Under Control: Compositionally Correct 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

ACM Reference Format:
Phillip Mates, Jamie Perconti, and Amal Ahmed. 2019. Under Control: Compositionally Correct Closure Conversion with Mutable State. In *Principles and Practice of Programming Languages 2019 (PPDP '19), October 7–9, 2019, Porto, Portugal.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3354166.3354181

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.
PPDP '19, October 7–9, 2019, Porto, Portugal
© 2019 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-7249-7/19/10…$15.00
https://doi.org/10.1145/3354166.3354181

1 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], Pilsner [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 ⅯⅯ 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 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_M 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 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 call/cc and benefit from 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.

2 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 *terms* of each language as *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.

$$
\begin{array}{rcl}
\tau & ::= & \alpha \mid \mathsf{unit} \mid \mathsf{int} \mid \forall[\overline{\alpha}].(\overline{\tau}) \to \tau \mid \exists\alpha.\tau \mid \mu\alpha.\tau \mid \mathsf{ref}\,\tau \mid \langle\overline{\tau}\rangle \\
p & ::= & + \mid - \mid * \\
v & ::= & x \mid () \mid n \mid \lambda[\overline{\alpha}](\overline{x:\tau}).e \mid \mathsf{pack}\,\langle\tau, v\rangle\,\mathsf{as}\,\exists\alpha.\tau \mid \mathsf{fold}_{\mu\alpha.\tau}\,v \\
 & & \mid \ell \mid \langle\overline{v}\rangle \\
e & ::= & v \mid v\,p\,v \mid \mathsf{if0}\,v\,e\,e \mid v[\overline{\tau}]\,\overline{v} \mid \mathsf{unpack}\,\langle\alpha, x\rangle = v\,\mathsf{in}\,e \mid \mathsf{unfold}\,v \\
 & & \mid \mathsf{new}\,v \mid v := v \mid !\,v \mid \pi_i(v) \mid \mathsf{let}\,x = e\,\mathsf{in}\,e \\
E & ::= & [\cdot] \mid \mathsf{let}\,x = E\,\mathsf{in}\,e \\
H & ::= & \cdot \mid H, \ell \mapsto v
\end{array}
$$

$\boxed{\langle H \mid e\rangle \longmapsto \langle H \mid e'\rangle}$ $\vdots$

$$
\begin{array}{rcll}
\langle H \mid E[(\lambda[\overline{\alpha}](\overline{x:\tau}).e)\,[\overline{\tau'}]\,\overline{v}]\rangle & \longmapsto & \langle H \mid E[e[\overline{\tau'}/\overline{\alpha}][\overline{v/x}]]\rangle & \\
\langle H \mid E[\mathsf{new}\,v]\rangle & \longmapsto & \langle H[\ell \mapsto v] \mid E[\ell]\rangle & \ell \notin H \\
\langle H \mid E[\ell := v]\rangle & \longmapsto & \langle H[\ell \mapsto v] \mid E[()]\rangle & \ell \in H \\
\langle H \mid E[!\,\ell]\rangle & \longmapsto & \langle H \mid E[v]\rangle & H(\ell) = v
\end{array}
$$

$\boxed{\Psi \vdash H : \Psi'}$

$$\frac{\mathrm{dom}(\Psi) \cap \mathrm{dom}(\Psi') = \emptyset \qquad \vdash \Psi' \qquad (\Psi, \Psi'); \cdot; \cdot \vdash v_1 : \Psi'(\ell_1), \ldots, (\Psi, \Psi'); \cdot; \cdot \vdash v_n : \Psi'(\ell_n)}{\vdash \{\ell_1 \mapsto v_1, \ldots, \ell_n \mapsto v_n\} : \Psi'}$$

$\boxed{\Psi; \Gamma; \Delta \vdash e : \tau}$ where $\Psi ::= \cdot \mid \Psi, \ell : \tau$ and $\Gamma ::= \cdot \mid \Gamma, \alpha$ and $\Delta ::= \cdot \mid \Delta, x : \tau$

$$\frac{x : \tau \in \Delta}{\Psi; \Gamma; \Delta \vdash x : \tau} \qquad \frac{\Psi(\ell) = \tau}{\Psi; \Gamma; \Delta \vdash \ell : \mathsf{ref}\,\tau} \qquad \frac{\Psi; \Gamma, \overline{\alpha}; \Delta, \overline{x:\tau} \vdash e : \tau'}{\Psi; \Gamma; \Delta \vdash \lambda[\overline{\alpha}](\overline{x:\tau}).e : \forall[\overline{\alpha}].(\overline{\tau}) \to \tau'}$$

$$\frac{\Psi; \Gamma; \Delta \vdash v_0 : \forall[\overline{\alpha}].(\overline{\tau}) \to \tau' \qquad \Gamma \vdash \overline{\tau_0} \qquad \Psi; \Gamma; \Delta \vdash \overline{v} : \overline{\tau[\tau_0/\alpha]}}{\Psi; \Gamma; \Delta \vdash v_0\,[\overline{\tau_0}]\,\overline{v} : \tau'[\overline{\tau_0/\alpha}]}$$

$$\frac{\Psi; \Gamma; \Delta \vdash v : \tau}{\Psi; \Gamma; \Delta \vdash \mathsf{new}\,v : \mathsf{ref}\,\tau} \qquad \frac{\Psi; \Gamma; \Delta \vdash v_1 : \mathsf{ref}\,\tau \qquad \Psi; \Gamma; \Delta \vdash v_2 : \tau}{\Psi; \Gamma; \Delta \vdash v_1 := v_2 : \mathsf{unit}} \qquad \frac{\Psi; \Gamma; \Delta \vdash v : \mathsf{ref}\,\tau}{\Psi; \Gamma; \Delta \vdash !\,v : \tau}$$

Figure 2: Source Language M: Syntax & Semantics (excerpt)

We start with some notes about typesetting and notational conventions. We typeset the source language M in blue, the target language C in **red bold**, and later in the paper, the multi-language M+C in black. For each of our languages, we use the metavariable τ for types, e for terms or *components*, v for values, ℓ for locations, H for heaps, E for evaluation contexts, and C for general contexts. We write $\mathrm{fv}(e)$ to denote the free term variables of e and $\mathrm{ftv}(e)$ (or $\mathrm{ftv}(\tau)$) to denote the free type variables of e (or of type τ). We use a line above a syntactic element to indicate a list of repeated instances of this element, e.g., $\overline{\alpha} = \alpha_1, \ldots, \alpha_n$ for $n \geq 0$. When the arities of different lists are required to match up in a definition or inference rule, these constraints will usually be obvious from context. Whenever two environments (e.g. Ψ or Δ or Γ) are joined by a comma, this should be interpreted as a disjoint union.

Source Language: M. The source language M is call-by-value System F with dynamically allocated, ML-style mutable references, existential types, recursive types, and tuples. Figure 2 presents the complete syntax along with excerpts of the static and dynamic semantics. We combine type- and term-level abstractions of arbitrary arity into a single binding form $\forall[\overline{\alpha}].(\overline{\tau}) \to \tau'$, abbreviating $\forall[].(\overline{\tau}) \to \tau'$ as $(\overline{\tau}) \to \tau'$. A program configuration in M, written $\langle H \mid e\rangle$ is a pair of a heap H and a closed term e. A heap H is a

mapping from locations ℓ 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 $\bar{}; \acute{}; \grave{} \vdash e : \tau$. The heap type $\bar{}$ tracks the types τ of the contents of heap locations ℓ in scope, where τ must be a closed type. The type environment $\acute{}$ tracks the type variables α in scope. The value environment $\grave{}$ tracks the term variables x in scope along with their types τ, which must be well formed under $\acute{}$ (written $\acute{} \vdash \tau$ and defined as $\mathrm{ftv}(\tau) \subseteq \acute{}$). 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 $\bar{} \vdash H : \bar{}'$, which says that the heap fragment H is assigned the heap type $\bar{}'$ under the assumption that there is some (external) heap type $\bar{}$ with locations that may be referenced by values stored in H. Here $\bar{}'$ 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 $(\bar{}, \bar{}')$.

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 languages into a multi-language system whose heap H is a pair $(H, \mathbf{H})$ of source and target heaps, each of whose contents may refer to locations in the other.

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 $\mathbf{v}[\tau]$.

The shaded parts of Figure 3 extend the base language with first-class control. First-class continuations are represented by $\mathbf{cont}_\tau\ \mathbf{E}$, a value form that injects evaluation contexts **E** (with a hole of type τ) into the term language. The type $\mathbf{cont}\ \tau$ is the type ascribed to first-class continuations $\mathbf{cont}_\tau\ \mathbf{E}$ that expect a value of type τ. 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 $\vdash \mathbf{C} : (\bar{}; \acute{}; \grave{} \vdash \tau) \rightsquigarrow (\bar{}'; \acute{}'; \grave{}' \vdash \tau')$. Context typing ensures that, given any term **e** that satisfies the type of the hole, $\bar{}; \acute{}; \grave{} \vdash \mathbf{e} : \tau$, we have that $\bar{}'; \acute{}'; \grave{}' \vdash \mathbf{C[e]} : \tau'$.[1] We extend the base language

[1]The grammar for general contexts is essentially standard except that, since our language is in monadic normal form, we have to be careful about when a hole can be plugged with an expression and when it must be plugged with a value. We discuss this further in §5. Note that this does not matter for evaluation contexts since their holes accept expressions.

$$
\begin{array}{rcl}
\tau & ::= & \alpha \mid \mathbf{unit} \mid \mathbf{int} \mid \forall[\overline{\alpha}].(\overline{\tau}) \to \tau \mid \exists \alpha.\tau \mid \mu\alpha.\tau \mid \mathbf{ref}\ \tau \\
& & \mid \langle \overline{\tau} \rangle \mid \mathbf{cont}\ \tau \\
p & ::= & + \mid - \mid * \\
v & ::= & x \mid () \mid n \mid \lambda[\overline{\alpha}](\overline{x:\tau}).e \mid v[\tau] \mid \mathbf{pack}\ \langle \tau, v \rangle\ \mathbf{as}\ \exists\alpha.\tau \\
& & \mid \mathbf{fold}_{\mu\alpha.\tau}\ v \mid \ell \mid \langle \overline{v} \rangle \mid \mathbf{cont}_\tau\ E \\
e & ::= & v \mid v\ p\ v \mid \mathbf{if0}\ v\ e\ e \mid v\,[]\,\overline{v} \mid \mathbf{unpack}\ \langle \alpha, x \rangle = v\ \mathbf{in}\ e \\
& & \mid \mathbf{unfold}\ v \mid \mathbf{new}\ v \mid v := v \mid !\,v \mid \pi_i(v) \mid \mathbf{let}\ x = e\ \mathbf{in}\ e \\
& & \mid \mathbf{call/cc}_\tau(x.\,e) \mid \mathbf{throw}_\tau\ v\ \mathbf{to}\ v \\
E & ::= & [\cdot] \mid \mathbf{let}\ x = E\ \mathbf{in}\ e \\
H & ::= & \cdot \mid H, \ell \mapsto v
\end{array}
$$

$\boxed{\langle H \mid e \rangle \longmapsto \langle H \mid e' \rangle}$ ⋮

$$\langle H \mid E[(\lambda[\overline{\alpha}](\overline{x:\tau}).e)\,[\overline{\tau'}]\,\overline{v}] \rangle \longmapsto \langle H \mid E[e[\overline{\tau'}/\overline{\alpha}][\overline{v}/\overline{x}]] \rangle$$

$$\langle H \mid E[\mathbf{call/cc}_\tau(x.\,e)] \rangle \longmapsto \langle H \mid E[e[\mathbf{cont}_\tau\ E/x]] \rangle$$

$$\langle H \mid E[\mathbf{throw}_\tau\ v\ \mathbf{to}\ \mathbf{cont}_\tau\ E'] \rangle \longmapsto \langle H \mid E'[v] \rangle$$

$\boxed{\bar{} \vdash H : \bar{}'}$ (analogous to M rule for heap fragments)

$\boxed{\bar{}; \acute{}; \grave{} \vdash e : \tau}$ and $\boxed{\bar{}; \acute{}; \grave{} \vdash E \div \tau}$

where $\bar{} ::= \cdot \mid \bar{}, \ell : \tau$ and $\acute{} ::= \cdot \mid \acute{}, \alpha$ and $\grave{} ::= \cdot \mid \grave{}, x : \tau$

$$\frac{x : \tau \in \grave{}}{\bar{}; \acute{}; \grave{} \vdash x : \tau} \qquad \frac{\bar{}; \overline{\alpha}; \overline{x:\tau} \vdash e : \tau'}{\bar{}; \acute{}; \grave{} \vdash \lambda[\overline{\alpha}](\overline{x:\tau}).e : \forall[\overline{\alpha}].(\overline{\tau}) \to \tau'}$$

$$\frac{\bar{}; \acute{}; \grave{} \vdash v : \forall[\beta, \overline{\alpha}].(\overline{\tau}) \to \tau' \qquad \acute{} \vdash \tau_0}{\bar{}; \acute{}; \grave{} \vdash v[\tau_0] : \forall[\overline{\alpha}].(\overline{\tau[\tau_0/\beta]}) \to \tau'[\tau_0/\beta]}$$

$$\frac{\bar{}; \acute{}; \grave{} \vdash v : \forall[].(\overline{\tau}) \to \tau' \qquad \bar{}; \acute{}; \grave{} \vdash \overline{v} : \overline{\tau}}{\bar{}; \acute{}; \grave{} \vdash v\,[]\,\overline{v} : \tau'}$$

$$\frac{\bar{}; \acute{}; \grave{}, x : \mathbf{cont}\ \tau \vdash e : \tau}{\bar{}; \acute{}; \grave{} \vdash \mathbf{call/cc}_\tau(x.\,e) : \tau} \qquad \frac{\bar{}; \acute{}; \grave{} \vdash v' : \tau' \qquad \bar{}; \acute{}; \grave{} \vdash v : \mathbf{cont}\ \tau'}{\bar{}; \acute{}; \grave{} \vdash \mathbf{throw}_\tau\ v'\ \mathbf{to}\ v : \tau}$$

$$\frac{\bar{}; \acute{}; \grave{} \vdash E \div \tau}{\bar{}; \acute{}; \grave{} \vdash \mathbf{cont}_\tau\ E : \mathbf{cont}\ \tau} \qquad \frac{\vdash E : (\bar{}; \acute{}; \grave{} \vdash \tau) \rightsquigarrow (\bar{}; \acute{}; \grave{} \vdash \tau')}{\bar{}; \acute{}; \grave{} \vdash E \div \tau}$$

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

with a continuation typing judgment $\bar{}; \acute{}; \grave{} \vdash E \div \tau$ that says that **E** is an evaluation context with a hole of type τ. We need the latter to type check first-class continuations $\mathbf{cont}_\tau\ \mathbf{E}$.

We also add **call/cc** and **throw** with the standard operational semantics and typing rules. As usual, $\mathbf{call/cc}_\tau(\mathbf{x.\,e})$ captures its current continuation **E**, binds it to **x**, and continues with $\mathbf{E[e[cont}_\tau\ \mathbf{E/x]]}$. Meanwhile, $\mathbf{throw}_\tau\ \mathbf{v}\ \mathbf{to}\ \mathbf{cont}_\tau\ \mathbf{E'}$ throws away its current continuation **E** and continues with $\mathbf{E'[v]}$.

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*

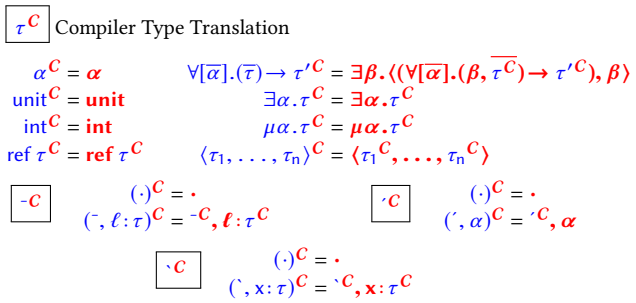

Figure 4: Closure Conversion: Type Translation

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 functions with the same type but different free variables may end up with differently typed closure environments. As an example, consider two functions λx. y + 1 and λx. x of type int $\to$ int, where y is a free variable of type int. The function part of their translations would have types $(\langle\text{int}\rangle, \text{int}) \to \text{int}$ and $(\langle\rangle, \text{int}) \to \text{int}$, 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-erasure 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 τ to C terms of type τ^C. Figure 4 presents the type translation τ^C. The only interesting case is that for function types which are transformed into an existential type where β represents the (abstract) type of the closure environment. Figure 4 also presents the translation of environments ^-C, $'C$, and $`C$, 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 $\ell : \text{ref}\ \tau$ into a C location $\ell : \text{ref}\ \tau^C$. 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 **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 ${}^\tau\mathcal{MC}$ and $\mathcal{CM}^\tau$ to allow cross-language communication. The term ${}^\tau\mathcal{MC}\ \mathbf{e}$ (C inside, M outside) allows a term $\mathbf{e}$ of target type τ^C to be used as a term of source type τ, while the term $\mathcal{CM}^\tau\ \mathbf{e}$ (M inside, C outside) allows a term $\mathbf{e}$ of source type τ to be used as a target term of translation type τ^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 $\mathcal{CM}^\tau\ \mathbf{e}$ has type τ^C if $\mathbf{e}$ has type τ and to evaluate it we first reduce $\mathbf{e}$ to a value $\mathbf{v}$ (using M reduction rules). Then a type-directed meta-function called the *value translation* is applied to $\mathbf{v}$, yielding a value $\mathbf{v}$ in C of type τ^C (written $\mathbf{CM}^\tau(\mathbf{v}) = \mathbf{v}$). 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 $\mathbf{n}$ of type int is converted to the same integer $\mathbf{n}$ 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.

$$\mathbf{CM}^{\tau \to \tau'}(\mathbf{v}) = \mathbf{pack}\ \langle \mathbf{unit}, \langle \mathbf{v}, ()\rangle\rangle\ \mathbf{as}\ \exists\beta.\langle((\beta, \tau^C) \to \tau'^C), \beta\rangle$$
$$\text{where } \mathbf{v} = \lambda(\mathbf{z} : \mathbf{unit}, \mathbf{x} : \tau^C).\mathcal{CM}^{\tau'}\ (\mathbf{v}\ {}^\tau\mathcal{MC}\ \mathbf{x})$$
$${}^{\tau \to \tau'}\mathbf{MC}(\mathbf{v}) =$$
$$\lambda(\mathbf{x} : \tau).{}^{\tau'}\mathcal{MC}\ (\mathbf{unpack}\ \langle\beta, \mathbf{y}\rangle = \mathbf{v}\ \mathbf{in}\ \pi_1(\mathbf{y})\ \pi_2(\mathbf{y})\ \mathcal{CM}^\tau\ \mathbf{x})$$

Next, consider the type $\forall[\alpha].(\alpha) \to \alpha$. Since $\alpha^C = \alpha$, the translation of this type is $\exists\beta.\langle(\forall[\alpha].(\beta, \alpha) \to \alpha), \beta\rangle$. Naively value-translating a value $\mathbf{v}$ of this type, we get:

$${}^{\forall[\alpha].(\alpha) \to \alpha}\mathbf{MC}(\mathbf{v}) =$$
$$\lambda[\alpha](\mathbf{x} : \alpha).{}^\alpha\mathcal{MC}\ (\mathbf{unpack}\ \langle\beta, \mathbf{y}\rangle = \mathbf{v}\ \mathbf{in}\ \pi_1(\mathbf{y})\ [\alpha^C]\ \pi_2(\mathbf{y})\ \mathcal{CM}^\alpha\ \mathbf{x})$$

But note that $\alpha^C = \alpha$, which means α would be unbound! Perconti and Ahmed point out that what we want instead is that when α is instantiated with a concrete type τ, the positions inside language C where that type is needed receive τ^C. To this end, they add a type $\lceil\alpha\rceil$ ("α suspended in C") that allows an M type variable to appear in a C type. The M type variable α needs to be translated, but the translation is *delayed* until α is instantiated with a concrete type. This is enforced in the definition of type substitution: $\lceil\alpha\rceil[\tau/\alpha] = \tau^C$. In addition, they define a modified type translation—the *boundary type translation* $\tau^{\langle C\rangle}$—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

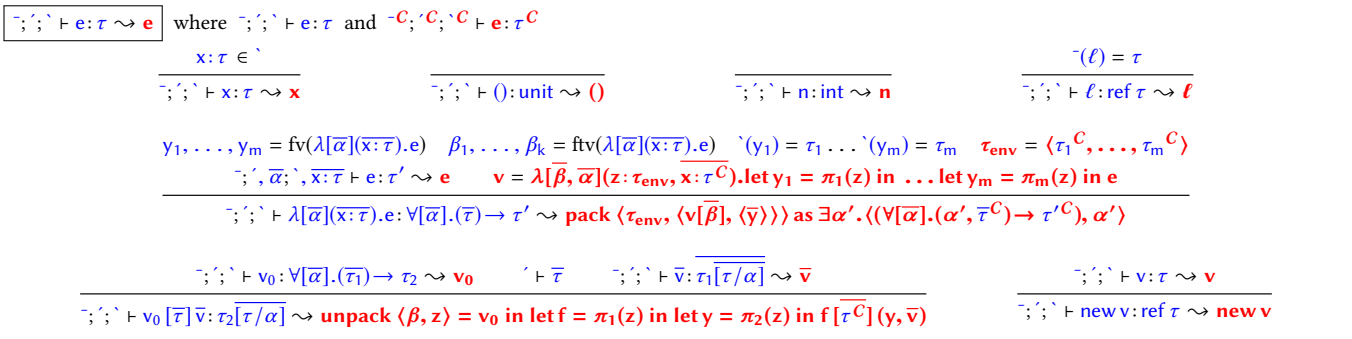

Figure 5: Closure Conversion: Term Translation (excerpt)

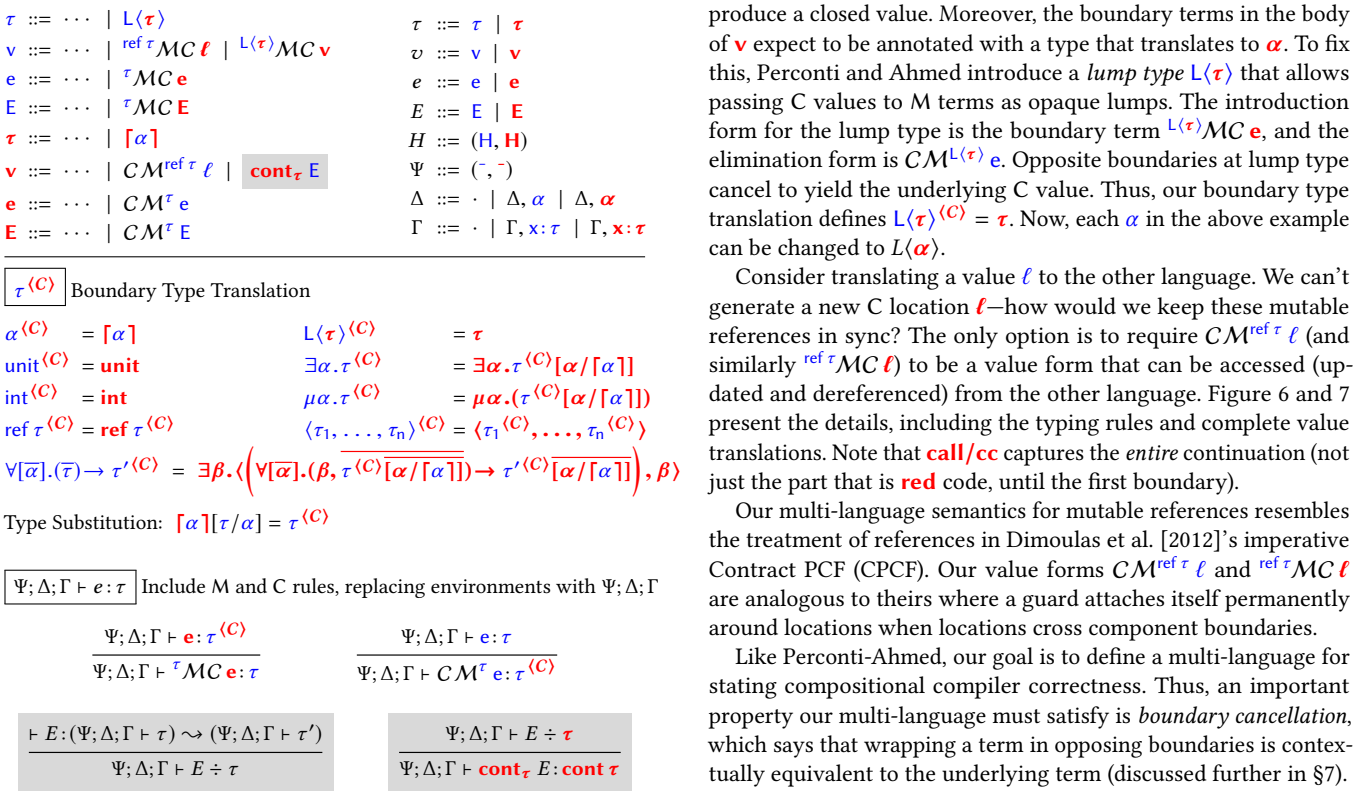

Figure 6: M+C: Syntax & Static Semantics (extends Figs 2, 3)

by the rule $\alpha^{\langle C \rangle} = \lceil \alpha \rceil$ in the boundary type translation. Since we only want to suspend *free* type variables, when translating a type that contains bound variables we must restore the behavior of the compiler's type translation when translating the binding position: e.g., $(\exists \alpha.\tau)^{\langle C \rangle} = \exists \boldsymbol{\alpha}.(\tau^{\langle C \rangle}[\boldsymbol{\alpha}/\lceil \alpha \rceil])$. Thus the boundary type translation preserves the binding structure of the original type.

An additional issue arises if we naïvely translate values of type $\forall[\alpha].(\alpha) \to \alpha$ from M into C:

$$\mathbf{CM}^{\forall[\alpha].(\alpha)\to\alpha}(\mathbf{v}) = \mathbf{pack}\ \langle \mathbf{unit}, \langle \mathbf{v}, () \rangle \rangle\ \mathbf{as}\ (\forall[\alpha].(\alpha) \to \alpha)^{\langle C \rangle}$$
$$\text{where } \mathbf{v} = \boldsymbol{\lambda}[\boldsymbol{\alpha}](\mathbf{z}:\mathbf{unit}, \mathbf{x}:\boldsymbol{\alpha}).C\mathcal{M}^{\alpha}\ (\mathbf{v}\,[\alpha]\,{}^{\alpha}\mathcal{MC}\,\mathbf{x})$$

Translating the binder for α into a C binder for $\boldsymbol{\alpha}$ leaves free occurrences of α in the result, which is a problem since we must produce a closed value. Moreover, the boundary terms in the body of $\mathbf{v}$ expect to be annotated with a type that translates to $\boldsymbol{\alpha}$. To fix this, Perconti and Ahmed introduce a *lump type* $\mathsf{L}\langle\tau\rangle$ that allows passing C values to M terms as opaque lumps. The introduction form for the lump type is the boundary term ${}^{\mathsf{L}\langle\tau\rangle}\mathcal{MC}\,\mathbf{e}$, and the elimination form is $C\mathcal{M}^{\mathsf{L}\langle\tau\rangle}\,\mathbf{e}$. Opposite boundaries at lump type cancel to yield the underlying C value. Thus, our boundary type translation defines $\mathsf{L}\langle\tau\rangle^{\langle C \rangle} = \tau$. Now, each α in the above example can be changed to $L\langle\boldsymbol{\alpha}\rangle$.

Consider translating a value ℓ to the other language. We can't generate a new C location $\boldsymbol{\ell}$—how would we keep these mutable references in sync? The only option is to require $C\mathcal{M}^{\mathsf{ref}\,\tau}\,\ell$ (and similarly ${}^{\mathsf{ref}\,\tau}\mathcal{MC}\,\boldsymbol{\ell}$) to be a value form that can be accessed (updated 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 $C\mathcal{M}^{\mathsf{ref}\,\tau}\,\ell$ and ${}^{\mathsf{ref}\,\tau}\mathcal{MC}\,\boldsymbol{\ell}$ 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-conversion 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 of the right *outermost* color—i.e., a term

$\boxed{\mathbf{CM}^{\tau}(\mathsf{v}) = \mathbf{v}}$ Value Translation

$$\mathbf{CM}^{\mathsf{unit}}(()) = () \qquad \mathbf{CM}^{\mathsf{int}}(\mathsf{n}) = \mathbf{n} \qquad \mathbf{CM}^{\mathsf{L}\langle\tau\rangle}(^{\mathsf{L}\langle\tau\rangle}\mathcal{MC}\,\mathsf{v}) = \mathbf{v}$$

$$\mathbf{CM}^{\forall[\overline{\alpha}].(\overline{\tau})\to\tau'}(\mathsf{v}) = \mathbf{pack}\,\langle \mathbf{unit}, \langle \mathbf{v}, ()\rangle\rangle\,\mathbf{as}\,(\forall[\overline{\alpha}].(\overline{\tau})\to\tau')^{\langle C\rangle} \quad \text{where } \mathbf{v} = \lambda[\overline{\alpha}](\mathbf{z}:\mathbf{unit}, \overline{\mathbf{x}:\tau^{\langle C\rangle}[\alpha/\lceil\alpha\rceil]}).C\mathcal{M}^{\tau'[\mathsf{L}\langle\alpha\rangle/\alpha]}\,(\mathsf{v}\,\overline{[\mathsf{L}\langle\alpha\rangle]}\,\overline{\tau[\mathsf{L}\langle\alpha\rangle/\alpha]}\mathcal{MC}\,\mathbf{x})$$

$$\mathbf{CM}^{\exists\alpha.\tau}(\mathsf{pack}\,\langle\tau',\mathsf{v}\rangle\,\mathsf{as}\,\exists\alpha.\tau) = \mathbf{pack}\,\langle\tau'^{\langle C\rangle}, \mathbf{v}\rangle\,\mathbf{as}\,\exists\alpha.\tau^{\langle C\rangle} \quad \text{where } \mathbf{CM}^{\tau[\tau'/\alpha]}(\mathsf{v}) = \mathbf{v}$$

$$\mathbf{CM}^{\mu\alpha.\tau}(\mathsf{fold}_{\mu\alpha.\tau}\,\mathsf{v}) = \mathbf{fold}_{\mu\alpha.\tau^{\langle C\rangle}}\,\mathbf{v} \quad \text{where } \mathbf{CM}^{\tau[\mu\alpha.\tau/\alpha]}(\mathsf{v}) = \mathbf{v}$$

$$\mathbf{CM}^{\mathbf{ref}\,\tau}(\ell) = C\mathcal{M}^{\mathbf{ref}\,\tau}\,\ell$$

$$\mathbf{CM}^{\mathbf{ref}\,\tau}(^{\mathbf{ref}\,\tau}\mathcal{MC}\,\mathbf{v}) = \mathbf{v}$$

$$\mathbf{CM}^{\langle\tau_1,\ldots,\tau_n\rangle}(\langle\mathsf{v}_1,\ldots,\mathsf{v}_n\rangle) = \langle\mathbf{v_1},\ldots,\mathbf{v_n}\rangle \quad \text{where } \mathbf{CM}^{\tau_i}(\mathsf{v}_i) = \mathbf{v_i}$$

$\boxed{{}^{\tau}\mathbf{MC}(\mathsf{v}) = \mathsf{v}}$ Value Translation

$$^{\mathsf{unit}}\mathbf{MC}(()) = () \qquad ^{\mathsf{int}}\mathbf{MC}(\mathbf{n}) = \mathsf{n} \qquad ^{\mathsf{L}\langle\tau\rangle}\mathbf{MC}(\mathbf{v}) = {}^{\mathsf{L}\langle\tau\rangle}\mathcal{MC}\,\mathsf{v}$$

$$^{\forall[\overline{\alpha}].(\overline{\tau})\to\tau'}\mathbf{MC}(\mathbf{v}) = \lambda[\overline{\alpha}](\overline{\mathsf{x}:\tau}).^{\tau'}\mathcal{MC}\,(\mathbf{unpack}\,\langle\boldsymbol{\beta},\mathbf{y}\rangle = \mathbf{v}\,\mathbf{in}\,\boldsymbol{\pi}_1(\mathbf{y})\,\overline{[\lceil\alpha\rceil]}\,\boldsymbol{\pi}_2(\mathbf{y}),\overline{C\mathcal{M}^{\tau}\,\mathsf{x}})$$

$$^{\exists\alpha.\tau}\mathbf{MC}(\mathbf{pack}\,\langle\tau',\mathbf{v}\rangle\,\mathbf{as}\,\exists\alpha.\tau^{\langle C\rangle}) = \mathsf{pack}\,\langle\mathsf{L}\langle\tau'\rangle,\mathsf{v}\rangle\,\mathsf{as}\,\exists\alpha.\tau \quad \text{where } {}^{\tau[\mathsf{L}\langle\tau'\rangle/\alpha]}\mathbf{MC}(\mathbf{v}) = \mathsf{v}$$

$$^{\mu\alpha.\tau}\mathbf{MC}(\mathbf{fold}_{\mu\alpha.\tau^{\langle C\rangle}}\,\mathbf{v}) = \mathsf{fold}_{\mu\alpha.\tau}\,\mathsf{v} \quad \text{where } {}^{\tau[\mu\alpha.\tau/\alpha]}\mathbf{MC}(\mathbf{v}) = \mathsf{v}$$

$$^{\mathbf{ref}\,\tau}\mathbf{MC}(\boldsymbol{\ell}) = {}^{\mathbf{ref}\,\tau}\mathcal{MC}\,\boldsymbol{\ell}$$

$$^{\mathbf{ref}\,\tau}\mathbf{MC}(C\mathcal{M}^{\mathbf{ref}\,\tau}\,\mathsf{v}) = \mathsf{v}$$

$$^{\langle\tau_1,\ldots,\tau_n\rangle}\mathbf{MC}(\langle\mathbf{v_1},\ldots,\mathbf{v_n}\rangle) = \langle\mathsf{v}_1,\ldots,\mathsf{v}_n\rangle \quad \text{where } {}^{\tau_i}\mathbf{MC}(\mathbf{v_i}) = \mathsf{v}_i$$

$\boxed{\langle H \mid e\rangle \longmapsto \langle H' \mid e'\rangle}$ Lift the M and C reduction rules to the new configuration—with heaps (H, H)—and replace evaluation contexts E and E with E

$$\frac{\mathbf{CM}^{\tau}(\mathsf{v}) = \mathbf{v} \qquad C\mathcal{M}^{\tau}\,\mathsf{v} \text{ not a value}}{\langle H \mid E[C\mathcal{M}^{\tau}\,\mathsf{v}]\rangle \longmapsto \langle H' \mid E[\mathbf{v}]\rangle} \qquad \frac{{}^{\tau}\mathbf{MC}(\mathsf{v}) = \mathsf{v} \qquad {}^{\tau}\mathcal{MC}\,\mathbf{v} \text{ not a value} \qquad \tau \neq \mathsf{L}\langle\tau\rangle}{\langle H \mid E[{}^{\tau}\mathcal{MC}\,\mathbf{v}]\rangle \longmapsto \langle H' \mid E[\mathsf{v}]\rangle}$$

$$\begin{array}{lcll}
\langle H \mid E[!\,(C\mathcal{M}^{\mathbf{ref}\,\tau}\,\mathsf{v})]\rangle & \longmapsto & \langle H \mid E[C\mathcal{M}^{\tau}\,!\,\mathsf{v}]\rangle & C\mathcal{M}^{\mathbf{ref}\,\tau}\,\mathsf{v} \text{ a value} \\
\langle H \mid E[!\,(^{\mathbf{ref}\,\tau}\mathcal{MC}\,\mathbf{v})]\rangle & \longmapsto & \langle H \mid E[{}^{\tau}\mathcal{MC}\,!\,\mathbf{v}]\rangle & {}^{\mathbf{ref}\,\tau}\mathcal{MC}\,\mathbf{v} \text{ a value} \\
\langle H \mid E[(C\mathcal{M}^{\mathbf{ref}\,\tau}\,\mathsf{v}) := \mathbf{v}]\rangle & \longmapsto & \langle H \mid E[C\mathcal{M}^{\mathsf{unit}}\,(\mathsf{v} := {}^{\tau}\mathcal{MC}\,\mathbf{v})]\rangle & C\mathcal{M}^{\mathbf{ref}\,\tau}\,\mathsf{v} \text{ a value} \\
\langle H \mid E[(^{\mathbf{ref}\,\tau}\mathcal{MC}\,\mathbf{v}) := \mathsf{v}]\rangle & \longmapsto & \langle H \mid E[{}^{\mathsf{unit}}\mathcal{MC}\,(\mathbf{v} := C\mathcal{M}^{\tau}\,\mathsf{v})]\rangle & {}^{\mathbf{ref}\,\tau}\mathcal{MC}\,\mathbf{v} \text{ a value} \\
\langle H \mid E[\mathbf{call/cc}_{\tau}(\mathbf{x}.\,\mathbf{e})]\rangle & \longmapsto & \langle H \mid E[\mathbf{e}[\mathbf{cont}_{\tau}\,E/\mathbf{x}]]\rangle & \\
\langle H \mid E[\mathbf{throw}_{\tau}\,\mathbf{v}\,\mathbf{to}\,\mathbf{cont}_{\tau'}\,E']\rangle & \longmapsto & \langle H \mid E'[\mathsf{v}]\rangle &
\end{array}$$

Figure 7: M+C: Dynamic Semantics (extends Figs 2 and 3)

$$\begin{array}{rcl}
C^{\mathsf{v}} & ::= & [\cdot]^{\mathsf{v}} \mid \lambda[\overline{\alpha}](\overline{\mathsf{x}:\tau}).C \mid \mathsf{pack}\,\langle\tau, C^{\mathsf{v}}\rangle\,\mathsf{as}\,\exists\alpha.\tau \mid \ldots \\
C & ::= & [\cdot] \mid C^{\mathsf{v}} \mid \ldots \mid C^{\mathsf{v}}\,[\overline{\tau}]\,\overline{\mathsf{v}} \mid \mathsf{v}\,[\overline{\tau}]\,\overline{\mathsf{v}}\,C^{\mathsf{v}}\,\overline{\mathsf{v}} \\
 & & \mid \mathsf{unpack}\,\langle\alpha,\mathsf{x}\rangle = C^{\mathsf{v}}\,\mathsf{in}\,\mathsf{e} \mid \mathsf{unpack}\,\langle\alpha,\mathsf{x}\rangle = \mathsf{v}\,\mathsf{in}\,C \mid \ldots \\
 & & \mid \mathsf{let}\,\mathsf{x} = C\,\mathsf{in}\,\mathsf{e} \mid \mathsf{let}\,\mathsf{x} = \mathsf{e}\,\mathsf{in}\,C \mid {}^{\tau}\mathcal{MC}\,C \\
\mathbf{C^{v}} & ::= & \mathbf{[\cdot]^{v}} \mid \boldsymbol{\lambda[\overline{\alpha}](\overline{x:\tau}).C} \mid \mathbf{pack}\,\langle\boldsymbol{\tau}, \mathbf{C^{v}}\rangle\,\mathbf{as}\,\boldsymbol{\exists\alpha.\tau} \mid \ldots \\
 & & \mid \mathbf{throw}_{\tau}\,\mathbf{C^{v}}\,\mathbf{to}\,\mathbf{v} \mid \mathbf{throw}_{\tau}\,\mathbf{v}\,\mathbf{to}\,\mathbf{C^{v}} \\
\mathbf{C} & ::= & \mathbf{[\cdot]} \mid \mathbf{C^{v}} \mid \mathbf{C^{v}}\,\mathbf{[]}\,\overline{\mathbf{v}} \mid \mathbf{v}\,\mathbf{[]}\,\overline{\mathbf{v}}\,\mathbf{C^{v}}\,\overline{\mathbf{v}} \mid \mathbf{unpack}\,\langle\boldsymbol{\alpha},\mathbf{x}\rangle = \mathbf{C^{v}}\,\mathbf{in}\,\mathbf{e} \\
 & & \mid \mathbf{unpack}\,\langle\boldsymbol{\alpha},\mathbf{x}\rangle = \mathbf{v}\,\mathbf{in}\,\mathbf{C} \mid \ldots \mid \mathbf{let}\,\mathbf{x} = \mathbf{C}\,\mathbf{in}\,\mathbf{e} \\
 & & \mid \mathbf{let}\,\mathbf{x} = \mathbf{e}\,\mathbf{in}\,\mathbf{C} \mid C\mathcal{M}^{\tau}\,\mathbf{C} \\
C & ::= & \mathsf{C} \mid \mathbf{C}
\end{array}$$

$\boxed{\vdash C : (\Psi;\Delta;\Gamma \vdash \tau) \rightsquigarrow (\Psi';\Delta';\Gamma' \vdash \tau')}$ Context Typing

$$\frac{\Psi \subseteq \Psi' \qquad \Delta \subseteq \Delta' \qquad \Gamma \subseteq \Gamma'}{\vdash [\cdot] : (\Psi;\Delta;\Gamma \vdash \tau) \rightsquigarrow (\Psi';\Delta';\Gamma' \vdash \tau)} \qquad \frac{\vdash \mathbf{C} : (\Psi;\Delta;\Gamma \vdash \tau) \rightsquigarrow (\Psi';\Delta';\Gamma' \vdash \tau^{\langle C\rangle})}{\vdash {}^{\tau}\mathcal{MC}\,\mathbf{C} : (\Psi;\Delta;\Gamma \vdash \tau) \rightsquigarrow (\Psi';\Delta';\Gamma' \vdash \tau)}$$

Figure 8: M+C General Contexts (excerpt)

whose outermost layer is the same language as the hole. Hence, we have both blue and **red** holes. Second, since our languages are in monadic normal form, certain holes can only be plugged with value forms, not with arbitrary terms. Hence, we have holes $[\cdot]^{\mathsf{v}}$ and $\mathbf{[\cdot]^{v}}$ which accept only values. Figure 8 presents an excerpt of the syntax of general contexts, along with the typing judgment for contexts and typing rules for a hole and a boundary form. The remaining syntax and typing rules are entirely standard and given in the supplementary material.

Contextual equivalence for M+C (written $\Psi;\Delta;\Gamma \vdash e_1 \approx^{ctx}_{\mathsf{M+C}} e_2 : \tau$) is defined below in the standard way. It says that two components e_1 and e_2 are contextually equivalent if (1) they both have type τ under environments Ψ, Δ, Γ, and (2) for any well typed context C—with a hole that accepts components typed τ under Ψ, Δ, Γ yielding a closed program that may be run with a heap of type Ψ'—the programs $C[e_1]$ and $C[e_2]$ co-terminate when run in any initial heap H of type Ψ'.

Definition 5.1 (M+C Contextual Equivalence).

$$\begin{array}{l}
\Psi;\Delta;\Gamma \vdash e_1 \approx^{ctx}_{\mathsf{M+C}} e_2 : \tau \stackrel{\text{def}}{=} \Psi;\Delta;\Gamma \vdash e_1 : \tau \;\wedge\; \Psi;\Delta;\Gamma \vdash e_2 : \tau \;\wedge \\
\qquad \forall C, \Psi', \tau'.\, \vdash C : (\Psi;\Delta;\Gamma \vdash \tau) \rightsquigarrow (\Psi';\cdot;\cdot \vdash \tau') \wedge \vdash H : \Psi' \\
\qquad\qquad \Longrightarrow (\langle H \mid C[e_1]\rangle \Downarrow \iff \langle H \mid C[e_2]\rangle \Downarrow)
\end{array}$$

In the above definition, notice that the hole in the context C may be any one of: $[\cdot]^{\mathsf{v}}$, $[\cdot]$, $\mathbf{[\cdot]^{v}}$, or $\mathbf{[\cdot]}$. An implicit requirement in the above definition is that the context C and expressions e_1 and

e_2 are such that the expressions can be legally plugged into the context's hole. In other words, we assume that $C[e_1]$ and $C[e_2]$ are syntactically valid expressions such that $\Psi'; \cdot; \cdot \vdash C[e_i] : \tau'$.

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* $\overline{\ell : \tau'}; \overline{\alpha}; \overline{x : \tau''} \vdash e : \tau \rightsquigarrow \mathbf{e}$, *then*

$$\overline{\ell : \tau'}; \overline{\alpha}; \overline{x : \tau''} \vdash e \approx^{ctx}_{M+C} {}^{\tau}\mathcal{MC}\,(\mathbf{e}[\overline{C\mathcal{M}^{\mathrm{ref}\,\tau'}\,\ell/\ell}][\overline{\lceil\alpha\rceil/\alpha}][\overline{C\mathbf{M}^{\tau''}(x)/x}]) : \tau.$$

The formal theorem essentially says that if e compiles to $\mathbf{e}$ then $e \approx^{ctx}_{M+C} \mathbf{e}$, 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 α, and free term variables $x : \tau''$, which the compiler translates into $\boldsymbol{\ell} : \tau'^{C}$, $\boldsymbol{\alpha}$, and $\mathbf{x} : \tau''^{C}$, respectively, in the output term $\mathbf{e}$. But the contextual equivalence in the above theorem uses the blue M environments, so the compiled component $\mathbf{e}$ 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 $C\mathcal{M}^{\mathrm{ref}\,\tau}\,\ell$ for translated locations, suspended type variables $\lceil\alpha\rceil$ for translated type variables, and the wrapped term $\mathbf{CM}^{\tau}(x)$ for translated term variables. Note that since we are in a call-by-value language, we are 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:

$$\overline{\boldsymbol{\ell} : \tau'^{C}}; \overline{\boldsymbol{\alpha}}; \overline{\mathbf{x} : \tau''^{C}} \vdash \mathbf{e}[\overline{{}^{\mathrm{ref}\,\widetilde{\tau'}}\mathcal{MC}\,\boldsymbol{\ell}/\ell}][\overline{\mathsf{L}\langle\boldsymbol{\alpha}\rangle/\alpha}][\overline{{}^{\widetilde{\tau''}}\mathbf{MC}(\mathbf{x})/\mathbf{x}}] \approx^{ctx}_{M+C} {}^{\widetilde{\tau}}\mathcal{MC}\,\mathbf{e} : \widetilde{\tau}$$

where $\widetilde{\tau} = \tau[\overline{\mathsf{L}\langle\boldsymbol{\alpha}\rangle/\alpha}]$, and similarly, $\widetilde{\tau'} = \tau'[\overline{\mathsf{L}\langle\boldsymbol{\alpha}\rangle/\alpha}]$ and $\widetilde{\tau''} = \tau''[\overline{\mathsf{L}\langle\boldsymbol{\alpha}\rangle/\alpha}]$.

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:

$$\overline{\ell : \tau'}; \overline{\alpha}; \overline{x : \tau''} \vdash C\mathcal{M}^{\tau}\,e \approx^{ctx}_{M+C} \mathbf{e}[\overline{C\mathcal{M}^{\mathrm{ref}\,\tau'}\,\ell/\ell}][\overline{\lceil\alpha\rceil/\alpha}][\overline{\mathbf{CM}^{\tau''}(x)/x}] : \tau^{\langle C\rangle}.$$

Note that this version of the theorem uses the boundary type translation $\tau^{\langle C\rangle}$ for the result type (instead of the compiler's type translation τ^{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.[2] 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 inexpressible 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 $\mathbf{C}$ that can distinguish between the compiled versions of e_1 and e_2. This means that the context $\mathbf{C}$ 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 τ. (This is the "awkward" example taken from Ahmed et al. [2009]; Dreyer et al. [2012].)

$$\begin{aligned}\tau &= ((\mathsf{unit}) \to \mathsf{unit}) \to \mathsf{int}\\ e_1 &= \mathsf{let}\ x = \mathsf{new}\ 0\ \mathsf{in}\ \lambda(f : \_).x := 0; f\,(); x := 1; f\,(); !\,x\\ e_2 &= \lambda(f : \_).f\,(); f\,(); 1\end{aligned}$$

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 $\mathbf{e_1}$ and $\mathbf{e_2}$ below, of translation type τ^{C} (i.e., $((\mathsf{unit}) \to \mathsf{unit}) \to \mathsf{int}^{C}$), are the compiled versions of e_1 and e_2:

$$\begin{aligned}&\tau^{C} = \exists\beta.\langle(((\beta, \exists\alpha.\langle(((\alpha, \mathsf{unit}) \to \mathsf{unit}), \alpha\rangle) \to \mathsf{int}), \beta\rangle\\ &\mathbf{e_1} = \mathbf{let\ x = new\ 0\ in}\\ &\quad \mathbf{pack}\,\langle\langle \mathbf{ref\ int}\rangle, \langle\lambda(\mathbf{env} : \langle \mathbf{ref\ int}\rangle, \mathbf{f} : (\mathsf{unit}) \to \mathsf{unit}^{C}).\mathbf{e_1'}, \langle \mathbf{x}\rangle\rangle\rangle\\ &\text{where } \mathbf{e_1'} = \mathbf{let\ x' = \pi_1(env)\ in\ x' := 0; unpack\ \langle\beta, fp\rangle = f\ in}\\ &\quad \mathbf{let\ (f', f_{env}) = (\pi_1(fp), \pi_2(fp))\ in\ f'\ (f_{env}, ()); x' := 1; f'\ (f_{env}, ()); !\,x'}\\ &\mathbf{e_2} = \mathbf{pack}\,\langle\langle\rangle, \langle\lambda(\mathbf{env} : \langle\rangle, \mathbf{f} : (\mathsf{unit}) \to \mathsf{unit}^{C}).\mathbf{e_2'}, \langle\rangle\rangle\rangle\\ &\text{where } \mathbf{e_2'} = \mathbf{unpack}\,\langle\beta, fp\rangle = f\ \mathbf{in}\\ &\quad \mathbf{let\ (f', f_{env}) = (\pi_1(fp), \pi_2(fp))\ in\ f'\ (f_{env}, ()); f'\ (f_{env}, ()); 1}\end{aligned}$$

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

$$\begin{aligned}\mathsf{C} \;=\;& \mathsf{let\ g} = [\cdot]\ \mathsf{in\ let\ b} = \mathsf{new\ 1\ in}\\ &\mathsf{let\ f} = \lambda(\_).\mathsf{let\ bv} = !\,\mathsf{b\ in}\\ &\qquad \mathsf{if0\ bv\ (call/cc}_{\mathsf{unit}}(\mathsf{k.\ g}\,(\lambda(\_).\mathsf{throw}_{\mathsf{unit}}\,()\,\mathsf{to\ k})))\\ &\qquad\quad (\mathsf{b} := 0)\ \mathsf{in}\\ &\mathsf{g\ f}\end{aligned}$$

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 $\mathbf{C}$ below, which has a hole of type $\tau^{\langle C\rangle}$.

[2] Technically speaking, our compiler correctness statement allows us to model linking as substitution for free variables in a component. For instance, if $\mathbf{e}$ is compiled code with a free variable $\mathbf{x} : \tau$, then we can link it with some $\mathbf{e'} : \tau$ by substituting for $\mathbf{x}$, as in $\mathbf{e}[\mathbf{e'}/\mathbf{x}]$). However, in the interest of readability, here we will show linking as context plugging: for instance, we can link appropriately typed $\mathbf{C}$ and $\mathbf{e}$ to get $\mathbf{C}[\mathbf{e}]$. Hence, we are loosely treating a context $\mathbf{C}$ as a component—which is justified since it can easily be turned into a component $\boldsymbol{\lambda}(\mathbf{x} : \_).\mathbf{C}[\mathbf{x}]$—which lets us eliminate extra clutter in our examples.

$$
\begin{array}{l}
\mathbf{C} = \mathbf{let\ g\ =\ [\cdot]\ in} \\
\quad \mathbf{unpack\ \langle \beta, gp\rangle = g\ in} \\
\quad \mathbf{let\ (g', g_{env}) = (\pi_1(gp), \pi_2(gp))\ in} \\
\quad \mathbf{let\ b\ =\ new\ 1\ in} \\
\quad \mathbf{let\ f\ =\ pack\ \langle \langle ref\ int\rangle,\ \langle (\lambda(f_{env} : \langle ref\ int\rangle, \_ : unit).} \\
\qquad\qquad\qquad\qquad \mathbf{let\ b'\ =\ \pi_1(f_{env})\ in} \\
\qquad\qquad\qquad\qquad \mathbf{let\ bv\ =\ !\,b'\ in} \\
\qquad\qquad\qquad\qquad \mathbf{if0\ bv\ (call/cc_{unit}(k.\ g'\ (g_{env}, p)))} \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathbf{(b' := 0)),} \\
\qquad\qquad\qquad\quad \mathbf{\langle b\rangle\rangle\ as\ ((unit) \to unit)^C\ in} \\
\quad \mathbf{g'(g_{env}, f)} \\
\text{where } \mathbf{p\ =\ pack\ \langle \langle cont\ unit\rangle,} \\
\quad \mathbf{\langle (\lambda(p_{env} : \langle cont\ unit\rangle,\ \_ : unit).\ throw_{unit}\ ()\ to\ \pi_1(p_{env})),\ \langle k\rangle\rangle}
\end{array}
$$

Note that $\mathbf{C[e_1]} \longmapsto^* \mathbf{0}$ and $\mathbf{C[e_2]} \longmapsto^* \mathbf{1}$.

Reasoning about Source, Target, and Mixed-Language Programs. As in Perconti and Ahmed [2014], having formalized and verified compiler correctness using a multi-language semantics, we can leverage certain ways of reasoning about programs after compilation and linking.

Consider our earlier component e_1. By compiler correctness, we know $\cdot;\cdot;\cdot \vdash e_1 \approx^{ctx}_{M+C} {}^{\tau}\mathcal{MC}\ \mathbf{e_1} : \tau$. Using boundary cancellation, we equivalently have $\cdot;\cdot;\cdot \vdash \mathcal{CM}^{\tau}\ e_1 \approx^{ctx}_{M+C} \mathbf{e_1} : \tau^{\langle C\rangle}$ where $\tau^{\langle C\rangle} = \tau^C$. Now if we link with the target context $\mathbf{C}$ from above, we have:

$$\cdot;\cdot;\cdot \vdash \mathbf{C}[\mathcal{CM}^{\tau}\ e_1] \approx^{ctx}_{M+C} \mathbf{C[e_1]} : \mathbf{int},$$

The right-hand side of this equivalence is exactly the purely C program that we would ultimately run, while the left-hand side is an M+C program that models it.

If we instead want to link with a different C context $\widetilde{\mathbf{C}}$ (of the same type as $\mathbf{C}$) that was compiled from an M context $\widetilde{C}$, we have:

$$\cdot;\cdot;\cdot \vdash \widetilde{\mathbf{C}}[\mathcal{CM}^{\tau}\ e_1] \approx^{ctx}_{M+C} \widetilde{\mathbf{C}}[\mathbf{e_1}] : \mathbf{int}$$

but we can also simplify this statement since we have the source code $\widetilde{C}$ that we compiled to $\widetilde{\mathbf{C}}$. Since our compiler correctness theorem tells us (roughly) that $\mathcal{CM}\ \widetilde{C} \approx^{ctx}_{M+C} \widetilde{\mathbf{C}}$, we can infer that

$$\cdot;\cdot;\cdot \vdash (\mathcal{CM}\ \widetilde{C})[\mathcal{CM}^{\tau}\ e_1] \approx^{ctx}_{M+C} \widetilde{\mathbf{C}}[\mathbf{e_1}] : \mathbf{int}$$

We can push the context plugging under the boundary to get:

$$\cdot;\cdot;\cdot \vdash \mathcal{CM}^{\mathsf{int}}\ \widetilde{C}[{}^{\tau}\mathcal{MC}\ \mathcal{CM}^{\tau}\ e_1] \approx^{ctx}_{M+C} \widetilde{\mathbf{C}}[\mathbf{e_1}] : \mathbf{int}$$

Applying boundary cancellation, we have:

$$\cdot;\cdot;\cdot \vdash \mathcal{CM}^{\mathsf{int}}\ \widetilde{C}[e_1] \approx^{ctx}_{M+C} \widetilde{\mathbf{C}}[\mathbf{e_1}] : \tau^C$$

Thus, we are now essentially equating a purely M program with a purely C program, since the only multi-language element in this statement is the integer boundary at the outermost level which will merely convert a source n to a target $\mathbf{n}$. This illustrates that when we do have source-language equivalents for all our target-level components, our multi-language framework allows us to model target-level linking with source-level linking.

6.2 Linking with a Threads Library

We return to our motivating example from §1 of linking with a simple threads package. It should be clear that the simple threads package shown in Figure 1 can easily be implemented in our source language M extended with call/cc, essentially as a package of existential type that exports three functions: fork, yield, and exit. Such a threads package can be closure converted to our target language C, yielding $\mathbf{e_{threads}}$. (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.)

```
structure T = Thread()
fun some1 () =
   (< some1 work start >
    T.yield ();
    < some1 work continue >
    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 ();
```

```
< some1 work start >
< some2 work start >
< some1 work continue >
< some2 work continue >
< some1 work finish >
< some2 work finish >
```

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

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 $\mathbf{e_{threads}}$. 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 ($\approx^{ctx}_{M+C}$) 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 ${}^{\mathsf{ref}\ \tau}\mathcal{MC}\ \ell$ and $\mathcal{CM}^{\mathsf{ref}\ \tau}\ \ell$ 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 $\tau_1 \to \tau_2$ if, given related arguments at type τ_1, the functions yield related results at type τ_2. Two tuples of length n are related at type $\langle \tau_1, \ldots, \tau_n \rangle$ if their i-th components (for all $1 \le i \le n$) are related at type τ_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 W. For instance, two locations ℓ_1 and ℓ_2 should only be related at type **ref** τ if: they actually exist in some heaps that satisfy the current world W; if they contain heap values related at type τ; 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 each consume 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 $\mathcal{V}[\![\tau]\!]$ that relates closed values at type τ, a continuation relation $\mathcal{K}[\![\tau]\!]$ that relates closed continuations (evaluation contexts) that have a hole of type τ, and a term relation $\mathcal{E}[\![\tau]\!]$ that relates closed terms at type τ. 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 $\Psi; \Delta; \Gamma \vdash e_1 \approx^{log}_{\mathsf{M+C}} e_2 : \tau$).

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, Ψ_1 and Ψ_2 are the heap types that any heaps H_1 and H_2, respectively, must have if they are to satisfy W, and Θ 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; δ are all possible transitions; π is the subset of δ that are marked as "public" transitions[3]; 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 ($\lfloor \cdot \rfloor_k$) 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, $\mathcal{V}[\![\tau]\!]$, $\mathcal{K}[\![\tau]\!]$, and $\mathcal{E}[\![\tau]\!]$, to be parametrized by a mapping ρ that provides relational interpretations for the free type variables in τ. For now, assume that ρ maps type variables α to triples $\text{VR} ::= (\tau_1, \tau_2, R)$, where τ_1 and τ_2 are the types used to instantiate α 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 φ 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 ρ_1 for the substitution that instantiates each $\alpha \in \text{dom}(\rho)$ with the corresponding τ_1, and ρ_2 for the substitution that instantiates each α with the corresponding τ_2. 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, $\mathcal{V}[\![\tau]\!]$, 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 α if they are in the relational interpretation of α, namely $\rho(\alpha).R$. (For now, ignore the $[M, M]$ that follows R in the figure).

[3] Dreyer et al. [2012] use private transitions to reason about well-bracketed control flow in the *absence* of call/cc. Note the shaded 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.

$$
\begin{array}{lcl}
\text{HeapAtom}_n & \stackrel{\text{def}}{=} & \{(W, H_1, H_2) \mid W \in \text{World}_n\} \\
\text{HeapRel}_n & \stackrel{\text{def}}{=} & \{\varphi_H \subseteq \text{HeapAtom}_n \mid \forall (W, H_1, H_2) \in \varphi_H.\ \forall W' \sqsupseteq W.\ (W', H_1, H_2) \in \varphi_H\} \\
\text{TermAtom}_n[\tau_1, \tau_2] & \stackrel{\text{def}}{=} & \{(W, e_1, e_2) \mid W \in \text{World}_n \wedge W.\Psi_1; \cdot; \cdot \vdash e_1 : \tau_1 \wedge W.\Psi_2; \cdot; \cdot \vdash e_2 : \tau_2\} \\
\text{ValAtom}_n[\tau_1, \tau_2] & \stackrel{\text{def}}{=} & \{(W, v_1, v_2) \in \text{TermAtom}_n[\tau_1, \tau_2]\} \\
\text{Island}_n & \stackrel{\text{def}}{=} & \{\theta = (s, S, \delta, \pi, \text{HR}, \text{bij}) \mid s \in S \wedge S \in \text{Set} \wedge \delta \subseteq S \times S \wedge \pi \subseteq \delta \wedge \pi = \delta \wedge \\
 & & \quad \delta, \pi \text{ reflexive} \wedge \delta, \pi \text{ transitive} \wedge \text{HR} \in S \to \text{HeapRel}_n \wedge \text{bij} \in S \to \mathbb{P}(\text{Val} \times \text{Val})\} \\
\text{World}_n & \stackrel{\text{def}}{=} & \{W = (k, \Psi_1, \Psi_2, \Theta) \mid k < n \wedge \exists m.\ \Theta \in \text{Island}_k^m\} \\
\text{ValAtom}[\tau]\rho & \stackrel{\text{def}}{=} & \text{ValAtom}[\rho_1(\tau), \rho_2(\tau)]
\end{array}
$$

$$
\begin{array}{l}
\text{ContAtom}[\tau_1, \tau_2] \rightsquigarrow [\tau_1', \tau_2'] \stackrel{\text{def}}{=} \{(W, E_1, E_2) \mid W \in \text{World} \wedge \exists \Psi_1, \Psi_2.\ \vdash E_1 : (W.\Psi_1; \cdot; \cdot \vdash \tau_1) \rightsquigarrow (\Psi_1; \cdot; \cdot \vdash \tau_1') \wedge \vdash E_2 : (W.\Psi_2; \cdot; \cdot \vdash \tau_2) \rightsquigarrow (\Psi_2; \cdot; \cdot \vdash \tau_2')\} \\
\text{ContAtom}[\tau_1, \tau_2]\rho \rightsquigarrow [\tau_1', \tau_2']\rho' \stackrel{\text{def}}{=} \text{ContAtom}[\rho_1(\tau_1), \rho_2(\tau_2)] \rightsquigarrow [\rho_1'(\tau_1'), \rho_2'(\tau_2')]
\end{array}
$$

$$
\begin{array}{lcllcl}
\lfloor(\theta_1, \ldots, \theta_m)\rfloor_k & \stackrel{\text{def}}{=} & (\lfloor\theta_1\rfloor_k, \ldots, \lfloor\theta_m\rfloor_k) & \triangleright(k+1, \Psi_1, \Psi_2, \Theta) & \stackrel{\text{def}}{=} & (k, \Psi_1, \Psi_2, \lfloor\Theta\rfloor_k) \\
\lfloor(s, S, \delta, \pi, \text{HR}, \text{bij})\rfloor_k & \stackrel{\text{def}}{=} & (s, S, \delta, \pi, \lfloor\text{HR}\rfloor_k, \text{bij}) & \triangleright\varphi & \stackrel{\text{def}}{=} & \{(W, e_1, e_2) \mid W.k > 0 \implies (\triangleright W, e_1, e_2) \in \varphi\} \\
\lfloor\text{HR}\rfloor_k & \stackrel{\text{def}}{=} & \lambda s.\ \lfloor\text{HR}(s)\rfloor_k & \varphi_H \otimes \varphi_H' & \stackrel{\text{def}}{=} & \{(W, H_1 \uplus H_1', H_2 \uplus H_2') \mid (W, H_1, H_2) \in \varphi_H \wedge (W, H_1', H_2') \in \varphi_H'\} \\
\lfloor\varphi_H\rfloor_k & \stackrel{\text{def}}{=} & \{(W, H_1, H_2) \in \varphi_H \mid W.k < k\} & & &
\end{array}
$$

$$
\begin{array}{lcl}
(k', \Psi_1', \Psi_2', \Theta') \sqsupseteq (k, \Psi_1, \Psi_2, \Theta) & \stackrel{\text{def}}{=} & k' \le k \wedge \Psi_1' \sqsupseteq \Psi_1 \wedge \Psi_2' \sqsupseteq \Psi_2 \wedge \Theta' \sqsupseteq \lfloor\Theta\rfloor_{k'} \\
(\theta_1', \ldots, \theta_{m'}') \sqsupseteq (\theta_1, \ldots, \theta_m) & \stackrel{\text{def}}{=} & m' \ge m \wedge \forall j \in \{1, \ldots, m\}.\ \theta_j' \sqsupseteq \theta_j \\
(s', S', \delta', \pi', \text{HR}', \text{bij}') \sqsupseteq (s, S, \delta, \pi, \text{HR}, \text{bij}) & \stackrel{\text{def}}{=} & (S', \delta', \pi', \text{HR}', \text{bij}') = (S, \delta, \pi, \text{HR}, \text{bij}) \wedge (s, s') \in \delta
\end{array}
$$

$$
\begin{array}{lcl}
\mathcal{V}[\![\tau, \tau^{\langle C\rangle}]\!]\rho & = & \{(W, v_1, \mathbf{v_2}) \in \text{ValAtom}[\rho_1(\tau), \rho_2(\tau^{\langle C\rangle})] \mid (W, v_1, {}^{\rho_2(\tau)}\mathbf{MC}(\mathbf{v_2})) \in \mathcal{V}[\![\tau]\!]\rho \wedge (W, \mathbf{CM}^{\rho_1(\tau)}(v_1), \mathbf{v_2}) \in \mathcal{V}[\![\tau^{\langle C\rangle}]\!]\rho\} \\
\mathcal{V}[\![\tau^{\langle C\rangle}, \tau]\!]\rho & = & \{(W, \mathbf{v_1}, v_2) \in \text{ValAtom}[\rho_1(\tau^{\langle C\rangle}), \rho_2(\tau)] \mid (W, {}^{\rho_1(\tau)}\mathbf{MC}(\mathbf{v_1}), v_2) \in \mathcal{V}[\![\tau]\!]\rho \wedge (W, \mathbf{v_1}, \mathbf{CM}^{\rho_2(\tau)}(v_2)) \in \mathcal{V}[\![\tau^{\langle C\rangle}]\!]\rho\} \\
\mathcal{V}[\![\tau, \tau]\!]\rho & = & \mathcal{V}[\![\tau]\!]\rho
\end{array}
$$

$$
\begin{array}{l}
\mathcal{V}[\![\alpha]\!]\rho = \rho(\alpha).R[M, M] \qquad \mathcal{V}[\![\text{unit}]\!]\rho = \{(W, (), ()) \in \text{ValAtom}[\text{unit}]\rho\} \qquad \mathcal{V}[\![\text{int}]\!]\rho = \{(W, \mathsf{n}, \mathsf{n}) \in \text{ValAtom}[\text{int}]\rho\} \\
\mathcal{V}[\![\forall[\overline{\alpha}].(\overline{\tau}) \to \tau']\!]\rho = \{(W, v_1, v_2) \in \text{ValAtom}[\forall[\overline{\alpha}].(\overline{\tau}) \to \tau']\rho \mid \forall W' \sqsupseteq W.\ \forall \overline{\text{VR} \in \text{MMValRel}}.\ \forall \overline{v_1', v_2'}.\ (W', v_1', v_2') \in \mathcal{V}[\![\tau]\!]\rho\overline{[\alpha \mapsto \text{VR}]} \implies \\
\qquad\qquad (W', v_1\,\overline{[\text{VR}.\tau_1]\,v_1'}, v_2\,\overline{[\text{VR}.\tau_2]\,v_2'}) \in \mathcal{E}[\![\tau', \tau']\!]\rho[\alpha \mapsto \overline{\text{VR}}]\} \\
\mathcal{V}[\![\exists\alpha.\tau]\!]\rho = \{(W, \text{pack}\,\langle\tau_1, v_1\rangle\,\text{as}\,\rho_1(\exists\alpha.\tau), \text{pack}\,\langle\tau_2, v_2\rangle\,\text{as}\,\rho_2(\exists\alpha.\tau)) \in \text{ValAtom}[\exists\alpha.\tau]\rho \mid \\
\qquad\qquad \exists \text{VR} \in \text{MMValRel} \wedge \text{VR}.\tau_1 = \tau_1 \wedge \text{VR}.\tau_2 = \tau_2 \wedge (W, v_1, v_2) \in \mathcal{V}[\![\tau]\!]\rho[\alpha \mapsto \text{VR}]\} \\
\mathcal{V}[\![\mu\alpha.\tau]\!]\rho = \{(W, \text{fold}_{\rho_1(\mu\alpha.\tau)}\,v_1, \text{fold}_{\rho_2(\mu\alpha.\tau)}\,v_2) \in \text{ValAtom}[\mu\alpha.\tau]\rho \mid (W, v_1, v_2) \in \triangleright\mathcal{V}[\![\tau[\mu\alpha.\tau/\alpha]]\!]\rho\} \\
\mathcal{V}[\![\text{ref}\,\tau]\!]\rho = \{(W, v_1, v_2) \in \text{ValAtom}[\text{ref}\,\tau]\rho \mid \exists i.\ \forall W' \sqsupseteq W.\ (\text{loc}(v_1), \text{loc}(v_2)) \in W'(i).\text{bij}(W'(i).s) \wedge \\
\qquad\qquad \exists\varphi_H.\ W'(i).\text{HR}(W'(i).s) = \varphi_H \otimes \{(\widetilde{W}, H_1, H_2) \in \text{HeapAtom} \mid \forall\tau_1, \tau_2, v_1', v_2'. \\
\qquad\qquad\qquad\qquad \text{lookup}^{\tau}(v_1, H_1) \hookrightarrow (v_1', \tau_1) \wedge \text{lookup}^{\tau}(v_2, H_2) \hookrightarrow (v_2', \tau_2) \implies (\widetilde{W}, v_1', v_2') \in \mathcal{V}[\![\tau_1, \tau_2]\!]\rho\}\} \\
\mathcal{V}[\![\langle\tau_1, \ldots, \tau_n\rangle]\!]\rho = \{(W, \langle v_{11}, \ldots, v_{1n}\rangle, \langle v_{21}, \ldots, v_{2n}\rangle) \in \text{ValAtom}[\langle\tau_1, \ldots, \tau_n\rangle]\rho \mid \forall j \in \{1, \ldots, n\}.\ (W, v_{1j}, v_{2j}) \in \mathcal{V}[\![\tau_j]\!]\rho\} \\
\mathcal{V}[\![\mathsf{L}\langle\tau\rangle]\!]\rho \stackrel{\text{def}}{=} \{(W, {}^{\rho_1(\mathsf{L}\langle\tau\rangle)}\mathcal{MC}\,\mathbf{v_1}, {}^{\rho_2(\mathsf{L}\langle\tau\rangle)}\mathcal{MC}\,\mathbf{v_2}) \in \text{ValAtom}[\mathsf{L}\langle\tau\rangle]\rho \mid (W, \mathbf{v_1}, \mathbf{v_2}) \in \mathcal{V}[\![\tau]\!]\rho\}
\end{array}
$$

$$
\begin{array}{ll}
\text{lookup}^{\tau}(\ell, \{\ell \mapsto v\}) \hookrightarrow (v, \tau) & \text{lookup}^{\tau}(\mathcal{CM}^{\text{ref}\,\tau'}\,\ell, \{\ell \mapsto v\}) \hookrightarrow (v, \hat{\tau}) \quad \text{where } \exists\hat{\tau}.\ \hat{\tau}^{\langle C\rangle} = \tau \\
\text{lookup}^{\tau}({}^{\text{ref}\,\tau'}\mathcal{MC}\,\ell, \{\ell \mapsto v\}) \hookrightarrow (v, \tau^{\langle C\rangle}) & \text{lookup}^{\tau}(\ell, \{\ell \mapsto v\}) \hookrightarrow (v, \tau)
\end{array}
$$

$$
\begin{array}{l}
\mathcal{V}[\![\boldsymbol{\alpha}]\!]\rho = \rho(\boldsymbol{\alpha}).R[C, C] \qquad \mathcal{V}[\![\mathbf{unit}]\!]\rho = \{(W, (), ()) \in \text{ValAtom}[\mathbf{unit}]\rho\} \qquad \mathcal{V}[\![\mathbf{int}]\!]\rho = \{(W, \mathbf{n}, \mathbf{n}) \in \text{ValAtom}[\mathbf{int}]\rho\} \\
\mathcal{V}[\![\mathbf{\forall[\overline{\alpha}].(\overline{\tau}) \to \tau'}]\!]\rho = \{(W, \mathbf{v_1}, \mathbf{v_2}) \in \text{ValAtom}[\mathbf{\forall[\overline{\alpha}].(\overline{\tau}) \to \tau'}]\rho \mid \forall W' \sqsupseteq W.\ \forall \overline{\text{VR} \in \text{CCValRel}}.\ \forall \overline{\mathbf{v_1', v_2'}}.\ (W', \mathbf{v_1', v_2'}) \in \mathcal{V}[\![\tau]\!]\rho\overline{[\alpha \mapsto \text{VR}]} \implies \\
\qquad\qquad (W', \mathbf{v_1}\,\overline{[\text{VR}.\tau_1]\,v_1'}, \mathbf{v_2}\,\overline{[\text{VR}.\tau_2]\,v_2'}) \in \mathcal{E}[\![\tau', \tau']\!]\rho[\alpha \mapsto \overline{\text{VR}}]\} \\
\mathcal{V}[\![\mathbf{\exists\alpha.\tau}]\!]\rho = \{(W, \mathbf{pack}\,\langle\tau_1, \mathbf{v_1}\rangle\,\mathbf{as}\,\rho_1(\exists\alpha.\tau), \mathbf{pack}\,\langle\tau_2, \mathbf{v_2}\rangle\,\mathbf{as}\,\rho_2(\exists\alpha.\tau)) \in \text{ValAtom}[\mathbf{\exists\alpha.\tau}]\rho \mid \\
\qquad\qquad \exists \text{VR} \in \text{CCValRel} \wedge \text{VR}.\tau_1 = \tau_1 \wedge \text{VR}.\tau_2 = \tau_2 \wedge (W, \mathbf{v_1}, \mathbf{v_2}) \in \mathcal{V}[\![\tau]\!]\rho[\alpha \mapsto \text{VR}]\} \\
\mathcal{V}[\![\mathbf{ref}\,\tau]\!]\rho = \{(W, \mathbf{v_1}, \mathbf{v_2}) \in \text{ValAtom}[\mathbf{ref}\,\tau]\rho \mid \exists i.\ \forall W' \sqsupseteq W.\ (\text{loc}(\mathbf{v_1}), \text{loc}(\mathbf{v_2})) \in W'(i).\text{bij}(W'(i).s) \wedge \\
\qquad\qquad \exists\varphi_H.\ W'(i).\text{HR}(W'(i).s) = \varphi_H \otimes \{(\widetilde{W}, H_1, H_2) \in \text{HeapAtom} \mid \forall\tau_1, \tau_2, v_1', v_2'. \\
\qquad\qquad\qquad\qquad \text{lookup}^{\tau}(\mathbf{v_1}, H_1) \hookrightarrow (v_1', \tau_1) \wedge \text{lookup}^{\tau}(\mathbf{v_2}, H_2) \hookrightarrow (v_2', \tau_2) \implies (\widetilde{W}, v_1', v_2') \in \mathcal{V}[\![\tau_1, \tau_2]\!]\rho\}\} \\
\mathcal{V}[\![\lceil\alpha\rceil]\!]\rho = \rho(\alpha).R[C, C] \\
\mathcal{V}[\![\mathbf{cont}\,\tau]\!]\rho = \{(W, \mathbf{cont}_{\rho_1(\tau)}\,E_1, \mathbf{cont}_{\rho_2(\tau)}\,E_2) \in \text{ValAtom}[\mathbf{cont}\,\tau]\rho \mid \forall W', \mathbf{v_1}, \mathbf{v_2}.\ W' \sqsupseteq W \wedge (W', \mathbf{v_1}, \mathbf{v_2}) \in \mathcal{V}[\![\tau]\!]\rho \implies (W', E_1[\mathbf{v_1}], E_2[\mathbf{v_2}]) \in O\}
\end{array}
$$

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-level abstraction, so we say that two functions are related at type $\forall[\overline{\alpha}].(\overline{\tau}) \to \tau'$ if the following holds: given any admissible relation triple $\text{VR} = (\tau_1, \tau_2, R)$ for each α that the function abstracts over, applying the functions in some future world W' to (1) the types τ_1 and τ_2, respectively, and (2) argument values related in W' at types τ under a mapping ρ extended to account for the new VR for each α, they yield related results. Here, the definition in the figure says the functions yields results related in the relation $\mathcal{E}[\![\tau', \tau']\!]$. We will explain this "two-dimensional" $\mathcal{E}$ relation later (when we discuss mutable references). For now, it suffices to think of $\mathcal{E}[\![\tau', \tau']\!]$ simply as $\mathcal{E}[\![\tau']\!]$ which specifies when computations are related at the type τ'.

Packages are related at existential type $\exists\alpha.\tau$ if there exists some interpretation VR of the abstract type under which their bodies are related. Values of recursive type $\mu\alpha.\tau$ are related if unfolding the folded value yields values that are related at the type $\tau[\mu\alpha.\tau/\alpha]$ 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, $\mathcal{V}[\![\boldsymbol{\tau}]\!]$, 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 $\lceil\alpha\rceil$ which we discuss below.

Figure 11 presents the term and continuation relations. In the term relation $\mathcal{E}[\![\tau]\!]\rho$, two terms are related if running them in related continuations gives related observations. Two continuations are related in $\mathcal{K}[\![\tau]\!]\rho$ 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 $\top\top$-closure, and it yields a logical relation that is complete with respect to contextual equivalence.

Under the relation 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 ℓ_1 and ℓ_2 are related at type $\text{ref}\,\tau$ in world W if there exists an island in W (the ith 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 τ , and (2) the pair of locations (ℓ_1, ℓ_2) are included in the bijection on locations (bij) tracked by that island.[4]

[4]The intuition for requirement (2) (i.e., location bijection) is that when relating two programs with heaps, we must keep track of which pairs of locations from the two programs should be considered related, and this bijection must be preserved as the programs run. But note that only "global" locations—informally, those accessible to a context—need to be part of the bijection; locations that are considered "local state" need not correspond to any location in the other program.

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 ${}^{\text{ref}\,\tau}\mathcal{MC}\,\boldsymbol{\ell_2}$ 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 ℓ_1 (a location) be related to a value ${}^{\text{ref}\,\tau}\mathcal{MC}\,\boldsymbol{\ell_2}$ (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 $H_1(\ell_1) = \mathsf{v}_1$ and let $H_2(\boldsymbol{\ell_2}) = \mathbf{v_2}$. Then, we could state our requirement that the locations' contents be related by requiring that the values v_1 and ${}^{\tau}\mathbf{MC}(\mathbf{v_2})$ be related at type τ.

Unfortunately, the above strategy of defining $\mathcal{V}[\![\text{ref}\,\tau]\!]$ (and similarly $\mathcal{V}[\![\mathbf{ref}\,\boldsymbol{\tau}]\!]$) 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 τ and the other has type $\tau^{\langle C\rangle}$). Thus, we have $\mathcal{V}[\![\tau, \tau^{\langle C\rangle}]\!]\rho$ and $\mathcal{V}[\![\tau^{\langle C\rangle}, \tau]\!]\rho$, 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 $\mathcal{V}[\![\tau, \tau]\!]\rho$ and $\mathcal{V}[\![\boldsymbol{\tau}, \boldsymbol{\tau}]\!]\rho$. Off-diagonal, we have the cross-language versions $\mathcal{V}[\![\tau, \tau^{\langle C\rangle}]\!]\rho$ and $\mathcal{V}[\![\tau^{\langle C\rangle}, \tau]\!]\rho$. 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 VR of a type variable α must satisfy to be considered admissible. As usual, these requirements stem from lemmas that we need about $\mathcal{V}[\![\tau]\!]\rho$: Since $\tau = \alpha$ is a base case, these properties need to hold for any interpretation of α. 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* $(W, \mathsf{v}_1, \mathsf{v}_2) \in \mathcal{V}[\![\tau]\!]\rho$, *and* ${}^{\rho_2'(\tau)}\boldsymbol{MC}(\boldsymbol{CM}^{\rho_2(\tau)}(\mathsf{v}_2)) = \mathsf{v}_2'$, *then* $(W, \mathsf{v}_1, \mathsf{v}_2') \in \mathcal{V}[\![\tau]\!]\rho'$.

$$(H_1, H_2) : W \stackrel{\text{def}}{=} \vdash H_1 : W.\Phi_1 \wedge \vdash H_2 : W.\Phi_2 \wedge (W.k > 0 \implies (\triangleright W, H_1, H_2) \in \bigotimes \{\, \theta.\mathrm{HR}(\theta.s) \mid \theta \in W.\Theta \,\})$$

$$\mathrm{running}(k, \langle H \mid e \rangle) \stackrel{\text{def}}{=} \exists H', e'.\ \langle H \mid e \rangle \longmapsto^k \langle H' \mid e' \rangle$$

$$\mathcal{O} = \{\, (W, e_1, e_2) \mid \forall (H_1, H_2) : W.\ (\langle H_1 \mid e_1 \rangle \Downarrow \wedge \langle H_2 \mid e_2 \rangle \Downarrow) \quad \vee \quad (\mathrm{running}(W.k, \langle H_1 \mid e_1 \rangle) \wedge \mathrm{running}(W.k, \langle H_2 \mid e_2 \rangle)) \,\}$$

$$\mathcal{K}[\![\tau_1, \tau_2]\!]\rho = \{\, (W, E_1, E_2) \in \mathrm{ContAtom}[\tau_1, \tau_2]\rho \rightsquigarrow [\tau_1', \tau_2']\rho' \mid \forall W', v_1, v_2.\ W' \sqsupseteq_{\mathrm{pub}} W \wedge (W', v_1, v_2) \in \mathcal{V}[\![\tau_1, \tau_2]\!]\rho \implies (W', E_1[v_1], E_2[v_2]) \in \mathcal{O} \,\}$$

$$\mathcal{E}[\![\tau_1, \tau_2]\!]\rho = \{\, (W, e_1, e_2) \in \mathrm{TermAtom}[\rho_1(\tau_1), \rho_2(\tau_2)] \mid \forall E_1, E_2.\ (W, E_1, E_2) \in \mathcal{K}[\![\tau_1, \tau_2]\!]\rho \implies (W, E_1[e_1], E_2[e_2]) \in \mathcal{O} \,\}$$

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

$$\mathrm{ValRel}[\tau_1, \tau_2] \stackrel{\text{def}}{=} \{\, \varphi \subseteq \mathrm{ValAtom}[\tau_1, \tau_2] \mid \forall (W, v_1, v_2) \in \varphi.\ \forall W' \sqsupseteq W.\ (W', v_1, v_2) \in \varphi \,\}$$

$$\mathcal{CM}_1(\tau_1, \varphi) \stackrel{\text{def}}{=} \{\, (W, \mathbf{v_1}, v_2) \mid (W, v_1, v_2) \in \varphi \wedge \mathbf{CM}^{\tau_1}(v_1) = \mathbf{v_1} \,\}$$

$$\mathcal{MC}_1(\tau_1, \varphi) \stackrel{\text{def}}{=} \{\, (W, v_1, v_2) \mid (W, \mathbf{v_1}, v_2) \in \varphi \wedge {}^{\tau_1}\mathbf{MC}(\mathbf{v_1}) = v_1 \,\}$$

$$\mathcal{CM}_2(\tau_2, \varphi) \stackrel{\text{def}}{=} \{\, (W, v_1, \mathbf{v_2}) \mid (W, v_1, v_2) \in \varphi \wedge \mathbf{CM}^{\tau_2}(v_2) = \mathbf{v_2} \,\}$$

$$\mathcal{MC}_2(\tau_2, \varphi) \stackrel{\text{def}}{=} \{\, (W, v_1, v_2) \mid (W, v_1, \mathbf{v_2}) \in \varphi \wedge {}^{\tau_2}\mathbf{MC}(\mathbf{v_2}) = v_2 \,\}$$

$$\begin{aligned}\mathrm{MMValRel} \stackrel{\text{def}}{=} \{\, & \mathrm{VR} = (\tau_1, \tau_2, R) \mid \\ & R[M, M] \in \mathrm{ValRel}[\tau_1, \tau_2] \wedge R[C, M] \in \mathrm{ValRel}[{\tau_1}^{\langle C \rangle}, \tau_2] \wedge \\ & R[M, C] \in \mathrm{ValRel}[\tau_1, {\tau_2}^{\langle C \rangle}] \wedge R[C, C] \in \mathrm{ValRel}[{\tau_1}^{\langle C \rangle}, {\tau_2}^{\langle C \rangle}] \wedge \\ & \mathcal{CM}_1(\tau_1, R[M, M]) \subseteq R[C, M] \wedge \mathcal{MC}_1(\tau_1, R[C, M]) \subseteq R[M, M] \wedge \\ & \mathcal{CM}_1(\tau_1, R[M, C]) \subseteq R[C, C] \wedge \mathcal{MC}_1(\tau_1, R[C, C]) \subseteq R[M, C] \wedge \\ & \mathcal{CM}_2(\tau_2, R[M, M]) \subseteq R[M, C] \wedge \mathcal{MC}_2(\tau_2, R[M, C]) \subseteq R[M, M] \wedge \\ & \mathcal{CM}_2(\tau_2, R[C, M]) \subseteq R[C, C] \wedge \mathcal{MC}_2(\tau_2, R[C, C]) \subseteq R[C, M] \,\}\end{aligned}$$

$$\mathrm{CCValRel} \stackrel{\text{def}}{=} \{\, \mathrm{VR} = (\boldsymbol{\tau_1}, \boldsymbol{\tau_2}, R) \mid R[C, C] \in \mathrm{ValRel}[\boldsymbol{\tau_1}, \boldsymbol{\tau_2}] \,\}$$

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 $\Delta \vdash \tau$ and $\rho \in \mathcal{D}[\![\Delta]\!]$.*
1. If $(W, \mathbf{e_1}, \mathbf{e_2}) \in \mathcal{E}[\![\tau^{\langle C \rangle}]\!]\rho$ then $(W, {}^{\rho_1(\tau)}\mathcal{MC}\ \mathbf{e_1}, {}^{\rho_2(\tau)}\mathcal{MC}\ \mathbf{e_2}) \in \mathcal{E}[\![\tau]\!]\rho$
2. If $(W, \mathbf{e_1}, \mathbf{e_2}) \in \mathcal{E}[\![\tau]\!]\rho$ then $(W, \mathcal{CM}^{\rho_1(\tau)}\ \mathbf{e_1}, \mathcal{CM}^{\rho_2(\tau)}\ \mathbf{e_2}) \in \mathcal{E}[\![\tau^{\langle C \rangle}]\!]\rho$

A further complication is that the admissibility criteria for value relations and the choice of how to define $\mathcal{V}[\![\lceil\alpha\rceil]\!]\rho$ are intertwined. Intuitively, the interpretation of $\lceil\alpha\rceil$ requires that we be able to take the relation VR for α and be able to "translate" the relation into some VR′ for $\lceil\alpha\rceil$, 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 $\mathcal{V}[\![\lceil\alpha\rceil]\!]\rho$, 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 φ on values in the language of the type variable whose interpretation is being given, but rather an $i+1$-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 left (or the right) values in one relation should preserve relatedness in the appropriate neighboring relation.

$$\mathcal{D}[\![\cdot]\!] = \{\, \emptyset \,\}$$

$$\mathcal{D}[\![\Delta, \alpha]\!] = \{\, \rho[\alpha \mapsto \mathrm{VR}] \mid \rho \in \mathcal{D}[\![\Delta]\!] \wedge \mathrm{VR} \in \mathrm{MMValRel} \,\}$$

$$\mathcal{D}[\![\Delta, \boldsymbol{\alpha}]\!] = \{\, \rho[\boldsymbol{\alpha} \mapsto \mathrm{VR}] \mid \rho \in \mathcal{D}[\![\Delta]\!] \wedge \mathrm{VR} \in \mathrm{CCValRel} \,\}$$

$$\mathcal{G}[\![\cdot]\!]\rho = \{\, (W, \emptyset) \mid W \in \mathrm{World} \,\}$$

$$\mathcal{G}[\![\Gamma, x : \tau]\!]\rho = \{\, (W, \gamma[x \mapsto (v_1, v_2)] \mid (W, \gamma) \in \mathcal{G}[\![\Gamma]\!]\rho \wedge (W, v_1, v_2) \in \mathcal{V}[\![\tau]\!]\rho \,\}$$

$$\mathcal{H}[\![\{\cdot\}]\!] = \mathrm{World}$$

$$\mathcal{H}[\![^-, \ell : \tau]\!] = \mathcal{H}[\![^-]\!] \cap \{\, W \in \mathrm{World} \mid (W, \ell, \ell) \in \mathcal{V}[\![\mathbf{ref}\ \tau]\!]\emptyset \,\}$$

$$\mathcal{H}[\![\{\cdot\}]\!] = \mathrm{World}$$

$$\mathcal{H}[\![^-, \boldsymbol{\ell} : \boldsymbol{\tau}]\!] = \mathcal{H}[\![^-]\!] \cap \{\, W \in \mathrm{World} \mid (W, \boldsymbol{\ell}, \boldsymbol{\ell}) \in \mathcal{V}[\![\mathbf{ref}\ \boldsymbol{\tau}]\!]\emptyset \,\}$$

$$\mathcal{H}[\![^-; ^-]\!] = \mathcal{H}[\![^-]\!] \cap \mathcal{H}[\![^-]\!]$$

$$\Psi; \Delta; \Gamma \vdash e_1 \approx^{log}_{\mathrm{M+C}} e_2 : \tau \stackrel{\text{def}}{=} \Psi; \Delta; \Gamma \vdash e_1 : \tau \wedge \Psi; \Delta; \Gamma \vdash e_2 : \tau \wedge \forall W, \rho, \gamma.\ W \in \mathcal{H}[\![\Psi]\!] \wedge \rho \in \mathcal{D}[\![\Delta]\!] \wedge (W, \gamma) \in \mathcal{G}[\![\Gamma]\!]\rho \implies (W, \rho_1(\gamma_1(e_1)), \rho_2(\gamma_2(e_2))) \in \mathcal{E}[\![\tau, \tau]\!]\rho$$

$$\Psi; \Delta; \Gamma \vdash E_1 \approx^{log}_{\mathrm{M+C}} E_2 \div \tau \stackrel{\text{def}}{=} \Psi; \Delta; \Gamma \vdash E_1 \div \tau \wedge \Psi; \Delta; \Gamma \vdash E_2 \div \tau \wedge \forall W, \rho, \gamma.\ W \in \mathcal{H}[\![\Psi]\!] \wedge \rho \in \mathcal{D}[\![\Delta]\!] \wedge (W, \gamma) \in \mathcal{G}[\![\Gamma]\!]\rho \implies (W, \rho_1(\gamma_1(E_1)), \rho_2(\gamma_2(E_2))) \in \mathcal{K}[\![\tau]\!]\rho$$

Figure 13: Logical Relation for Open Terms

Now, we can simply define $\mathcal{V}[\![\lceil\alpha\rceil]\!]\rho = \rho(\alpha).R[C, C]$ and define $\mathcal{V}[\![\alpha]\!]\rho = \rho(\alpha).R[M, M]$. With our admissibility criteria, we are able to prove the boundary cancellation and bridge lemmas (see the technical appendix for details).

Logical Relation: Open Terms and Continuations. Figure 13 shows how we lift the closed relations $\mathcal{E}$ and $\mathcal{K}$ to open terms and continuations. Two open terms e_1 and e_2 (or open continuations E_1 and E_2) are related if, given a world (which must satisfy the heap type Ψ), a mapping ρ (that maps type variables to admissible VR's), and a pair of substitutions γ (where the values being substituted must be related), we get related components (or related continuations) by closing off e_1 and e_2 (respectively, E_1 and E_2) with ρ and γ.

7.1 Properties of the M+C Logical Relation

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 $\rho \in \mathcal{D}[\![\Delta]\!]$, $\Delta \vdash \tau$, $\Delta \vdash \boldsymbol{\tau}$ and $W' \sqsupseteq W$, then*

(1) $(W, \mathbf{v_1}, \mathbf{v_2}) \in \mathcal{V}[\![\tau]\!]\rho \implies (W', \mathbf{v_1}, \mathbf{v_2}) \in \mathcal{V}[\![\tau]\!]\rho$
(2) $(W, v_1, v_2) \in \mathcal{V}[\![\tau]\!]\rho \implies (W', v_1, v_2) \in \mathcal{V}[\![\tau]\!]\rho$
(3) $(W, v_1, \mathbf{v_2}) \in \mathcal{V}[\![\tau, \tau^{\langle C \rangle}]\!]\rho \implies (W', v_1, \mathbf{v_2}) \in \mathcal{V}[\![\tau, \tau^{\langle C \rangle}]\!]\rho$
(4) $(W, \mathbf{v_1}, v_2) \in \mathcal{V}[\![\tau^{\langle C \rangle}, \tau]\!]\rho \implies (W', \mathbf{v_1}, v_2) \in \mathcal{V}[\![\tau^{\langle C \rangle}, \tau]\!]\rho$

The term relation $\mathcal{E}[\![\tau]\!]$ 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 $\mathcal{E}[\![\tau]\!]$, then the former must be as well.

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

$$\forall (H_1, H_2) : W.\ \exists (H_1', H_2') : W'.\ \langle H_1 \mid e_1 \rangle \longmapsto^{k_1} \langle H_1' \mid e_1' \rangle \ \wedge\ \langle H_2 \mid e_2 \rangle \longmapsto^{k_2} \langle H_2' \mid e_2' \rangle,$$

then $(W', e_1', e_2') \in \mathcal{E}[\![\tau]\!]\rho \implies (W, e_1, e_2) \in \mathcal{E}[\![\tau]\!]\rho$.

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

Lemma 7.5 (Monadic Bind). *If* $(W, e_1, e_2) \in \mathcal{E}[\![\tau]\!]\rho, (W, E_1, E_2) \in \text{ContAtom}[\tau, \tau]\rho \rightsquigarrow [\tau', \tau']\rho'$ *and*

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

then $(W, E_1[e_1], E_2[e_2]) \in \mathcal{E}[\![\tau']\!]\rho$.

We also prove the boundary cancellation and bridge lemmas stated earlier and then prove that the $\mathcal{V}$ relations are admissible.

Lemma 7.6 (Admissibility of $\mathcal{V}$). *Let* $\rho \in \mathcal{D}[\![\Delta]\!]$.

(1) *If* $\Delta \vdash \tau$, *then* $(\rho_1(\tau), \rho_2(\tau), R) \in \text{MMValRel}$.
where $R = \begin{bmatrix} \mathcal{V}[\![\tau, \tau]\!]\rho & \mathcal{V}[\![\tau, \tau^{\langle C \rangle}]\!]\rho \\ \mathcal{V}[\![\tau^{\langle C \rangle}, \tau]\!]\rho & \mathcal{V}[\![\tau^{\langle C \rangle}, \tau^{\langle C \rangle}]\!]\rho \end{bmatrix}$
(2) *If* $\Delta \vdash \boldsymbol{\tau}$, *then* $(\rho_1(\boldsymbol{\tau}), \rho_2(\boldsymbol{\tau}), [\mathcal{V}[\![\boldsymbol{\tau}]\!]\rho]) \in \text{CCValRel}$.

The proof of admissibility follows easily from monotonicity (Lemma 7.3), boundary cancellation for $\mathcal{V}$ (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* $\Psi; \Delta; \Gamma \vdash e \approx^{log}_{M+C} e : \tau$

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* $\Psi; \cdot; \cdot \vdash e_1 \approx^{log}_{M+C} e_2 : \tau, \vdash H : \Psi$, *then* $\langle H \mid e_1 \rangle \Downarrow$ *if and only if* $\langle H \mid e_2 \rangle \Downarrow$.

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

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* $\Psi; \Delta; \Gamma \vdash e_1 \approx^{ctx}_{M+C} e_2 : \tau$, *then* $\Psi; \Delta; \Gamma \vdash e_1 \approx^{log}_{M+C} e_2 : \tau$.

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* $[\cdot]$ *then:* $\Psi; \Delta; \Gamma \vdash e_1 \approx^{log}_{M+C} C[\mathbf{e_2}] : \tau$ *iff* $\Psi; \Delta; \Gamma \vdash e_1 \approx^{log}_{M+C} C[{}^{\tau'}\mathcal{MC}\, \mathcal{CM}^{\tau'}\, \mathbf{e_2}] : \tau$.
(2) *If the hole in* C *is* $\mathbf{[\cdot]}$ *and* $\Psi; \Delta; \Gamma \vdash \mathbf{e_2} : \tau'^{\langle C \rangle}$ *then:* $\Psi; \Delta; \Gamma \vdash e_1 \approx^{log}_{M+C} C[\mathbf{e_2}] : \tau$ *iff* $\Psi; \Delta; \Gamma \vdash e_1 \approx^{log}_{M+C} C[\mathcal{CM}^{\tau'\ \tau'}\mathcal{MC}\, \mathbf{e_2}] : \tau$.
(3) *If the hole in* C *is* $[\cdot]^{\mathbf{v}}$ *then:* $\Psi; \Delta; \Gamma \vdash e \approx^{log}_{M+C} C[\mathbf{v}] : \tau$ *iff* $\Psi; \Delta; \Gamma \vdash e \approx^{log}_{M+C} C[{}^{\tau'}\boldsymbol{MC}(\boldsymbol{CM}^{\tau'}(\mathbf{v}))] : \tau$.
(4) *If the hole in* C *is* $\mathbf{[\cdot]^v}$ *then:* $\Psi; \Delta; \Gamma \vdash e \approx^{log}_{M+C} C[\mathbf{v}] : \tau$ *iff* $\Psi; \Delta; \Gamma \vdash e \approx^{log}_{M+C} C[\boldsymbol{CM}^{\tau'}({}^{\tau'}\boldsymbol{MC}(\mathbf{v}))] : \tau$.

Next, we prove correctness of closure conversion.

Theorem 7.12 (Closure Conversion is Semantics Preserving). *If* $^{-}; ^{\prime}; ^{`} \vdash \mathbf{e} : \tau \rightsquigarrow \mathbf{e}$, *then*

$$^{-}; ^{\prime}; ^{`} \vdash \mathbf{e} \approx^{log}_{M+C} {}^{\tau}\mathcal{MC}\ (\overline{\mathbf{e}[\mathcal{CM}^{\mathbf{ref}\ \tau'}\ \ell/\ell]}[\overline{\lceil\alpha\rceil/\alpha}][\overline{\boldsymbol{CM}^{\tau''}(\mathbf{x})/\mathbf{x}}]) : \tau.$$

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 Perconti-Ahmed and our result—restrict linking to target code that can be expressed in the compiler's source language,

Benton and Hur [2009] and Hur 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 Hur prove correctness of a compiler from STLC with recursion to an SECD machine, while Hur 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 Pilsner, a multi-pass compiler from an ML-like language to assembly [Neis et al. 2015; Neis 2018]. However, all of these cross-language-/inter-language-relation 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 specification.

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 compiler 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 $\mathcal{W}^+$ and $\mathcal{W}^-$, that translate components of type τ to τ^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 $\mathcal{V}$ relation in terms of the diagonals so this approach scales to more languages without making the $\mathcal{V}$ 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 to 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 $\lceil\alpha\rceil$: substituting τ for α in the suspension, requires translating the type to $\tau^{\langle C\rangle}$. 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.

REFERENCES

Amal Ahmed. 2006. Step-Indexed Syntactic Logical Relations for Recursive and Quantified Types. In *European Symposium on Programming (ESOP)*. 69–83.

Amal Ahmed and Matthias Blume. 2008. Typed Closure Conversion Preserves Observational Equivalence. In *International Conference on Functional Programming (ICFP), Victoria, British Columbia, Canada*. 157–168.

Amal Ahmed and Matthias Blume. 2011. An Equivalence-Preserving CPS Translation via Multi-Language Semantics. In *International Conference on Functional Programming (ICFP), Tokyo, Japan*. 431–444.

Amal Ahmed, Derek Dreyer, and Andreas Rossberg. 2009. State-Dependent Representation Independence. In *ACM Symposium on Principles of Programming Languages (POPL), Savannah, Georgia*.

Amal Jamil Ahmed. 2004. *Semantics of Types for Mutable State*. Ph.D. Dissertation. Princeton University.

Nick Benton and Chung-Kil Hur. 2009. Biorthogonality, Step-Indexing and Compiler Correctness. In *International Conference on Functional Programming (ICFP), Edinburgh, Scotland*.

Nick Benton, Andrew Kennedy, and George Russell. 1998. Compiling Standard ML to Java Bytecodes. In *International Conference on Functional Programming (ICFP), Baltimore, Maryland, USA*. 129–140. http://doi.acm.org/10.1145/289423.289435

Christos Dimoulas, Sam Tobin-Hochstadt, and Matthias Felleisen. 2012. Complete Monitors for Behavioral Contracts. In *European Symposium on Programming (ESOP)*.

Derek Dreyer, Georg Neis, and Lars Birkedal. 2012. The Impact of Higher-Order State and Control Effects on Local Relational Reasoning. *Journal of Functional Programming* 22, 4&5 (2012), 477–528.

Chung-Kil Hur and Derek Dreyer. 2011. A Kripke logical relation between ML and assembly. In *ACM Symposium on Principles of Programming Languages (POPL), Austin, Texas*.

Jeehoon Kang, Yoonseung Kim, Chung-Kil Hur, Derek Dreyer, and Viktor Vafeiadis. 2016. Lightweight Verification of Separate Compilation. In *ACM Symposium on Principles of Programming Languages (POPL), St. Petersburg, Florida*. ACM, 178–190.

Shriram Krishnamurthi, Peter Walton Hopkins, Jay Mccarthy, Paul T. Graunke, Greg Pettyjohn, and Matthias Felleisen. 2007. Implementation and use of the PLT Scheme web server. (2007).

Phillip Mates, Jamie Perconti, and Amal Ahmed. 2019. Under Control: Compositionally Correct Closure Conversion with Mutable State (Technical Appendix). (July 2019). Available at `http://www.ccs.neu.edu/home/amal/papers/refcc-tr.pdf`.

Jacob Matthews and Robert Bruce Findler. 2007. Operational Semantics for Multi-Language Programs. In *ACM Symposium on Principles of Programming Languages (POPL), Nice, France*. 3–10.

Yasuhiko Minamide, Greg Morrisett, and Robert Harper. 1996. Typed Closure Conversion. In *ACM Symposium on Principles of Programming Languages (POPL), St. Petersburg Beach, Florida*. 271–283.

Greg Morrisett, David Walker, Karl Crary, and Neal Glew. 1999. From System F to Typed Assembly Language. *ACM Transactions on Programming Languages and Systems* 21, 3 (May 1999), 527–568.

Georg Neis. 2018. *Compositional Compiler Correctness via Parametric Simulations*. Ph.D. Dissertation. Saarland University.

Georg Neis, Chung-Kil Hur, Jan-Oliver Kaiser, Craig McLaughlin, Derek Dreyer, and Viktor Vafeiadis. 2015. Pilsner: A Compositionally Verified Compiler for a Higher-Order Imperative Language. In *International Conference on Functional Programming (ICFP), Vancouver, British Columbia, Canada*.

Max S. New, William J. Bowman, and Amal Ahmed. 2016. Fully Abstract Compilation via Universal Embedding. In *International Conference on Functional Programming (ICFP), Nara, Japan*.

Marco Patrignani, Amal Ahmed, and Dave Clarke. 2019. Formal Approaches to Secure Compilation: A Survey of Fully Abstract Compilation and Related Work. *Comput. Surveys* 51, 6, Article 125 (Feb. 2019), 36 pages.

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

Daniel Patterson and Amal Ahmed. 2019. The Next 700 Compiler Correctness Theorems (Functional Pearl). *PACMPL* 3, ICFP (Aug. 2019).

Daniel Patterson, Jamie Perconti, Christos Dimoulas, and Amal Ahmed. 2017. FunTAL: Reasonably Mixing a Functional Language with Assembly. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Barcelona, Spain*.

James T. Perconti and Amal Ahmed. 2014. Verifying an Open Compiler Using Multi-Language Semantics. In *European Symposium on Programming (ESOP)*.

Christian Queinnec. 2003. Inverting Back the Inversion of Control or, Continuations Versus Page-centric Programming. *SIGPLAN Not.* 38, 2 (Feb. 2003), 57–64.

Gabriel Scherer, Max S. New, Nick Rioux, and Amal Ahmed. 2018. FabULous Interoperability for ML and a Linear Language. In *FOSSACS*.

Gordon Stewart, Lennart Beringer, Santiago Cuellar, and Andrew W. Appel. 2015. Compositional CompCert. In *ACM Symposium on Principles of Programming Languages (POPL), Mumbai, India*.

James Gordon Stewart. 2015. *Verified Separate Compilation for C*. Ph.D. Dissertation. Princeton University.

Peng Wang, Santiago Cuellar, and Adam Chlipala. 2014. Compiler Verification Meets Cross-Language Linking via Data Abstraction. In *ACM Symposium on Object Oriented Programming: Systems, Languages, and Applications (OOPSLA)*.

Yuting Wang, Pierre Wilke, and Zhong Shao. 2019. An Abstract Stack Based Approach to Verified Compositional Compilation to Machine Code. In *ACM Symposium on Principles of Programming Languages (POPL), Lisbon, Portugal*.