Verifying an Open Compiler Using Multi-Language Semantics

James T. Perconti and Amal Ahmed

Northeastern University

Abstract. Existing verified compilers are proved correct under a closed-world assumption, i.e., that the compiler will only be used to compile *whole* programs. We present a new methodology for verifying correct compilation of program *components*, while formally allowing linking with target code of arbitrary provenance. To demonstrate our methodology, we present a two-pass type-preserving open compiler and prove that compilation preserves semantics. The central novelty of our approach is that we define a combined language that embeds the source, intermediate, and target languages and formalizes a semantics of interoperability between them, using boundaries in the style of Matthews and Findler. Compiler correctness is stated as contextual equivalence in the combined language.

Note to reader: We use blue, red, and purple to typeset terms in various languages. This paper will be difficult to follow unless read/printed in color.

1 Introduction

There has been remarkable progress on formally verified compilers over the last few years, with researchers proving the correctness of increasingly sophisticated compilers for increasingly realistic languages. The most well known instance of this is the CompCert compiler [1, 2] which uses the Coq proof assistant to both implement and verify a multi-pass optimizing compiler from C to PowerPC, ARM, and x86 assembly, proving that the compiler preserves semantics of source programs. Several other compiler-verification efforts have successfully followed CompCert's lead and basic methodology, for instance, focusing on multithreaded Java [3], just-in-time compilation [4], and C with relaxed memory concurrency [5].

Unfortunately, these projects prove compiler correctness under a *closed-world* assumption, that is, assuming that the verified compiler will always compile *whole* programs. Despite the immense effort put into verification, the compiler correctness theorem provides no guarantees about correct compilation of *components*. This whole-program assumption is completely unrealistic since most software systems today are comprised of many components written in different languages compiled to a common target, as well as runtime-library routines that may be handwritten in the target language. We need compiler correctness theorems applicable to the way we actually use these compilers.

Formally verifying that components are compiled correctly—often referred to as *compositional compiler correctness*—is a challenging problem. A key difficulty is that, in the setting of compiling components, it is not clear how to even state the compiler correctness theorem. CompCert's compiler correctness theorem is easy to state thanks to the whole program assumption: informally, it says that if a source program P_S compiles to a target program P_T, then running P_S and P_T results in the same trace of observable

events. The same sort of theorem does not make sense when we compile a component e_S to a component e_T: we cannot "run" a component since it is not a complete program.

Intuitively, we want the compiler correctness theorem to say that if a component e_S compiles to e_T, then some desired relationship $e_S \simeq e_T$ holds between e_S and e_T. The central question is: how do we *formally specify* $e_S \simeq e_T$? To answer this question, we must consider how the compiled component is actually used: it needs to be linked with some e'_T, creating a whole program that can be run. Informally, the compiler correctness theorem should guarantee that if we link e_T with e'_T, then the resulting target-level program should correspond to the source component e_S linked with e'_T. But, formally speaking, how can one link a source component with a target component and what are the rules for running the resulting source-target hybrid? These questions demand a *semantics of interoperability* between the source and target languages. We give our semantics of interoperability as a multi-language operational model. We then define $e_S \simeq e_T$ as a contextual equivalence in that model.

There are two other important issues to consider when evaluating a compositional compiler correctness theorem and its supporting formalism. The first is the degree of *horizontal compositionality* that the model allows, that is, which target components e'_T may formally be linked with a compiled component. At the lower end of the horizontal compositionality spectrum are *fully abstract* compilers. Full abstraction states that the compiler both preserves and reflects contextual equivalence. Hence, a fully abstract compiler preserves all of the source language's abstractions, and compiled components are only allowed to link with components that can be expressed in the source language.

But real systems often link together components from multiple languages with different guarantees and different expressive power. We are particularly interested in supporting interoperability between parametric typed languages like ML and low-level languages like C. Thus, full abstraction is often too restrictive. To support the whole programs that we actually run, the compiler correctness theorem should formally support linking with as large a class of programs as possible, and in particular, should not require an e'_T to have been compiled from the same source language as e_T.

Abandoning full abstraction in favor of greater horizontal compositionality does not require giving up all the guarantees of the source language. The compiler and its verification framework can be designed to preserve the source-level equivalences that are critically needed without forbidding all foreign behavior. To show that different levels of abstraction preservation are possible, we will deliberately pick a target language that is more expressive than the source and design our compiler so that it is *not* fully abstract. Our focus in this paper is on how to preserve the representation independence and information hiding guarantees provided by type abstraction in our source language.

The second important issue for a compiler correctness framework is that we want to be able to verify multi-pass compilers. For example, if we have a two-pass compiler that compiles a source component e_S to an intermediate-language component e_I to a target component e_T, we should be able verify each pass separately, showing $e_S \simeq e_I$ and $e_I \simeq e_T$, and then compose these results to get a correctness theorem for the whole compiler saying $e_S \simeq e_T$. This is typically referred to as *vertical compositionality*.

We will show that our approach of using a multi-language operational model succeeds at both horizontal and vertical compositionality. In particular, we validate our

methodology by applying it to a two-pass type-preserving compiler. The compiler deals with three languages: our source language F (System F with existential and recursive types), an intermediate language C (the target of a typed closure conversion pass), and our target language A (the target of a heap allocation pass).[1] The target language A allows tuples and closures to live only on the heap and supports both mutable and immutable references. Our closure conversion pass translates F components of type τ to C components of type $\tau^{\mathcal{C}}$, where $\tau^{\mathcal{C}}$ denotes the type translation of τ. The subsequent allocation pass translates C components of type $\boldsymbol{\tau}$ to A components of type $\boldsymbol{\tau}^{\mathcal{A}}$, where $\boldsymbol{\tau}^{\mathcal{A}}$ is the type translation of $\boldsymbol{\tau}$.

To define the semantics of interoperability between these languages, we embed them all into one language, FCA, and add syntactic boundary forms between each pair of adjacent languages, in the style of Matthews and Findler [7] and of Ahmed and Blume [8]. For instance, the term $\mathcal{CF}^{\tau}(\mathsf{e_F})$ allows an F component $\mathsf{e_F}$ of type τ to be used as a C component of type $\tau^{\mathcal{C}}$, while ${}^{\tau}\mathcal{FC}(\mathbf{e_C})$ allows a C component $\mathbf{e_C}$ of translation type $\tau^{\mathcal{C}}$ to be used as an F component of type τ. Similarly, we have boundary forms $\mathcal{AC}$ and $\mathcal{CA}$ for the next language pair. Non-adjacent languages can interact by stacking up boundaries: for example, $\mathcal{FC}(\mathcal{CA}\,\mathbf{e_A})$ (abbreviated $\mathcal{FCA}(\mathbf{e_A})$) allows an A component $\mathbf{e_A}$ to be embedded in an F term.

FCA *Design Principles* Our goal is for the FCA interoperability semantics to give us a useful specification of when a component in one of the underlying languages should be considered equivalent to a component in another language. We realize that goal by following three principles.

First, we define the operational semantics of FCA so that the original languages are *embedded* into FCA unchanged: running an FCA program that's written solely in one of the embedded languages is identical to running it in that language alone. For instance, execution of the A program $\mathbf{e_A}$ proceeds in exactly the same way whether we use the operational semantics of A or the augmented semantics for FCA.

Next, we ensure that the typing rules are similarly embedded: a component that contains syntax from only one underlying language should typecheck under that language's individual type system if and only if it typechecks under FCA's type system.

The final property we need is *boundary cancellation*, which says that wrapping two opposite language boundaries around a component yields the same behavior as the underlying component with no boundaries. For example, any $\mathsf{e_F} : \tau$ must be contextually equivalent to ${}^{\tau}\mathcal{FC}(\mathcal{CF}^{\tau}\mathsf{e_F})$, and any $\mathbf{e_C} : \tau^{\mathcal{C}}$ must be equivalent to $\mathcal{CF}^{\tau}({}^{\tau}\mathcal{FC}\mathbf{e_C})$.

Compiler Correctness We state the correctness criterion for our compiler as a contextual equivalence. For each pass of the compiler from a source S to a target T, where S and T interoperate via boundaries $\mathcal{ST}$ and $\mathcal{TS}$, define our source-target relationship by

$$e_S \simeq e_T \stackrel{\text{def}}{=} e_S \approx^{ctx}_{\mathsf{FCA}} {}^{\tau}\mathcal{ST}(e_T) : \tau.$$

We prove that if $e_S : \tau$ compiles to e_T, then $e_S \simeq e_T$. Since contextual equivalence is transitive, our framework achieves vertical compositionality immediately: it is easy to combine the two correctness proofs for the individual compiler passes, giving the overall correctness result that if $\mathsf{e_F}$ compiles to $\mathbf{e_A}$, then $\mathsf{e_F} \simeq \mathbf{e_A}$, or

$$\mathsf{e_F} \approx^{ctx}_{\mathsf{FCA}} {}^{\tau}\mathcal{FCA}(\mathbf{e_A}) : \tau.$$

[1] We have extended our F to A compiler with a code-generation pass to an assembly language, much like Morrisett *et al.*'s stack-based TAL [6]. We will report on that work in a future paper.

Reasoning About Linking Our approach enjoys a strong horizontal compositionality property: we can link with any target component e'_A that has an appropriate type, with no requirement that e'_A was produced by any particular means or from any particular source language. Specifically, if e_F expects to be linked with a component of type τ' and compiles to e_A, then e_A will expect to be linked with a component of type $((\tau')^{\mathcal{C}})^{\mathcal{A}}$. If e'_A has this type, then using our compiler correctness theorem, we can conclude that

$$(\mathsf{e}_\mathsf{F}\ {}^{\tau'}\mathcal{FCA}(\mathsf{e}'_\mathsf{A})) \approx^{ctx} \mathcal{FCA}(\mathsf{e}_\mathsf{A}\ \mathsf{e}'_\mathsf{A}),$$

or equivalently,

$$\mathcal{ACF}(\mathsf{e}_\mathsf{F}\ {}^{\tau'}\mathcal{FCA}(\mathsf{e}'_\mathsf{A})) \approx^{ctx} \mathsf{e}_\mathsf{A}\ \mathsf{e}'_\mathsf{A}.$$

The right-hand side of this equality is exactly the A program we ultimately want to run, and the left-hand side is an FCA program that models that program.

Contributions Our main contributions are our methodology and that we have proven correctness for an open multi-pass compiler. We have designed a multi-language semantics that lets us state a strong compiler-correctness theorem, and to prove the theorems, we have developed a logical relation for proving contextual equivalences between FCA components. The most significant technical challenges were related to interoperability between languages with type abstraction, specifically, in designing the multi-language semantics so it preserves type abstraction between languages (§5), and in designing the parts of the logical relation that model the handling of type abstraction in a multi-language setting (§9).

Due to space constraints, we elide various technical details and omit proofs. All definitions, lemmas, and proofs are spelled out in full detail in the accompanying technical report [9], available at: `http://ccs.neu.edu/home/amal/voc/`

2 Related Work: Benton-Hur Approach

Before beginning our technical development, we compare our methodology to the only prominent existing approach to compositional compiler correctness.

To eliminate the closed-world assumption, Benton and Hur [10] advocate setting up a logical relation between the source and target languages, specifying when a source term semantically approximates target code and vice versa. We will refer to a logical relation that relates terms from two *different* languages as a *cross-language* logical relation. The relation is defined by induction on source-language types. Benton and Hur verified a compiler from the simply-typed λ-calculus with recursion [10]—and later, from System F with recursion [11]—to an SECD machine, proving that if source component e_S compiles to target code e_T, then e_S and e_T are logically related. Later, Hur and Dreyer [12] used essentially the same approach to prove correctness of a compiler from an idealized ML to assembly.

However, the Benton-Hur (henceforth, BH) approach suffers from serious drawbacks in both vertical and horizontal compositionality. First, the cross-language framework does not scale to a multi-pass compiler. Both Benton-Hur and Hur-Dreyer handle only a single pass. To achieve vertical compositionality in the BH style, one would have to define separate cross-language logical relations relating the source and target of each compiler pass, and then prove that the logical relations compose transitively in order to establish that the correctness of each pass implies correctness for the entire compiler. But this kind of transitive composition of cross-language logical relations has

been an open problem for some time. (We'll discuss recent work towards addressing this problem in §11.)

The second drawback to the BH approach is its limited horizontal compositionality. Consider the situation where a verified compiler from language S to language T is used to compile a source component e_S to some target code e_T. The BH compiler correctness theorem tells us that e_S and e_T are logically related. We wish to link the compiled code e_T with some other target code e'_T and verify the resulting program. To do this using the BH framework, we must now *come up with a source-level component* e'_S and show that it is logically related to e'_T. This is an onerous requirement: while it may be reasonable to come up with e'_S when the given e'_T is very simple, it seems almost impossible when e'_T consists of hundreds of lines of assembly! Further, if e'_T is compiled from some other source language R, it may not even be possible to write down an e'_S in language S that is related to e'_T.

Technically speaking, the BH approach does support linking with any target code that can be proved logically related to a source component. But it cannot support linking with any components that are not expressible in the source language. And we contend that even for the theoretically-allowed cases, in practice the approach is limited to allowing linking between only very simple components or components that were all compiled from the same source language.

Overcoming BH Limitations By reasoning about components in the FCA setting, we can overcome both limitations of the BH framework. We have already pointed out that our framework admits vertical compositionality thanks to the transitivity of contextual equivalence.

For the second limitation of the BH approach, consider a target component e'_{A}. While the BH approach would need to find a related source component to fit e'_{A} into their framework, we only need to find an FCA component that looks like a source component. Specifically, we can use e'_{A} itself in a source context by wrapping it in appropriate boundaries: $\mathcal{FCA}(\mathsf{e}'_{\mathsf{A}})$.

3 The Languages

We begin our technical development with a few notes on typesetting and notational conventions. We typeset the terms, types, and contexts of our various languages as follows:

- F (System F) in a blue sans-serif font;
- C (Closure conversion) in a **red bold font with serifs**;
- A (Allocation) in a **purple sans-serif bold font**.

For each of our languages, we will use the metavariable e for *components* and t for *terms*. In the first two languages, F and C, terms and components coincide, but the distinction will be meaningful in language A. Similarly, all languages use τ for types, v for values, 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. Δ or Γ or Ψ) are joined by a comma, this should be interpreted as a *disjoint* union.

$\tau ::= \alpha \mid \mathsf{unit} \mid \mathsf{int} \mid \forall[\overline{\alpha}].(\overline{\tau}) \to \tau \mid \langle \overline{\tau} \rangle \mid \exists \alpha.\tau \mid \mu\alpha.\tau$

$\mathsf{e} ::= \mathsf{t}$

$\mathsf{t} ::= \mathsf{x} \mid () \mid \mathsf{n} \mid \mathsf{t}\ \mathsf{p}\ \mathsf{t} \mid \mathsf{if0}\ \mathsf{t}\ \mathsf{t}\ \mathsf{t} \mid \lambda[\overline{\alpha}](\overline{\mathsf{x}:\tau}).\mathsf{t} \mid \mathsf{t}\,[\overline{\tau}]\,\overline{\mathsf{t}} \mid \langle \overline{\mathsf{t}} \rangle \mid \pi_i(\mathsf{t}) \mid \mathsf{pack}\langle \tau, \mathsf{t} \rangle\ \mathsf{as}\ \exists \alpha.\tau$
$\quad \mid \mathsf{unpack}\ \langle \alpha, \mathsf{x} \rangle = \mathsf{t}\ \mathsf{in}\ \mathsf{t} \mid \mathsf{fold}_{\mu\alpha.\tau}\ \mathsf{t} \mid \mathsf{unfold}\ \mathsf{t}$

$\mathsf{p} ::= + \mid - \mid *$

$\mathsf{v} ::= () \mid \mathsf{n} \mid \lambda[\overline{\alpha}](\overline{\mathsf{x}:\tau}).\mathsf{t} \mid \langle \overline{\mathsf{v}} \rangle \mid \mathsf{pack}\langle \tau, \mathsf{v} \rangle\ \mathsf{as}\ \exists \alpha.\tau \mid \mathsf{fold}_{\mu\alpha.\tau}\ \mathsf{v}$

$\mathsf{E} ::= [\cdot] \mid \mathsf{E}\ \mathsf{p}\ \mathsf{t} \mid \mathsf{v}\ \mathsf{p}\ \mathsf{E} \mid \mathsf{if0}\ \mathsf{E}\ \mathsf{t}\ \mathsf{t} \mid \mathsf{E}\,[\overline{\tau}]\,\overline{\mathsf{t}} \mid \mathsf{v}\,[\overline{\tau}]\,\overline{\mathsf{v}}\,\mathsf{E}\,\overline{\mathsf{t}} \mid \ldots$

$\boxed{\mathsf{e} \longmapsto \mathsf{e}'}$ $\quad \mathsf{E}[\lambda[\overline{\alpha}](\overline{\mathsf{x}:\tau}).\mathsf{t}\,[\overline{\tau'}]\,\overline{\mathsf{v}}] \longmapsto \mathsf{E}[\mathsf{t}[\overline{\tau'/\alpha}]\,[\overline{\mathsf{v}/\mathsf{x}}]] \quad \ldots$

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

$\tau ::= \boldsymbol{\alpha} \mid \mathbf{unit} \mid \mathbf{int} \mid \forall[\overline{\alpha}].(\overline{\tau}) \to \tau \mid \langle \overline{\tau} \rangle \mid \exists \boldsymbol{\alpha}.\tau \mid \boldsymbol{\mu\alpha}.\tau$

$\mathbf{e} ::= \mathbf{t}$

$\mathbf{t} ::= \mathbf{x} \mid () \mid \mathbf{n} \mid \mathbf{t}\ \mathbf{p}\ \mathbf{t} \mid \mathbf{if0}\ \mathbf{t}\ \mathbf{t}\ \mathbf{t} \mid \boldsymbol{\lambda}[\overline{\alpha}](\overline{\mathbf{x}:\tau}).\mathbf{t} \mid \mathbf{t}\,[]\,\overline{\mathbf{t}} \mid \mathbf{t}[\tau] \mid \langle \overline{\mathbf{t}} \rangle \mid \pi_i(\mathbf{t})$
$\quad \mid \mathbf{pack}\langle \tau, \mathbf{t} \rangle\ \mathbf{as}\ \exists \boldsymbol{\alpha}.\tau \mid \mathbf{unpack}\ \langle \alpha, \mathbf{x} \rangle = \mathbf{t}\ \mathbf{in}\ \mathbf{t} \mid \mathbf{fold}_{\mu\alpha.\tau}\ \mathbf{t} \mid \mathbf{unfold}\ \mathbf{t}$

$\mathbf{p} ::= + \mid - \mid *$

$\mathbf{v} ::= () \mid \mathbf{n} \mid \boldsymbol{\lambda}[\overline{\alpha}](\overline{\mathbf{x}:\tau}).\mathbf{t} \mid \langle \overline{\mathbf{v}} \rangle \mid \mathbf{pack}\langle \tau, \mathbf{v} \rangle\ \mathbf{as}\ \exists \boldsymbol{\alpha}.\tau \mid \mathbf{fold}_{\mu\alpha.\tau}\ \mathbf{v} \mid \mathbf{v}[\tau]$

$\mathbf{E} ::= [\cdot] \mid \ldots \mid \mathbf{E}\,[]\,\overline{\mathbf{t}} \mid \mathbf{v}\,[\overline{\tau}]\,\overline{\mathbf{v}}\,\mathbf{E}\,\overline{\mathbf{t}} \mid \mathbf{E}[\tau] \mid \ldots$

$\boxed{\mathbf{e} \longmapsto \mathbf{e}'}$ $\quad \mathbf{E}[\boldsymbol{\lambda}[\overline{\alpha}](\overline{\mathbf{x}:\tau}).\mathbf{t}\,[\overline{\tau'}]\,\overline{\mathbf{v}}] \longmapsto \mathbf{E}[\mathbf{t}[\overline{\tau'/\alpha}]\,[\overline{\mathbf{v}/\mathbf{x}}]] \quad \ldots$

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

$$\frac{\overline{\alpha}; \overline{\mathbf{x}:\tau} \vdash \mathbf{t} : \tau'}{\Delta; \Gamma \vdash \boldsymbol{\lambda}[\overline{\alpha}](\overline{\mathbf{x}:\tau}).\mathbf{t} : \forall[\overline{\alpha}].(\overline{\tau}) \to \tau'} \qquad \frac{\Delta; \Gamma \vdash \mathbf{t} : \forall[].(\overline{\tau}) \to \tau' \qquad \Delta; \Gamma \vdash \overline{\mathbf{t}} : \overline{\tau}}{\Delta; \Gamma \vdash \mathbf{t}\,[]\,\overline{\mathbf{t}} : \tau'}$$

$$\frac{\Delta; \Gamma \vdash \mathbf{t} : \forall[\beta, \overline{\alpha}].(\overline{\tau}) \to \tau' \qquad \Delta \vdash \tau_0}{\Delta; \Gamma \vdash \mathbf{t}[\tau_0] : \forall[\overline{\alpha}].(\overline{\tau[\tau_0/\beta]}) \to \tau'[\tau_0/\beta]} \quad \ldots$$

$\tau ::= \alpha \mid \mathsf{unit} \mid \mathsf{int} \mid \exists \alpha.\tau \mid \mu\alpha.\tau \mid \mathsf{ref}\ \psi \mid \mathsf{box}\ \psi$

$\psi ::= \forall[\overline{\alpha}].(\overline{\tau}) \to \tau \mid \langle \tau, \ldots, \tau \rangle$

$\mathsf{e} ::= (\mathsf{t}, \mathsf{H})$

$\mathsf{t} ::= \mathsf{x} \mid () \mid \mathsf{n} \mid \mathsf{t}\ \mathsf{p}\ \mathsf{t} \mid \mathsf{if0}\ \mathsf{t}\ \mathsf{t}\ \mathsf{t} \mid \ell \mid \mathsf{t}\,[]\,\overline{\mathsf{t}} \mid \mathsf{t}[\tau] \mid \mathsf{pack}\langle \tau, \mathsf{t} \rangle\ \mathsf{as}\ \exists \alpha.\tau \mid \mathsf{unpack}\ \langle \alpha, \mathsf{x} \rangle = \mathsf{t}\ \mathsf{in}\ \mathsf{t}$
$\quad \mid \mathsf{fold}_{\mu\alpha.\tau}\ \mathsf{t} \mid \mathsf{unfold}\ \mathsf{t} \mid \mathsf{ralloc}\ \langle \overline{\mathsf{t}} \rangle \mid \mathsf{balloc}\ \langle \overline{\mathsf{t}} \rangle \mid \mathsf{read}[\mathsf{i}]\ \mathsf{t} \mid \mathsf{write}\ \mathsf{t}\ [\mathsf{i}] \leftarrow \mathsf{t}$

$\mathsf{p} ::= + \mid - \mid *$

$\mathsf{v} ::= () \mid \mathsf{n} \mid \mathsf{pack}\langle \tau, \mathsf{v} \rangle\ \mathsf{as}\ \exists \alpha.\tau \mid \mathsf{fold}_{\mu\alpha.\tau}\ \mathsf{v} \mid \ell \mid \mathsf{v}[\tau]$

$\mathsf{E} ::= (\mathsf{E}_t, \cdot) \qquad \mathsf{E}_t ::= [\cdot] \mid \ldots \mid \mathsf{balloc}\ \langle \overline{\mathsf{v}}, \mathsf{E}_t, \overline{\mathsf{t}} \rangle \mid \ldots$

$\mathsf{h} ::= \lambda[\overline{\alpha}](\overline{\mathsf{x}:\tau}).\mathsf{t} \mid \langle \mathsf{v}, \ldots, \mathsf{v} \rangle \qquad \mathsf{H} ::= \cdot \mid \mathsf{H}, \ell \mapsto \mathsf{h}$

$\boxed{\langle \mathsf{H} \mid \mathsf{e} \rangle \longmapsto \langle \mathsf{H}' \mid \mathsf{e}' \rangle}$ Reduction Relation (selected cases)

$$\langle \mathsf{H} \mid (\mathsf{t}, (\mathsf{H}', \ell \mapsto \mathsf{h})) \rangle \longmapsto \langle \mathsf{H}, \ell' \mapsto \mathsf{h} \mid (\mathsf{t}[\ell'/\ell], \mathsf{H}'[\ell'/\ell]) \rangle \quad \text{if } \ell' \notin \mathrm{dom}(\mathsf{H})$$

$$\langle \mathsf{H} \mid \mathsf{E}[\ell\,[\overline{\tau'}]\,\overline{\mathsf{v}}] \rangle \longmapsto \langle \mathsf{H} \mid \mathsf{E}[\mathsf{t}[\overline{\tau'/\alpha}][\overline{\mathsf{v}/\mathsf{x}}]] \rangle \quad \text{if } \mathsf{H}(\ell) = \lambda[\overline{\alpha}](\overline{\mathsf{x}:\tau}).\mathsf{t}$$

$\boxed{\Psi \vdash \mathsf{h} : \psi}$ where $\Psi ::= \cdot \mid \Psi, \ell : {}^{\mathsf{ref}}\psi \mid \Psi, \ell : {}^{\mathsf{box}}\psi$

$\boxed{\Psi \vdash \mathsf{H} : \Psi'}$ which implies $\mathrm{dom}(\Psi) \cap \mathrm{dom}(\Psi') = \emptyset$

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

$$\frac{\Psi \vdash \mathsf{H} : \Psi' \qquad (\Psi, \Psi'); \Delta; \Gamma \vdash \mathsf{t} : \tau}{\Psi; \Delta; \Gamma \vdash (\mathsf{t}, \mathsf{H}) : \tau} \quad \ldots$$

$$\frac{\Psi; \Delta; \Gamma \vdash \overline{\mathsf{t}} : \overline{\tau}}{\Psi; \Delta; \Gamma \vdash \mathsf{balloc}\ \langle \overline{\mathsf{t}} \rangle : \mathsf{box}\ \langle \overline{\tau} \rangle} \qquad \frac{\Psi; \Delta; \Gamma \vdash \mathsf{t} : \mathsf{box}\ \langle \tau_0, \ldots \tau_i \ldots, \tau_n \rangle}{\Psi; \Delta; \Gamma \vdash \mathsf{read}[\mathsf{i}]\ \mathsf{t} : \tau_i}$$

Fig. 1. Definition of F (top), C (middle), and A (bottom)

Source Language Our source language F is System F with recursive types, existential types, and tuples. The syntax of types and terms in F is shown in Figure 1 (top). 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 $\overline{\tau} \to \tau'$. We define a small-step operational semantics for F (written $\mathsf{e} \longmapsto \mathsf{e}'$) using evaluation contexts E to lift the primitive reductions to a standard left-to-right call-by-value semantics for the language. The reduction rules are standard; we show only the application rule.

F's typing judgment has the form $\Delta; \Gamma \vdash \mathsf{e} : \tau$. The type environment Δ tracks the type variables in scope. The value environment Γ tracks the term variables in scope along with their types τ, which must be well formed under Δ (written $\Delta \vdash \tau$ and defined as $\mathrm{ftv}(\tau) \subseteq \Delta$). The typing rules are standard and hence omitted.

Intermediate Language Our intermediate language C, shown in Figure 1 (middle), is nearly identical to F, with two exceptions. First, since this language is the target of closure conversion, functions are not allowed to contain free type or term variables. Second, we allow the partial application of a function to a type. Hence, C terms include $\mathbf{t}[\boldsymbol{\tau}]$ and we consider $\mathbf{v}[\boldsymbol{\tau}]$ to be a value.

The reduction relation $\mathbf{e} \longmapsto \mathbf{e}'$ is identical to that of F, and the typing judgment $\boldsymbol{\Delta}; \boldsymbol{\Gamma} \vdash \mathbf{e} : \boldsymbol{\tau}$ differs only in the rules for abstraction and application which are shown in the figure. Note that the body of a C function must typecheck in an environment that contains only the function's formal arguments.

Target Language Our target A must serve as a target for heap allocation. Its design is similar to the language λ^A from [13]. Since we are compiling a source language without mutable references, it would suffice for A to provide only immutable references to functions and tuples that must now live on the heap. However, to provide a concrete illustration of the ability to link with target code that cannot be expressed in the source language, we augment A with mutable references to tuples.

The language A is shown in Figure 1 (bottom). Functions in A are stored only in immutable cells on the heap, while tuples are stored in heap cells that can be either mutable or immutable. We use ψ for the types of these *heap values* h. Mutable and immutable references have types $\mathsf{ref}\ \psi$ and $\mathsf{box}\ \psi$, respectively. The terms $\mathsf{ralloc}\ \langle \overline{\mathsf{t}} \rangle$ and $\mathsf{balloc}\ \langle \overline{\mathsf{t}} \rangle$—which allocate mutable and immutable cells, respectively—each allocate a new location ℓ and initialize it to the given tuple. The instructions $\mathsf{read}[\mathsf{i}]\ \ell$ and $\mathsf{write}\ \ell\ [\mathsf{i}]\ \leftarrow \mathsf{v}$ respectively read from and write the value v to the i-th slot in the tuple (of length n) stored at ℓ, assuming $\mathsf{0} \leq \mathsf{i} < \mathsf{n}$. The type system ensures that writes are only performed on mutable tuples.

Unlike F and C, the syntax of A distinguishes components e from terms t. A component e pairs a term t with a *heap fragment* H. H can contain functions and tuples that t may use by referring to locations in H. Intuitively, we need this notion of components because a bare term t is not as expressive as C component. In particular, A does not provide any way to dynamically allocate a location and initialize it to a function. We discuss how the compiler produces components with heap fragments in §4.

Heap fragments are assigned heap types Ψ. A heap fragment may reference locations that are to be linked in by another component, so the judgment $\Psi \vdash \mathsf{H} : \Psi'$ includes an external heap type Ψ as an environment used in assigning H the type Ψ'. Here, Ψ' must provide types for exactly the locations in H. Each h in H must typecheck

under the disjoint union of the two heap types (Ψ, Ψ'). Similarly, a component (t, H) can reference both external locations and those bound by H, that is, locations in the domain of either the external heap type Ψ or of H.

Our operational semantics for A is a relation between configurations $\langle \mathsf{H} \mid \mathsf{e} \rangle$. Any code or data in the internal heap fragment of component e must be loaded into memory before it can be run. We formally capture this with a reduction rule that "loads" a component by merging its internal heap fragment with the external heap. When loading a component (t, H), we must rename the locations bound in H so that they do not conflict with the external heap. After the loading step, the term component t can be evaluated using standard reduction rules.

The structure of A components also entails a small change to the structure of evaluation contexts, which are defined in two layers: contexts E expect components e, and term contexts $\mathsf{E_t}$ expect terms t. Terms are plugged into term contexts in the obvious way. Plugging a component-level evaluation context $\mathsf{E} = (\mathsf{E_t}, \cdot)$ with a component e is defined by $(\mathsf{E_t}, \cdot)[(\mathsf{t}, \mathsf{H})] = (\mathsf{E_t}[\mathsf{t}], \mathsf{H})$

4 The Compiler

Compiling F *to* C Closure conversion collects a function's free term variables in a tuple called the *closure environment* that is passed as an additional argument to the function, thus turning the function into a closed term. The closed function is paired with its environment to create a *closure*. The basic idea of typed closure conversion goes back to Minamide *et al.* [14], whom we follow in using an existential type to abstract the type of the environment. This ensures that two functions with the same type but different free variables still have the same type after closure conversion: the abstract type hides the fact that the closures' environments have different types.

We must also rewrite 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.* [13] and directly substitute the types into the function. Like the latter, 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 closure-conversion pass compiles F terms of type τ to C terms of type $\tau^{\mathcal{C}}$. Figure 2 (top) presents the type translation $\tau^{\mathcal{C}}$ and some of the compilation rules. Since this is closure conversion, the only interesting parts are those that involve functions. The omitted rules are defined by structural recursion on terms.

Compiling C *to* A Our second compiler pass combines hoisting of functions with explicit allocation of tuples. It takes a C component (that is, just a C term t) of type τ, and produces an A term t as well as a heap fragment H with all the hoisted functions. The component (t, H) is the overall output, and has type $\tau^{\mathcal{A}}$ under an empty external heap. The heap fragment generated by the compiler does not contain tuples: the compiler translates C tuples by generating balloc expressions, not by putting them in a static heap fragment. The type translation and interesting parts of the term translation are shown in Figure 2 (bottom).

5 F and C Interoperability

5.1 The Basics

We now present a formal semantics for interoperability between F and C. For now, we define a combined language FC; in §6, we will extend this to FCA. Our FC multi-

$\boxed{\tau^{\mathcal{C}}}$ Type Translation

$$\alpha^{\mathcal{C}} = \alpha \quad \mathsf{unit}^{\mathcal{C}} = \mathbf{unit} \quad \mathsf{int}^{\mathcal{C}} = \mathbf{int} \quad \forall[\overline{\alpha}].(\overline{\tau}) \to \tau'^{\mathcal{C}} = \exists\beta.\langle(\forall[\overline{\alpha}].(\beta, \overline{\tau^{\mathcal{C}}}) \to \tau'^{\mathcal{C}}), \beta\rangle$$

$$\exists\alpha.\tau^{\mathcal{C}} = \exists\alpha.\tau^{\mathcal{C}} \qquad \mu\alpha.\tau^{\mathcal{C}} = \mu\alpha.\tau^{\mathcal{C}} \qquad \langle\tau_1, \ldots, \tau_n\rangle^{\mathcal{C}} = \langle\tau_1{}^{\mathcal{C}}, \ldots, \tau_n{}^{\mathcal{C}}\rangle$$

$\boxed{\Delta; \Gamma \vdash \mathsf{e} : \tau \rightsquigarrow \mathbf{e}}$ Compiler (implies $\Delta^{\mathcal{C}}; \Gamma^{\mathcal{C}} \vdash \mathbf{e} : \tau^{\mathcal{C}}$)

$$\frac{\mathsf{x} : \tau \in \Gamma}{\Delta; \Gamma \vdash \mathsf{x} : \tau \rightsquigarrow \mathbf{x}} \qquad \frac{}{\Delta; \Gamma \vdash () : \mathsf{unit} \rightsquigarrow ()} \qquad \frac{}{\Delta; \Gamma \vdash \mathsf{n} : \mathsf{int} \rightsquigarrow \mathbf{n}}$$

$$\frac{\begin{array}{c}\mathsf{y}_1, \ldots, \mathsf{y}_m = \mathrm{fv}(\lambda[\overline{\alpha}](\overline{\mathsf{x}:\tau}).\mathsf{t}) \qquad \beta_1, \ldots, \beta_k = \mathrm{ftv}(\lambda[\overline{\alpha}](\overline{\mathsf{x}:\tau}).\mathsf{t}) \\ \Delta, \overline{\alpha}; \Gamma, \overline{\mathsf{x}:\tau} \vdash \mathsf{t} : \tau' \rightsquigarrow \mathbf{t} \qquad \tau_{\mathbf{env}} = \langle(\Gamma(\mathsf{y}_1))^{\mathcal{C}}, \ldots, (\Gamma(\mathsf{y}_m))^{\mathcal{C}}\rangle \\ \mathbf{v} = \boldsymbol{\lambda}[\overline{\beta}, \overline{\alpha}](\mathbf{z} : \tau_{\mathbf{env}}, \overline{\mathbf{x} : \tau^{\mathcal{C}}}).(\mathbf{t}[\pi_1(\mathbf{z})/\mathbf{y}_1] \cdots [\pi_m(\mathbf{z})/\mathbf{y}_m])\end{array}}{\begin{array}{l}\Delta; \Gamma \vdash \lambda[\overline{\alpha}](\overline{\mathsf{x}:\tau}).\mathsf{t} : \forall[\overline{\alpha}].(\overline{\tau}) \to \tau' \rightsquigarrow \\ \qquad \mathbf{pack}\langle\tau_{\mathbf{env}}, \langle\mathbf{v}[\overline{\beta}], \langle\overline{\mathbf{y}}\rangle\rangle\rangle \ \mathbf{as}\ \exists\alpha'.\langle(\forall[\overline{\alpha}].(\alpha', \overline{\tau^{\mathcal{C}}}) \to \tau'^{\mathcal{C}}), \alpha'\rangle\end{array}}$$

$$\frac{\Delta; \Gamma \vdash \mathsf{t}_0 : \forall[\overline{\alpha}].(\overline{\tau_1}) \to \tau_2 \rightsquigarrow \mathbf{t_0} \qquad \Delta \vdash \overline{\tau} \qquad \Delta; \Gamma \vdash \overline{\mathsf{t}} : \overline{\tau_1[\tau/\alpha]} \rightsquigarrow \overline{\mathbf{t}}}{\Delta; \Gamma \vdash \mathsf{t}_0\,[\overline{\tau}]\,\overline{\mathsf{t}} : \tau_2[\overline{\tau/\alpha}] \rightsquigarrow \mathbf{unpack}\,\langle\boldsymbol{\beta}, \mathbf{z}\rangle = \mathbf{t_0}\ \mathbf{in}\ \pi_1(\mathbf{z})\,[\overline{\tau^{\mathcal{C}}}]\,\pi_2(\mathbf{z}), \overline{\mathbf{t}}}$$

$\boxed{\tau^{\mathcal{A}}}$ Type Translation

$$\alpha^{\mathcal{A}} = \alpha \quad \mathbf{unit}^{\mathcal{A}} = \mathbf{unit} \quad \mathbf{int}^{\mathcal{A}} = \mathbf{int} \quad \forall[\overline{\alpha}].(\overline{\tau}) \to \tau'^{\mathcal{A}} = \mathbf{box}\,\forall[\overline{\alpha}].(\overline{\tau^{\mathcal{A}}}) \to \tau'^{\mathcal{A}}$$

$$\exists\alpha.\tau^{\mathcal{A}} = \exists\alpha.\tau^{\mathcal{A}} \qquad \mu\alpha.\tau^{\mathcal{A}} = \mu\alpha.\tau^{\mathcal{A}} \qquad \langle\tau_1, \ldots, \tau_n\rangle^{\mathcal{A}} = \mathbf{box}\,\langle(\tau_1{}^{\mathcal{A}}), \ldots (\tau_n{}^{\mathcal{A}})\rangle$$

$\boxed{\Delta; \Gamma \vdash \mathbf{e} : \tau \rightsquigarrow (\mathbf{t}, \mathbf{H} : \Psi)}$ Compiler (implies $\cdot \vdash \mathbf{H} : \Psi$, and $\cdot; \Delta^{\mathcal{A}}; \Gamma^{\mathcal{A}} \vdash (\mathbf{t}, \mathbf{H}) : \tau^{\mathcal{A}}$)

$$\frac{\mathbf{x} : \tau \in \Gamma}{\Delta; \Gamma \vdash \mathbf{x} : \tau \rightsquigarrow (\mathbf{x}, \cdot : \cdot)} \qquad \frac{}{\Delta; \Gamma \vdash () : \mathbf{unit} \rightsquigarrow ((), \cdot : \cdot)} \qquad \ldots$$

$$\frac{\overline{\alpha}; \overline{\mathbf{x} : \tau} \vdash \mathbf{t} : \tau' \rightsquigarrow (\mathbf{t}, \mathbf{H} : \Psi)}{\begin{array}{l}\Delta; \Gamma \vdash \boldsymbol{\lambda}[\overline{\alpha}](\overline{\mathbf{x} : \tau}).\mathbf{t} : \forall[\overline{\alpha}].(\overline{\tau}) \to \tau' \rightsquigarrow \\ \qquad (\ell, (\mathbf{H}, \ell \mapsto \boldsymbol{\lambda}[\overline{\alpha}](\overline{\mathbf{x} : \tau^{\mathcal{A}}}).\mathbf{t}) : (\Psi, \ell : {}^{\mathbf{box}}\forall[\overline{\alpha}].(\overline{\tau^{\mathcal{A}}}) \to \tau'^{\mathcal{A}}))\end{array}}$$

$$\frac{\Delta; \Gamma \vdash \mathbf{t}_1 : \tau_1 \rightsquigarrow (\mathbf{t}_1, \mathbf{H}_1 : \Psi_1) \ \cdots \ \Delta; \Gamma \vdash \mathbf{t}_n : \tau_n \rightsquigarrow (\mathbf{t}_n, \mathbf{H}_n : \Psi_n)}{\begin{array}{l}\Delta; \Gamma \vdash \langle\mathbf{t}_1, \ldots, \mathbf{t}_n\rangle : \langle\tau_1, \ldots, \tau_n\rangle \rightsquigarrow \\ \qquad (\mathbf{balloc}\,\langle\mathbf{t}_1, \ldots, \mathbf{t}_n\rangle, (\mathbf{H}_1, \ldots, \mathbf{H}_n) : (\Psi_1, \ldots, \Psi_n))\end{array}}$$

Fig. 2. Compiler from F to C (top) and from C to A (bottom)

language system embeds the languages F and C so that both languages have natural access to foreign values (i.e., values from the other language). In particular, we want F components of type τ to be usable as C components of type $\tau^{\mathcal{C}}$, and vice versa. To allow cross-language communication, FC extends the original F and C with syntactic boundaries, written ${}^{\tau}\mathcal{FC}\,\mathbf{e}$ (C inside, F outside) and $\mathcal{CF}^{\tau}\,\mathsf{e}$ (F inside, C outside).

The interesting cases in the semantics of boundaries are those that handle universal and existential types. These must be defined carefully to ensure that type abstraction is not broken as values pass between languages. First, though, we explain the general principles of our boundary semantics by looking at the cases for simple types and their translations.

CF *Boundary Semantics* A term $\mathcal{CF}^{\tau}\mathsf{e}$ has type $\tau^{\mathcal{C}}$ if e has type τ. To evaluate this boundary term, CF's operational semantics require first that e be reduced to a value v (using F reduction rules). Then a type-directed meta-function is applied to v, yielding a value in C of type $\tau^{\mathcal{C}}$ (written $\mathbf{CF}^{\tau}(\mathsf{v}) = \mathbf{v}$). An important restriction on this meta-function, which we call the *value translation*, is that it is only defined for *closed* values. This is sufficient for our needs because it is used only by the FC operational semantics, and substitution-based reduction relations are defined only for closed programs. We can still write FC programs with free variables appearing under boundaries, but by the time we evaluate the boundary term, we will have supplied values for all of these free variables.

At base types, value translation is easy: for example, translating a value n of type int yields the same integer in C, $\mathbf{n}$. Most of the other types are translated simply by structural recursion.

The interesting case is the case for function types. Consider the translation of a value v of type $\tau \to \tau'$. As per the type translation, this should produce a value of type $\exists\beta.\langle((\beta, \tau^{\mathcal{C}}) \to \tau'^{\mathcal{C}}), \beta\rangle$. Since v is closed, we can simply use $\mathbf{unit}$ for the type β of the closure environment:

$$\mathbf{CF}^{\tau \to \tau'}(\mathsf{v}) = \mathbf{pack}\langle \mathbf{unit}, \langle \mathbf{v}, ()\rangle\rangle \ \mathbf{as}\ \exists\beta.\langle((\beta, \tau^{\mathcal{C}}) \to \tau'^{\mathcal{C}}), \beta\rangle$$

We must still construct the underlying function $\mathbf{v}$ for this closure, which we can do using boundary terms and the original function v:

$$\mathbf{v} = \boldsymbol{\lambda}(\mathbf{z} : \mathbf{unit}, \mathbf{x} : \tau^{\mathcal{C}}).\mathcal{CF}^{\tau'}(\mathsf{v}\ {}^{\tau}\mathcal{FC}\ \mathbf{x}).$$

The function we build simply translates its argument from C to F, applies v to the translated argument, and finally translates the result back into C.

The full translation rule for functions must also handle type arguments and requires some additional machinery, which we will discuss momentarily.

FC *Boundary Semantics* The term ${}^{\tau}\mathcal{FC}\ \mathbf{e}$ has type τ when $\mathbf{e}$ has type $\tau^{\mathcal{C}}$. As before, to evaluate a boundary term, we first evaluate the component under the boundary, this time to a value $\mathbf{v}$. Then we apply a value translation ${}^{\tau}\mathbf{FC}(\mathbf{v}) = \mathsf{v}$ that yields an F value v of type τ. Again, this translation is only defined for closed values of translation type.

Let us consider the type $\tau \to \tau'$ again. A closure $\mathbf{v}$ of type $(\tau \to \tau')^{\mathcal{C}}$ must be translated to an F function that first translates its argument from F to C, then unpacks the closure $\mathbf{v}$ and applies the code to its environment and the translated argument, and finally translates the result back from C to F:

$${}^{\tau \to \tau'}\mathbf{FC}(\mathbf{v}) = \lambda(\mathsf{x} : \tau).{}^{\tau'}\mathcal{FC}(\mathbf{unpack}\ \langle\beta, \mathbf{y}\rangle = \mathbf{v}\ \mathbf{in}\ \boldsymbol{\pi}_1(\mathbf{y})\ \boldsymbol{\pi}_2(\mathbf{y})\ \mathcal{CF}^{\tau}\mathsf{x})$$

In both function cases, notice that the direction of the conversion (and the boundary used) reverses for function arguments.

5.2 Handling Abstract Types

Now that we have established the general structure of boundary rules, we come to the interesting cases, those for abstract types.

FC *Type Abstraction* Consider the type $\forall[\alpha].(\alpha) \to \alpha$. Since $\alpha^{\mathcal{C}} = \boldsymbol{\alpha}$, the translation of this type is

$$(\forall[\alpha].(\alpha) \to \alpha)^{\mathcal{C}} = \exists\beta.\langle(\forall[\boldsymbol{\alpha}].(\beta, \boldsymbol{\alpha}) \to \boldsymbol{\alpha}), \beta\rangle.$$

If we naively try to extend the function case of the value translation given above, we get the following:

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

Note that we have not expanded $\alpha^{\mathcal{C}}$ in the application produced by this translation. It would expand to a C type variable $\boldsymbol{\alpha}$, but we cannot allow this, because that $\boldsymbol{\alpha}$ would be unbound! What we really want is that when α is instantiated with a concrete type τ, the positions inside language C where that type is needed receive $\tau^{\mathcal{C}}$.

We resolve this by making two changes to our system: first, we add a type $\lceil\alpha\rceil$ (which may be read as "α suspended in C") that allows an F type variable to appear in a C type. The F type variable α needs to be translated, but the translation is *delayed* until α is instantiated with a concrete type. We enforce this semantics in the definition of type substitution: $\lceil\alpha\rceil[\tau/\alpha] = \tau^{\mathcal{C}}$.

Second, we adjust the type translation to turn F type variables into suspended type variables instead of C type variables. We call this modified version of the type translation the *boundary type translation*, and notate it by $\tau^{\langle\mathcal{C}\rangle}$. Formally, the rule for type variables in the compiler's type translation is replaced by the rule $\alpha^{\langle\mathcal{C}\rangle} = \lceil\alpha\rceil$ in the boundary type translation. We only want to suspend free type variables, so when we translate a type that contains bound variables, we need to restore the behavior of the compiler's type translation when we translate the binding position. We can do this using a substitution, e.g., $(\exists\alpha.\tau)^{\langle\mathcal{C}\rangle} = \exists\boldsymbol{\alpha}.(\tau^{\langle\mathcal{C}\rangle}[\boldsymbol{\alpha}/\lceil\alpha\rceil])$. Thus the boundary type translation preserves the binding structure of the type to which it is applied.

With these two changes, we can correct the example above by replacing the appearance of $\alpha^{\mathcal{C}}$ with $\alpha^{\langle\mathcal{C}\rangle}$, and we get a sensible translation from C to F for values of type $(\forall[\alpha].(\alpha)\to\alpha)^{\mathcal{C}}$.

CF *Type Abstraction* Next, consider translating values of type $\forall[\alpha].(\alpha)\to\alpha$ from F into C. Once again, the existing machinery is not quite sufficient. Here is a naive attempt:

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

This time, we have translated the binder for α into a C binder for $\boldsymbol{\alpha}$, but we are left with free occurrences of α in the result! This is not a suitable translation, as we must produce a closed value. Note that the boundary terms in the body of $\mathbf{v}$ expect to be annotated with a type that translates to $\boldsymbol{\alpha}$.

To fix this problem, we introduce a *lump type* $\mathsf{L}\langle\boldsymbol{\tau}\rangle$ that allows us to pass C values to F terms as opaque lumps. The introduction form for the lump type is the boundary term ${}^{\mathsf{L}\langle\tau\rangle}\mathcal{FC}\mathbf{e}$, and the elimination form is $\mathcal{CF}^{\mathsf{L}\langle\tau\rangle}\mathsf{e}$. A pair of opposite boundaries at lump type cancel, to yield the underlying C value. We extend the boundary type translation by defining $\mathsf{L}\langle\boldsymbol{\tau}\rangle^{\langle\mathcal{C}\rangle} = \boldsymbol{\tau}$.

Now the three free occurrences of α in $\mathbf{v}$ can be replaced with $\mathsf{L}\langle\boldsymbol{\alpha}\rangle$, yielding a well-typed translation.

Summary With the additional tools of lumps, suspensions, and the boundary type translation, we have now developed everything needed for the FC multi-language system. Figure 3 presents more of the details, including the complete value translations.

$\tau ::= \cdots \mid \mathsf{L}\langle\boldsymbol{\tau}\rangle$ $\quad \boldsymbol{\tau} ::= \cdots \mid \lceil\alpha\rceil$ $\quad \tau ::= \tau \mid \boldsymbol{\tau}$

$\mathsf{t} ::= \cdots \mid {}^{\tau}\mathcal{FC}\,\mathsf{e}$ $\quad \mathsf{t} ::= \cdots \mid \mathcal{CF}^{\tau}\mathsf{e}$ $\quad e ::= \mathsf{e} \mid \mathbf{e}$

$\mathsf{v} ::= \cdots \mid {}^{\mathsf{L}\langle\tau\rangle}\mathcal{FC}\mathbf{v}$ $\quad \mathbf{v} ::= \cdots$ $\quad v ::= \mathsf{v} \mid \mathbf{v}$ $\quad \Delta ::= \cdot \mid \Delta, \alpha \mid \Delta, \boldsymbol{\alpha}$

$\mathsf{E} ::= \cdots \mid {}^{\tau}\mathcal{FC}\,\mathbf{E}$ $\quad \mathbf{E} ::= \cdots \mid \mathcal{CF}^{\tau}\mathsf{E}$ $\quad E ::= \mathsf{E} \mid \mathbf{E}$ $\quad \Gamma ::= \cdot \mid \Gamma, \mathsf{x}:\tau \mid \Gamma, \mathbf{x}:\boldsymbol{\tau}$

$\boxed{\tau^{\langle\mathcal{C}\rangle}}$ Boundary Type Translation

$$\forall[\overline{\alpha}].(\overline{\tau}) \to \tau'^{\langle\mathcal{C}\rangle} = \exists\boldsymbol{\beta}.\langle\left(\forall[\overline{\boldsymbol{\alpha}}].(\boldsymbol{\beta}, \overline{\tau^{\langle\mathcal{C}\rangle}[\boldsymbol{\alpha}/\lceil\alpha\rceil]}) \to \tau'^{\langle\mathcal{C}\rangle}[\boldsymbol{\alpha}/\lceil\alpha\rceil]\right), \boldsymbol{\beta}\rangle$$

$$\alpha^{\langle\mathcal{C}\rangle} = \lceil\alpha\rceil \qquad \mathsf{unit}^{\langle\mathcal{C}\rangle} = \mathbf{unit} \qquad \mathsf{int}^{\langle\mathcal{C}\rangle} = \mathbf{int} \qquad \exists\alpha.\tau^{\langle\mathcal{C}\rangle} = \exists\boldsymbol{\alpha}.(\tau^{\langle\mathcal{C}\rangle}[\boldsymbol{\alpha}/\lceil\alpha\rceil])$$

$$\mu\alpha.\tau^{\langle\mathcal{C}\rangle} = \boldsymbol{\mu\alpha}.(\tau^{\langle\mathcal{C}\rangle}[\boldsymbol{\alpha}/\lceil\alpha\rceil]) \qquad \langle\overline{\tau}\rangle^{\langle\mathcal{C}\rangle} = \langle\overline{\tau^{\langle\mathcal{C}\rangle}}\rangle \qquad \mathsf{L}\langle\boldsymbol{\tau}\rangle^{\langle\mathcal{C}\rangle} = \boldsymbol{\tau}$$

Type Substitution: $\lceil\alpha\rceil[\tau/\alpha] = \tau^{\langle\mathcal{C}\rangle}$

$\boxed{\Delta;\Gamma \vdash e:\tau}$ Include F and C rules, with environments replaced by $\Delta;\Gamma$

$$\frac{\Delta;\Gamma \vdash \mathbf{e}:\tau^{\langle\mathcal{C}\rangle}}{\Delta;\Gamma \vdash {}^{\tau}\mathcal{FC}\,\mathbf{e}:\tau} \qquad \frac{\Delta;\Gamma \vdash \mathsf{e}:\tau}{\Delta;\Gamma \vdash \mathcal{CF}^{\tau}\mathsf{e}:\tau^{\langle\mathcal{C}\rangle}}$$

$\boxed{\mathbf{CF}^{\tau}(\mathsf{v}) = \mathbf{v}}$ Value Translation $\quad \mathbf{CF}^{\mathsf{unit}}(()) = () \quad \mathbf{CF}^{\mathsf{int}}(\mathsf{n}) = \mathbf{n} \quad \mathbf{CF}^{\mathsf{L}\langle\tau\rangle}({}^{\mathsf{L}\langle\tau\rangle}\mathcal{FC}\mathbf{v}) = \mathbf{v}$

$$\mathbf{CF}^{\forall[\overline{\alpha}].(\overline{\tau})\to\tau'}(\mathsf{v}) = \mathbf{pack}\langle\mathbf{unit}, \langle\mathbf{v}, ()\rangle\rangle\ \mathbf{as}\ (\forall[\overline{\alpha}].(\overline{\tau}) \to \tau')^{\langle\mathcal{C}\rangle}$$

$$\text{where } \mathbf{v} = \boldsymbol{\lambda}[\overline{\boldsymbol{\alpha}}](\mathbf{z}:\mathbf{unit}, \overline{\mathbf{x}:\tau^{\langle\mathcal{C}\rangle}[\boldsymbol{\alpha}/\lceil\alpha\rceil]}).\mathcal{CF}^{\tau'[\mathsf{L}\langle\boldsymbol{\alpha}\rangle/\alpha]}(\mathsf{v}\,[\overline{\mathsf{L}\langle\boldsymbol{\alpha}\rangle}])\ \overline{\tau[\mathsf{L}\langle\boldsymbol{\alpha}\rangle/\alpha]\mathcal{FC}\mathbf{x}})$$

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

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

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

$\boxed{{}^{\tau}\mathbf{FC}(\mathbf{v}) = \mathsf{v}}$ Value Translation $\quad {}^{\mathsf{unit}}\mathbf{FC}(()) = () \quad {}^{\mathsf{int}}\mathbf{FC}(\mathbf{n}) = \mathsf{n} \quad {}^{\mathsf{L}\langle\tau\rangle}\mathbf{FC}(\mathbf{v}) = {}^{\mathsf{L}\langle\tau\rangle}\mathcal{FC}\mathbf{v}$

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

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

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

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

$\boxed{e \longmapsto e'}$ Include F and C rules, replacing eval. contexts E, $\mathbf{E}$ with E.

$$\frac{\mathbf{CF}^{\tau}(\mathsf{v}) = \mathbf{v}}{E[\mathcal{CF}^{\tau}\mathsf{v}] \longmapsto E[\mathbf{v}]} \qquad \frac{{}^{\tau}\mathbf{FC}(\mathbf{v}) = \mathsf{v} \qquad \tau \neq \mathsf{L}\langle\tau\rangle}{E[{}^{\tau}\mathcal{FC}\mathbf{v}] \longmapsto E[\mathsf{v}]}$$

Fig. 3. FC multi-language system (extends F and C from Figure 1)

The syntax of FC simply combines the syntax of F with that of C, and adds boundaries, lumps, and suspensions. The type judgment combines the type rules for F and C, but with the environments replaced by environments that can contain variables from both languages. We also add rules to typecheck boundary terms.

The cases of the value translations we have not yet covered mostly proceed by structural recursion, but note that the cases for existential types need to make use of lumps and suspensions (the suspensions are introduced by the boundary type translation) in ways that are dual to the function cases.

The reduction relation combines the reduction rules from F and C and adds rules for boundaries. The boundary reduction rules use the value translations to produce a value in the other language.

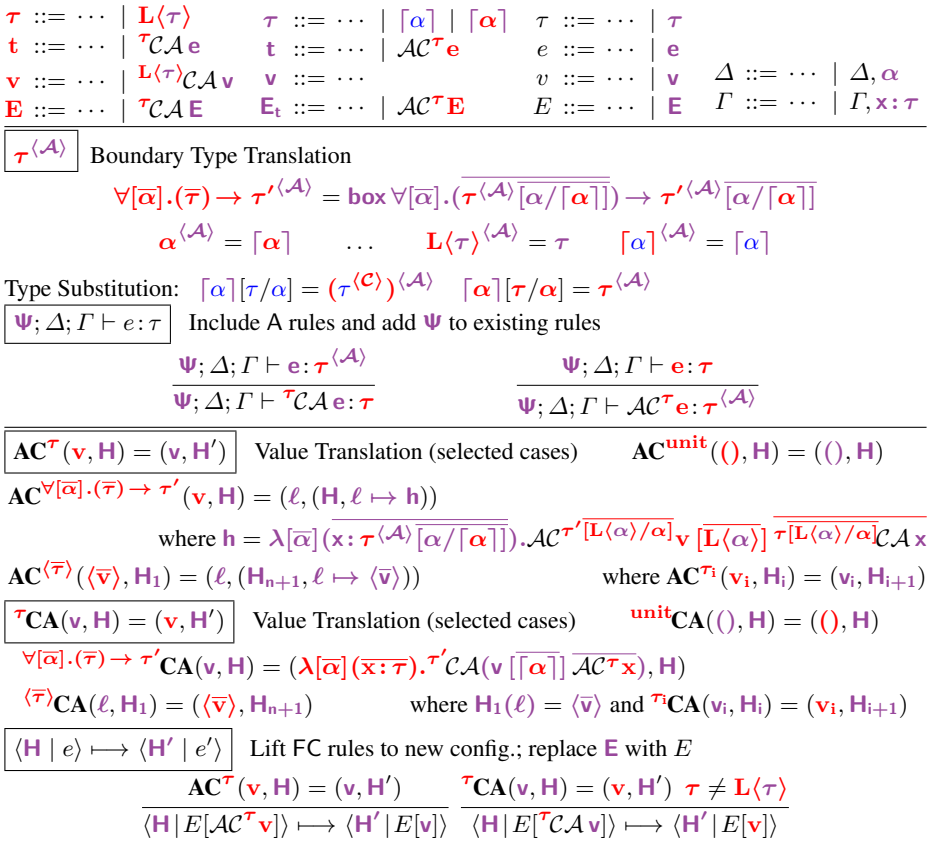

Fig. 4. FCA multi-language system (extends Figures 1 and 3)

6 C and A Interoperability

The extensions to FC for interoperability with A are given in Figure 4. The principles discussed in the development of FC still apply, but here we need to handle the presence of the heap. Specifically, since functions and tuples in A are contained in the heap, the value translations need access to the program's memory. Going from C to A, the value translation may allocate new memory for functions and tuples; going from A to C requires looking up the contents of locations and translating those contents to functions or tuples in C. Thus, we pass the current memory as an argument to the translations, and return a memory that may have had additional locations allocated. Memory cells allocated by boundaries are always immutable.

Aside from this change, the extension for the new language mostly follows what we did for FC: we augment the syntax with boundaries between C and A, a lump type $\mathbf{L}\langle\tau\rangle$ for opaquely embedding A values into C, and suspensions of type variables into A. Note that we need the boundary type translation from C to A to handle both C type variables $\boldsymbol{\alpha}$ and suspended F type variables $\lceil\alpha\rceil$. Thus A has both $\lceil\alpha\rceil$ and $\lceil\boldsymbol{\alpha}\rceil$ as suspension types. The boundary type translation $\boldsymbol{\tau}^{\langle\mathcal{A}\rangle}$ works similarly to $\tau^{\langle\mathcal{C}\rangle}$. The figure shows

$$
\begin{array}{rcl}
\mathsf{C} & ::= & [\cdot] \mid \mathsf{C}\ \mathsf{p}\ \mathsf{t} \mid \cdots \mid \lambda[\overline{\alpha}](\overline{\mathsf{x}:\tau}).\mathsf{C} \mid \cdots \mid {}^{\tau}\mathcal{FC}\ \mathbf{C} \\
\mathbf{C} & ::= & [\cdot] \mid \cdots \mid \boldsymbol{\lambda}[\overline{\alpha}](\overline{\mathbf{x}:\tau}).\mathbf{C} \mid \cdots \mid \mathcal{CF}^{\tau}\ \mathsf{C} \mid {}^{\tau}\mathcal{CA}\ \mathsf{C} \\
\mathsf{C} & ::= & (\mathsf{C_t}, \mathsf{H}) \mid (\mathsf{t}, \mathsf{C_H}) \\
\mathsf{C_t} & ::= & [\cdot] \mid \cdots \mid \mathcal{AC}^{\tau}\ \mathbf{C} \qquad \mathsf{C_H} ::= \mathsf{C_H}, \ell \mapsto \mathsf{h} \mid \mathsf{H}, \ell \mapsto \lambda[\overline{\alpha}](\overline{\mathsf{x}:\tau}).\mathsf{C_t} \\
C & ::= & \mathsf{C} \mid \mathbf{C} \mid \mathsf{C}
\end{array}
$$

$\boxed{C[e]}$ Context Plugging (A cases shown)

$$
(\mathsf{C_t}, \mathsf{H})[e] = \begin{cases} (\mathsf{C_t}[\mathsf{t}], (\mathsf{H}, \mathsf{H}')) & e = (\mathsf{t}, \mathsf{H}') \ \wedge\ \mathsf{C_t} \text{ contains no language boundaries} \\ (\mathsf{C_t}[e], \mathsf{H}) & \text{otherwise} \end{cases}
$$

$$
(\mathsf{t}, \mathsf{C_H})[e] = \begin{cases} (\mathsf{t}, (\mathsf{C_H}[\mathsf{t}'], \mathsf{H}')) & e = (\mathsf{t}', \mathsf{H}') \ \wedge\ \mathsf{C_H} \text{ contains no language boundaries} \\ (\mathsf{t}, \mathsf{C_H}[e]) & \text{otherwise} \end{cases}
$$

$$
[\cdot][\mathsf{t}] = \mathsf{t} \qquad (\mathsf{C_t}\ \mathsf{p}\ \mathsf{t})[e] = (\mathsf{C_t}[e])\ \mathsf{p}\ \mathsf{t} \qquad \cdots
$$

$$
(\mathsf{C_H}, \ell \mapsto \mathsf{h})[e] = (\mathsf{C_H}[e]), \ell \mapsto \mathsf{h}
$$

$$
(\mathsf{H}, \ell \mapsto \lambda[\overline{\alpha}](\overline{\mathsf{x}:\tau}).\mathsf{C_t})[e] = \mathsf{H}, \ell \mapsto \lambda[\overline{\alpha}](\overline{\mathsf{x}:\tau}).(\mathsf{C_t}[e])
$$

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

Contextual Equivalence

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

Fig. 5. General Contexts & Contextual Equivalence for FCA

the function case and the cases involving lumps and suspensions. The type judgment merges the A type rules with the FC type rules, but where the latter are modified to add the extra environment Ψ, and adds type rules for boundaries. Finally, the reduction relation for FCA lifts the FC reductions to use the configuration from A, with a program heap. We also add the reduction rules from A and a pair of boundary reduction rules that utilize the value translations.

7 Compiler Correctness

As mentioned in §1, we state compiler correctness in terms of FCA contextual equivalence. Below, we formally define contextual equivalence for FCA components and then present our compiler correctness theorems. We discuss how to prove these theorems in §9 and give a longer discussion and the full proofs in the technical report [9].

7.1 FCA Contextual Equivalence

A general context C is an FCA component with a hole. A component e can be plugged into the context only if it is from the same language as the hole. Since contexts can contain boundaries, e need not be from the same language as the outermost layer of C. The syntax of general contexts is given in Figure 5 (top). Contexts for F and C forms are standard. In A, we need contexts to be able to have their hole in either the term part of a component, or in the body of a function contained in the heap fragment. So in addition to contexts C that produce components, we have context forms $\mathsf{C_t}$ and $\mathsf{C_H}$ that produce terms and heap fragments, respectively.

When plugging an A component $(\mathbf{t}, \mathbf{H})$ into a context C, the heap fragment $\mathbf{H}$ is placed at the innermost component-level layer of C—that is, at the language boundary closest to the hole—and merged with the heap fragment already in that position. To formalize this, the A portion of the definition of plugging a component into a context is given in Figure 5 (middle). The definition of plugging for F and C contexts is standard.

Given this notion of general contexts, contextual equivalence for FCA is standard (see Figure 5, bottom). It says that two components e_1 and e_2 are contextually equivalent under environments $\mathbf{\Psi}$, Δ, Γ and at type τ if the following hold: First, both components must typecheck under $\mathbf{\Psi}$, Δ, Γ at type τ. Second, if C is a context that expects to be given a component that typechecks under $\mathbf{\Psi}$, Δ, Γ at type τ, and produces a resulting program that is closed but expects to be run with a heap of type $\mathbf{\Psi'}$, then $C[e_1]$ and $C[e_2]$ have the same termination behavior when we run them with any initial heap $\mathbf{H}$ that has type $\mathbf{\Psi'}$.

7.2 Compiler Correctness

We can now state our main result: compiler-correctness theorems for both passes of our compiler.

Theorem 1 (Closure Conversion is Semantics-Preserving). *If* $\overline{\alpha}; \overline{\mathsf{x}:\tau'} \vdash \mathsf{e}:\tau \rightsquigarrow \mathbf{e}$, *then* $\cdot; \overline{\alpha}; \overline{\mathsf{x}:\tau'} \vdash \mathsf{e} \approx^{ctx} {}^{\tau}\mathcal{FC}(\mathbf{e}[\overline{\lceil\alpha\rceil/\boldsymbol{\alpha}}]\ \overline{[\mathcal{CF}^{\tau'}\mathsf{x}/\mathsf{x}]}) : \tau$.

Theorem 2 (Allocation is Semantics-Preserving). *If* $\overline{\boldsymbol{\alpha}}; \overline{\mathbf{x}:\boldsymbol{\tau'}} \vdash \mathbf{e}:\boldsymbol{\tau} \rightsquigarrow (\mathbf{t}, \mathbf{H}:\mathbf{\Psi})$, *then* $\cdot; \overline{\boldsymbol{\alpha}}; \overline{\mathbf{x}:\boldsymbol{\tau'}} \vdash \mathbf{e} \approx^{ctx} {}^{\tau}\mathcal{CA}(\mathbf{t}[\overline{\lceil\boldsymbol{\alpha}\rceil/\alpha}]\ \overline{[\mathcal{AC}^{\boldsymbol{\tau'}}\mathbf{x}/\mathsf{x}]}, \mathbf{H}) : \boldsymbol{\tau}$.

The formal theorems are essentially as we described our compiler correctness results in §1, with only one additional subtlety: we need to perform a substitution so that the free variables of the original component match those of the compiled component. Recall that the compiler turns free type and term variables α and x into type and term variables $\boldsymbol{\alpha}$ and $\mathbf{x}$ from the next language, whereas FCA needs the binding structure of components to be preserved, including free variables being in the language prescribed by the type environments Δ and Γ. To get the free variables of the two components back into sync, we substitute suspended type variables for translated type variables, and we substitute boundary terms for translated term variables. Note that we do not need to perform a substitution in the heap fragment produced by the allocation pass, since heap values must be closed anyway.

We could equivalently have stated these theorems with the substitution on the other side, and the environments correspondingly translated; e.g.

$$\cdot; \overline{\alpha}^{\mathcal{C}}; \overline{\mathsf{x}:\tau'^{\mathcal{C}}} \vdash \mathsf{e}\overline{[\mathsf{L}\langle\boldsymbol{\alpha}\rangle/\alpha]}\ \overline{[\hat{\tau}'\mathcal{FC}\mathbf{x}/\mathsf{x}]} \approx^{ctx} {}^{\hat{\tau}}\mathcal{FC}\,\mathbf{e} : \hat{\tau},$$

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

It also does not matter which side the boundary term is placed on: boundary cancellation lemmas allow us to prove as a corollary that, for example,

$$\cdot; \overline{\alpha}; \overline{\mathsf{x}:\tau} \vdash \mathcal{F}^{\tau}\mathsf{e} \approx^{ctx} \mathbf{e}\overline{[\lceil\alpha\rceil/\boldsymbol{\alpha}]}\ \overline{[\mathcal{CF}^{\tau'}\mathsf{x}/\mathsf{x}]} : \tau^{\langle\mathcal{C}\rangle}.$$

Since we want to ensure that type variables in the environment remain tied to their free occurrences in the result type, this version of the theorem uses the boundary type translation $\tau^{\langle\mathcal{C}\rangle}$ for the result type (instead of the compiler's type translation $\tau^{\mathcal{C}}$).

Contextual equivalence is transitive, so we can easily chain these theorems together to prove correctness for the full compiler:

Corollary 1 (Compiler Correctness). *If* $\overline{\alpha}; \overline{\mathsf{x}:\tau'} \vdash \mathsf{e}:\tau \rightsquigarrow \mathbf{e} \rightsquigarrow \mathbf{e}$, *then*

$$\cdot; \overline{\alpha}; \overline{\mathsf{x}:\tau'} \vdash \mathsf{e} \approx^{ctx} {}^{\tau}\mathcal{FCA}(\mathsf{e}[\lceil\alpha\rceil/\alpha]\overline{[\mathcal{ACF}^{\tau'}\mathsf{x}/\mathsf{x}]}) : \tau.$$

8 An Example

We can use our compiler correctness theorem to make statements about linking with arbitrary A components, as long as they have translation type. In this section, we present an example showing how our framework allows linking both with A components that cannot be expressed in F, and with those that can. To keep our example concise, we use variable substitution as a simple notion of linking.

Consider the component

$$\mathsf{e} = (\lambda \mathsf{g}:\mathsf{unit}\to\mathsf{int}.\ (\mathsf{g}\ ()) * (\mathsf{g}\ ()))\ \mathsf{x},$$

where $\cdot;\cdot;(\mathsf{x}:\mathsf{unit}\to\mathsf{int}) \vdash \mathsf{e}:\mathsf{int}$. In F alone, only divergent or constant functions can have type $\mathsf{unit}\to\mathsf{int}$, but if we are compiling to A before linking, we could be given a component that makes use of A's mutable references.

Putting e through the first compiler pass, we get a C component that contains several administrative reductions. The complete result of compilation is shown in the technical report, but for readability, we pretend that e compiles to

$$\mathbf{e} = (\boldsymbol{\lambda}\mathbf{g}:\exists\alpha.\langle(\alpha,\mathbf{unit})\to\mathbf{int},\alpha\rangle.(\mathbf{unpack}\ \langle\beta,\mathbf{z}\rangle = \mathbf{g}\ \mathbf{in}\ (\boldsymbol{\pi}_1(\mathbf{z})\ \boldsymbol{\pi}_2(\mathbf{z})\ ()))$$
$$* (\mathbf{unpack}\ \langle\beta,\mathbf{z}\rangle = \mathbf{g}\ \mathbf{in}\ (\boldsymbol{\pi}_1(\mathbf{z})\ \boldsymbol{\pi}_2(\mathbf{z})\ ()))\ \mathbf{x},$$

which is equivalent to the actual result of compilation, and has exactly the same function body as the closure produced by the compiler.

The second pass brings us to an A component $\mathbf{e} = (\mathbf{t}, \mathbf{H})$, where $\mathbf{t} = \ell\ \mathbf{x}$ and

$$\mathbf{H} = \ell \mapsto \boldsymbol{\lambda}\mathbf{g}:\exists\alpha.\mathbf{box}\ \langle\mathbf{box}\ (\alpha,\mathbf{unit})\to\mathbf{int},\alpha\rangle.$$
$$((\mathbf{unpack}\ \langle\beta,\mathbf{z}\rangle = \mathbf{g}\ \mathbf{in}\ ((\mathbf{read}[1]\ \mathbf{z})\ (\mathbf{read}[2]\ \mathbf{z})\ ()))\ *$$
$$(\mathbf{unpack}\ \langle\beta,\mathbf{z}\rangle = \mathbf{g}\ \mathbf{in}\ ((\mathbf{read}[1]\ \mathbf{z})\ (\mathbf{read}[2]\ \mathbf{z})\ ()))).$$

By compiler correctness, we know that

$$\cdot;\cdot;(\mathsf{x}:\mathsf{unit}\to\mathsf{int}) \vdash \mathsf{e} \approx^{ctx} {}^{\mathsf{int}}\mathcal{FCA}(\mathsf{e}[\mathcal{ACF}^{\mathsf{unit}\to\mathsf{int}}\mathsf{x}/\mathsf{x}]) : \mathsf{int}.$$

Equivalently,

$$\cdot;\cdot;(\mathsf{x}:\tau) \vdash \mathcal{ACF}^{\mathsf{int}}(\mathsf{e}[{}^{\mathsf{unit}\to\mathsf{int}}\mathcal{FCA}\,\mathsf{x}/\mathsf{x}]) \approx^{ctx} \mathsf{e}:\mathbf{int},$$

where $\tau = \mathsf{unit}\to\mathsf{int}^{\langle\mathcal{C}\rangle\langle\mathcal{A}\rangle} = \exists\alpha.\mathbf{box}\ \langle\mathbf{box}\ (\alpha,\mathbf{unit})\to\mathbf{int},\alpha\rangle$.

Suppose we want to instantiate x with the following A component, which creates a function that uses a mutable reference to return the number of times it has been called:

$$\mathbf{e}' = (\mathbf{pack}\langle\mathbf{ref\ int},\mathbf{alloc}\ \langle\ell,\mathbf{ralloc}\ \langle\mathbf{0}\rangle\rangle\ \mathbf{as}\ \tau,$$
$$\ell \mapsto \boldsymbol{\lambda}(\mathbf{x}:\mathbf{ref\ int},\mathbf{z}:\mathbf{unit}).\ \mathbf{let}\ \mathbf{y} = \mathbf{read}[1]\ \mathbf{x}\ \mathbf{in}\ \mathbf{let}\ \mathbf{z} = \mathbf{write}\ \mathbf{x}\ [1] \leftarrow \mathbf{y}+\mathbf{1}\ \mathbf{in}\ \mathbf{y}+\mathbf{1}).$$

We would then have

$$\cdot;\cdot;\cdot \vdash \mathcal{ACF}^{\mathsf{int}}(\mathsf{e}[{}^{\mathsf{unit}\to\mathsf{int}}\mathcal{FCA}\,\mathsf{e}'/\mathsf{x}]) \approx^{ctx} \mathsf{e}[\mathsf{e}'/\mathsf{x}]:\mathbf{int},$$

The right-hand side of this equivalence is exactly the pure-A program that we would ultimately run, and the left-hand side is an FCA program that models it. Note that on either side of the equation, the function exported by $\mathbf{e}'$ will be applied to the unit value twice, returning 1 the first time and 2 the second time. An F function could not exhibit this behavior. This demonstrates how our framework allows for linking with components that are not expressible in F.

If we want instead to link with a different A component $\hat{e}$ that was compiled from an F component $\hat{e}$, we can still make the statement

$$\cdot;\cdot;\cdot \vdash \mathcal{ACF}^{\mathsf{int}}(e[{}^{\mathsf{unit}\to\mathsf{int}}\mathcal{FCA}\,\hat{e}/x]) \approx^{ctx} e[\hat{e}/x] : \mathbf{int},$$

but we can also simplify this statement using our additional knowledge of $\hat{e}$. Our compiler correctness theorem tells us that

$$\cdot;\cdot;\cdot \vdash \mathcal{ACF}^{\mathsf{unit}\to\mathsf{int}}\,\hat{e} \approx^{ctx} \hat{e} : \tau.$$

From this, we can infer that

$$\cdot;\cdot;\cdot \vdash \mathcal{ACF}^{\mathsf{int}}(e[{}^{\mathsf{unit}\to\mathsf{int}}\mathcal{FCA}(\mathcal{ACF}^{\mathsf{unit}\to\mathsf{int}}\,\hat{e})/x]) \approx^{ctx} e[\hat{e}/x] : \mathbf{int}.$$

Applying boundary cancellation yields

$$\cdot;\cdot;\cdot \vdash \mathcal{ACF}^{\mathsf{int}}(e[\hat{e}/x]) \approx^{ctx} e[\hat{e}/x] : \mathbf{int}.$$

Now we are essentially equating the pure-A program with a pure-F program, since the only multi-language element in this statement is the integer boundary at the outermost level, which merely converts an n to $\mathbf{n}$. This demonstrates that when we do have source-language equivalents for all our target-level components, our framework allows us to model target-level linking with source-level linking.

9 Proving Compiler Correctness

To prove the compiler correctness theorem, we design a step-indexed Kripke logical relation as a sound and complete model of contextual equivalence in FCA. Our logical relation extends that of Dreyer *et al.* [15] with the ability to handle multi-language type abstraction. We give an overview of the logical relation and a more detailed discussion of its novel features in the technical report [9]. In this section, we briefly discuss the high-level ideas behind our model's novel elements.

A logical-relations model provides a *relational value interpretation* of each type τ. This relation, which we denote $\mathcal{V}[\![\tau]\!]$, specifies when two values of type τ should be considered related or equivalent. When τ has free type variables, an environment ρ holds *arbitrary relational interpretations* for those abstract types. The relations in ρ capture the invariants of different instantiations of polymorphic values, which allows us to prove parametricity properties.

The interpretation $\mathcal{V}[\![\alpha]\!]\rho$ is defined by just looking up $\rho(\alpha)$. To prove important properties of $\mathcal{V}[\![\tau]\!]\rho$ for all types, we must ensure those properties hold in the α case by constraining the relations we can put into ρ to require these properties to hold up-front. Interpretations that satisfy these properties are called *candidates* or *admissible relations*.

In our multi-language setting, the two key properties we need to require for admissibility are boundary cancellation and the *bridge lemma*. The bridge lemma states that, given a pair of values v_1 and v_2 related according to the interpretation $\mathcal{V}[\![\tau]\!]\rho$, the $\mathbf{CF}^{\tau}$ translations of those values must be related according to $\mathcal{V}[\![\tau^{\langle C\rangle}]\!]\rho$. Similarly, given values $\mathbf{v_1}$ and $\mathbf{v_2}$ related according to $\mathcal{V}[\![\tau^{\langle C\rangle}]\!]\rho$, their ${}^{\tau}\mathbf{FC}$ translations must be related according to $\mathcal{V}[\![\tau]\!]\rho$. (We also require the analogous properties for the second pass.)

The type translation of α is $\lceil\alpha\rceil$, so in order for the bridge lemma to hold at type α, we need a suitable definition of $\mathcal{V}[\![\lceil\alpha\rceil]\!]\rho$, which necessarily will depend on $\rho(\alpha)$. One naïve definition we tried is the set of translations of values from $\rho(\alpha)$, roughly:

$$\mathcal{V}[\![\lceil\alpha\rceil]\!]\rho = \{(\mathbf{v_1}, \mathbf{v_2}) \mid (v_1, v_2) \in \rho(\alpha) \ \wedge\ \mathbf{CF}(v_i) = \mathbf{v_i}\}.$$

While this definition does let us prove the bridge lemma at type α, it does not satisfy boundary cancellation: if $\mathbf{v_1}$ and $\mathbf{v_2}$ are related according to this definition of $\mathcal{V}[\![\lceil\alpha\rceil]\!]\rho$, it is not necessarily the case that $\mathbf{CA}(\mathbf{AC}(\mathbf{v_1}))$ and $\mathbf{v_2}$ are related.

All the ways we tried to define $\mathcal{V}[\![\lceil\alpha\rceil]\!]\rho$ by a simple formula in terms of $\rho(\alpha)$ failed for similar reasons. Instead of giving a uniform definition, we took the viewpoint that if the properties of $\rho(\alpha)$ must be given *a priori*, then the particular relations with those properties that instantiate $\mathcal{V}[\![\alpha]\!]\rho$ and $\mathcal{V}[\![\lceil\alpha\rceil]\!]\rho$ should be given *a priori* as well. Specifically, in our model, an interpretation $\rho(\alpha)$ not just given by a relation on F values, but by a triple containing the relation on F values, a relation on C values to serve as its "translation" and instantiate $\mathcal{V}[\![\lceil\alpha\rceil]\!]\rho$, and a relation on A values to instantiate $\mathcal{V}[\![\lceil\alpha\rceil]\!]\rho$. Similarly, an interpretation $\rho(\boldsymbol{\alpha})$ is given by a pair containing a C-level relation and an A-level relation. For $\rho(\alpha)$, since A is the target language, only one relation is needed.

This strategy moves the burden for defining the "translations" of candidate relations to the places in our proof development where individual candidates are needed. But in all these places, there is some specific information available about the relation, so it was not difficult to construct them.

10 Discussion and Future Work

Software is composed from components written in different languages because different languages are suited to different tasks. We have provided a novel methodology for verifying *open*, *multi-pass* compilers, one that yields a stronger theorem than any existing work, allowing target-level linking with components of arbitrary provenance regardless of whether the component can be expressed in the source language compiled by the verified compiler.

Adding Compiler Passes Adding more intermediate languages to our compiler pipeline requires extending the multi-language model with new boundary forms and translation rules, and extending the logical relation with new clauses. Our aim is that the proof structure should be as modular as possible, so that the major lemmas and the correctness proof for one compiler pass can be completed independently of the rest of the pipeline. Presently, since our admissible relations design requires relations from multiple languages, we have a small number of places where a proof about one pass is affected by the other languages and passes. We hope to improve our proof engineering so that proofs for existing passes are unaffected when the compiler pipeline is changed.

Compiling to Assembly We have extended our compiler with a code-generation pass that translates A components to a stack-based typed assembly language, T. The latter is similar to Morrisett *et al.*'s stack-based TAL [6] but with a type system that tracks more information. Informally, the T type system allows us to track calls and returns of semantic "functions" that may span multiple basic blocks, and to determine the "return type" of such functions. With this information, we are able to give a formal definition of contextual equivalence for T that makes distinctions about assembly at an appropriate level of granularity. That is, we relate assembly language components comprised of any number of basic blocks, rather than relating individual basic blocks. An equivalence relation based on individual blocks would be too fine grained; for instance, it would be unable to relate two components with an unequal number of basic blocks that may

have been produced by compiling two equivalent source terms. We are working on the proofs for this pass and will report on it in a future paper.

Mutable References Consider adding mutable references to F and C. For the first compiler pass, we would extend the type translation with $(\mathsf{ref}\ \tau)^{\mathsf{C}} = \mathbf{ref}\ \tau^{\mathsf{C}}$. When defining interoperability at type $\mathsf{ref}\ \tau$, it doesn't make sense to convert an F location ℓ into a fresh C location $\boldsymbol{\ell}$ (and vice versa) since it would lead to duplication of mutable cells in the interoperating languages and these would be impossible to keep in sync. One solution is to treat a wrapped location (e.g., ${}^{\mathsf{ref}\ \tau}\mathcal{FC}\boldsymbol{\ell}$) as a value form. Operations on these wrapped locations can be performed by reduction rules such as these:

$$!({}^{\mathsf{ref}\ \tau}\mathcal{FC}\boldsymbol{\ell}) \longmapsto {}^{\tau}\mathcal{FC}(!\boldsymbol{\ell}) \quad ({}^{\mathsf{ref}\ \tau}\mathcal{FC}\boldsymbol{\ell}) := \mathsf{v} \longmapsto {}^{\mathsf{unit}}\mathcal{FC}(\boldsymbol{\ell} := \mathcal{CF}^{\tau}\mathsf{v}),$$

where $!\mathsf{v}$ is a dereference and $\mathsf{v} := \mathsf{v}'$ is an assignment. Passing references between C and A can be done analogously. While these interoperability semantics are straightforward, we expect to find nontrivial challenges in designing a logical relation to properly handle the wrapped-location value forms they introduce.

Supporting Realistic Interoperability We are particularly interested in supporting target-level interoperability between a language with parametric polymorphism such as ML and languages without type abstraction such as Scheme or C. For instance, given a generic tree library compiled from ML, we want to allow code compiled from Scheme or C to be able to use the library but ensure that such use cannot invalidate ML's parametricity guarantees by inspecting values that have abstract type on the ML side. In this paper, we have shown how to preserve ML's parametricity guarantees part-way through the compiler. Going forward we wish to develop a gradually typed assembly language that, following Matthews and Ahmed [16], uses dynamic sealing on the untyped side to enforce parametricity guarantees provided by type abstraction on the typed side.

11 Related Work

The literature on compiler verification spans over four decades but is mostly limited to whole-program compilation; we refer the reader to the bibliography by Dave [17] for compilers for first-order languages, and to Chlipala [18] for compilers for higher-order functional languages. We have already discussed the existing work [10, 12] on compositional compiler correctness in §2. Here we focus on other closely related work.

Dreyer et al. have recently been working on Relational Transition Systems (RTS's) [19] that may provide an alternative cross-language specification technique that is designed to make it possible to prove transitivity. Regardless, it is still not easy to do: see their technical report [20] where they prove transitivity for their *single-language* RTS system for an idealized ML. It is a non-trivial task to do this for multiple cross-language RTS's. Additionally, even if the RTS approach proves effective for verifying a multi-pass compiler, it still does not address the problem of linking with a component e'_T for which there is no related source-level e'_S.

The design of our multi-language system builds on that of Ahmed and Blume [8], who developed a boundary-based multi-language system embedding the source (STLC) and target (System F) of CPS translation. Ahmed and Blume did not have type abstraction in the source language, which meant that they did not have to make use of lumps or suspensions, nor design a logical relation to handle these. Our semantics preservation proof is analogous to theirs. However, since they were interested in fully abstract

CPS translation, they designed their type translation to disallow linking compiled code with target components whose behavior cannot be expressed at the source level. The additional work that they do to prove full abstraction provides a roadmap for how to extend our methodology to prove full abstraction in a setting where the type translation enforces it.

Acknowledgements We would like to thank Nick Benton, whose views on compositional compiler correctness have been an inspiration to us. In particular, our thinking has been influenced by Benton and Hur's introduction [11], which eloquently lays out desirable features of a compiler correctness specification. We would also like to thank Aaron Turon for helpful feedback on an earlier version of this paper. This research was supported by the National Science Foundation (grant CCF-1203008).

References

1. Leroy, X.: Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In POPL 2006.
2. Leroy, X.: A formally verified compiler back-end. J. Automated Reasoning **43**(4), 363–446, 2009.
3. Lochbihler, A.: Verifying a compiler for Java threads. In ESOP 2010.
4. Myreen, M.O.: Verified just-in-time compiler on x86. In POPL 2010
5. Sevcik, J., Vafeiadis, V., Nardelli, F.Z., Jagannathan, S., Sewell, P.: Relaxed-memory concurrency and verified compilation. In POPL 2011.
6. Morrisett, G., Crary, K., Glew, N., Walker, D.: Stack-based typed assembly language. J. Functional Programming **12**(1), 43–88, 2002.
7. Matthews, J., Findler, R.B.: Operational semantics for multi-language programs. In POPL 2007.
8. Ahmed, A., Blume, M.: An equivalence-preserving CPS translation via multi-language semantics. In ICFP 2011.
9. Perconti, J.T., Ahmed, A.: Verifying an open compiler using multi-language semantics (technical report). Available at `http://ccs.neu.edu/home/amal/voc/`, Jan. 2014.
10. Benton, N., Hur, C.K.: Biorthogonality, step-indexing and compiler correctness. In ICFP 2009.
11. Benton, N., Hur, C.K.: Realizability and compositional compiler correctness for a polymorphic language. Technical Report MSR-TR-2010-62, Microsoft Research, Apr. 2010.
12. Hur, C.K., Dreyer, D.: A Kripke logical relation between ML and assembly. In POPL 2011.
13. Morrisett, G., Walker, D., Crary, K., Glew, N.: From System F to typed assembly language. ACM TOPLAS **21**(3), 527–568, May 1999.
14. Minamide, Y., Morrisett, G., Harper, R.: Typed closure conversion. In POPL 1996.
15. Dreyer, D., Neis, G., Birkedal, L.: The impact of higher-order state and control effects on local relational reasoning. J. Functional Programming **22**(4&5) (2012) 477–528
16. Matthews, J., Ahmed, A.: Parametric polymorphism through run-time sealing, or, theorems for low, low prices! In ESOP 2008.
17. Dave, M.A.: Compiler verification: A bibliography. ACM SIGSOFT Software Engineering Notes **28**(6), 2003.
18. Chlipala, A.: A verified compiler for an impure functional language. In POPL 2010.
19. Hur, C.K., Dreyer, D., Neis, G., Vafeiadis, V.: The marriage of bisimulations and Kripke logical relations. In POPL 2012.
20. Hur, C.K., Dreyer, D., Neis, G., Vafeiadis, V.: The marriage of bisimulations and Kripke logical relations. Technical report, Max Planck Institute for Software Systems, Jan. 2012.