherer et al. [2018] who use lump types to pass ML values to a linear language).

A final idea for reducing proof burden when combining many compiler languages into one is to devise a multi-language where, instead of simply putting the languages together, we leverage commonalities in adjacent languages so we don’t have to deal with the same feature twice. This seems a bit like the language-independent interaction semantics idea from Compositional CompCert, but with syntactic boundaries—modelled as macros—since boundaries are needed when languages have different memory models.

ACKNOWLEDGMENTS

This material is based upon work supported by the National Science Foundation under grants CCF-1816837 and CCF-1453796. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.