An Equivalence-Preserving CPS Translation via Multi-Language Semantics *

Amal Ahmed
Indiana University
amal@cs.indiana.edu

Matthias Blume
Google
blume@google.com

Abstract

Language-based security relies on the assumption that all potential attacks follow the rules of the language in question. When programs are compiled into a different language, this is true only if the translation process preserves observational equivalence.

To prove that a translation preserves equivalence, one must show that if two program fragments cannot be distinguished by any source context, then their translations cannot be distinguished by any target context. Informally, target contexts must be no more powerful than source contexts, i.e., for every target context there exists a source context that "behaves the same." This seems to amount to being able to "back-translate" arbitrary target terms. However, that is simply not viable for practical compilers where the target language is lower-level and, thus, contains expressions that have no source equivalent.

In this paper, we give a CPS translation from a less expressive source language (STLC) to a more expressive target language (System F) and prove that the translation preserves observational equivalence. The key to our equivalence-preserving compilation is the choice of the right type translation: a source type σ mandates a set of behaviors and we must ensure that its translation σ^+ mandates semantically equivalent behaviors at the target level. Based on this type translation, we demonstrate how to prove that for every target term of type σ^+, there exists an equivalent source term of type σ — even when sub-terms of the target term are not necessarily "back-translatable" themselves. A key novelty of our proof, resulting in a pleasant proof structure, is that it leverages a multi-language semantics where source and target terms may interoperate.

Categories and Subject Descriptors D.3.1 [*Programming Languages*]: Formal Definitions and Theory—Semantics

General Terms Languages, Reliability, Security, Theory

Keywords full abstraction, equivalence-preserving compilation, continuation-passing style, multi-language semantics, logical relations, back-translation

* In electronic versions of this paper, we use blue to typeset our source language and **red** to typeset the target. The paper will be much easier to read if viewed/printed in color.

ICFP'11, September 19–21, 2011, Tokyo, Japan.

1. Introduction

Abstraction is a key tool for ensuring the reliability and security of large, complex systems. We modularize such systems into components, define *interfaces* between these components, and let each component's *implementation* depend only on the interfaces of other components, not their implementation. With the advent of languages like Java and C#, developers increasingly rely on programming language techniques for enforcing abstraction.

Language-based security relies on an *abstraction theorem* [40] which effectively states that no user of a component can observe the difference between two different implementations of that component (i.e., the two implementations are *contextually equivalent*) if all manifestations of that difference in the interface are masked by an *abstract type*. If we think of the context as the adversary, the potential attacker, it becomes clear that language-based security requires that the attacker obey the typing rules of the same language.

Most programs written in some language S (*source*) are compiled to another language T (*target*) from where they are then executed. Thus, components e_S and e'_S might be compiled to e_T and e'_T, which would then interact with a target context C_T. But what if there exists some C_T that can observe a difference between e_T and e'_T, even if e_S and e'_S are contextually equivalent? This is a question that programmers *should* care about! It is critical for a programmer writing code in language S be able to reason about the properties of her code by *thinking* in S—that is, by only considering the behavior of other S components that may interact with her code in a type-safe manner. (In particular, to reason about the properties of an S component, she *should not* have to consider all possible interactions with components written in a different language, such as the target language T.) This can only be achieved if compilation both *preserves* and *reflects* contextual equivalence and is, therefore, *fully abstract*.

If the set of possible contexts in T is restricted to exactly those that can be obtained by translating S contexts, then it would be easy to show that no C_T can distinguish between code fragments not distinguishable by S contexts. However, this is usually not the case. For instance, Microsoft's Common Language Runtime (CLR) was specifically designed to be the target of compilers for multiple source languages, and most traditional compilers generate machine code that can then be linked with other machine code, possibly obtained by compiling code written in other source languages. In these situations, it is possible that the target language contains features that have no source-level equivalent, leading to T contexts that are *too powerful* in the sense that they can make observations that S contexts cannot. In fact, Kennedy [26] describes a number of ways in which abstractions were broken in the process of compiling C# to the CLR intermediate language. Similar problems with Java have been previously examined by Abadi [1].

There are three approaches to repairing failures of full abstraction. First, we could add features to the source language so that every target-level observation has a source-level counterpart. But this is hardly a desirable solution as it amounts to weakening the abstraction facilities of the source language. Second, we could remove features from the target language until it becomes, in some sense, merely an alternative notation for source programs, thereby guaranteeing that the only expressible target contexts trivially correspond to source contexts. This might work in some specialized situations, but it does not apply to foreign-function interfaces, plugin architectures, or multi-language frameworks such as .NET. (As we discuss in Section 9, much of the existing work on proving full abstraction resorts to one of these two approaches.)

The third approach is to change the translation. Specifically, we advocate engineering the translation so that it uses target-level abstraction facilities in a clever enough fashion so that well-typed target contexts have no choice but to respect the original abstractions. Put another way, instead of removing features from the target language, engineer the type translation to use types at the target level to restrict the set of contexts that a compiled source component can interact (or be linked) with. Of course, this assumes the presence of a rich enough type system at the target language. Fortunately, JVM bytecode, the CLR, and Typed Assembly Language (TAL) [36] may already provide most of the necessary features.

Assuming that the translation can be engineered in this way, it is still not clear how to *prove* that the result really is fully abstract. Full abstraction for a full-fledged C#-to-CLR is currently too hard a problem to tackle. A nontrivial first step would be to attempt full abstraction for a more idealized compiler such as that from System F to TAL [36] (which was mentioned in that paper as future work).

In prior work [4], we proved that typed closure conversion as defined by Morrisett *et al.* [36] is fully abstract. For typed closure conversion the key target language feature required was existential types, while the particular way the translation assigns existential types to closure records is what makes full abstraction work.

An interesting aspect of our earlier proof—which, unfortunately, significantly limits the situations where that proof strategy may be used—is that it takes advantage of a setup where source language S and target language T are the *same* language. Of course, source and target languages are rarely the same in practice, but we argued there that as long as the "real" target language is *less* expressive than the source language (i.e., if the compiler "compiles away" certain high-level source features), there is no loss of generality. Simply speaking, if contexts in the more expressive target language T (where $T = S$) cannot distinguish between two target language expressions, then contexts in the less expressive "real" target language cannot make the distinction either. Thus, our earlier proof methodology suffices when the source language is at least as expressive as the target, but *not* when the latter is more expressive.

However, there are usually some features in typical target languages that have no source equivalent. A common example for this is *control*: low-level languages tend to have explicit representations of the program counter and the program's control stack. In this setting the assumption of having equally powerful source and target languages does not work. To prove full abstraction when it holds, a different proof technique is required.

In this paper we investigate the full abstraction problem for CPS translation from a *less expressive source language* to a *more expressive target language*. Since CPS conversion makes continuations explicit, it represents the above-mentioned situation of a target language with explicit control. Next, we show that the "standard" typed CPS translation is not fully abstract, after which we discuss the specific contributions of this paper.

"Standard" CPS Conversion is Not Fully Abstract As we have discussed, full abstraction is at least as much a property of the translation as it is one of the target language. Therefore, a particular translation scheme can fail to be fully abstract even if the source- and target languages are identical. Consider the simply-typed λ-calculus. In this setting, it is easy to see that the following two terms A and B are contextually equivalent:

$$\begin{aligned} A &= \lambda(f : \mathrm{int} \to \mathrm{int}, g : \mathrm{int} \to \mathrm{int}).(f\,0;\ g\,0;\ 0) \\ B &= \lambda(f : \mathrm{int} \to \mathrm{int}, g : \mathrm{int} \to \mathrm{int}).(g\,0;\ f\,0;\ 0). \end{aligned}$$

When applied to concrete arguments, the results of calling f and g are ignored in either case, and the overall answer is always 0.

Now consider A' and B', which are the results of translating A and B using the standard typed CPS-conversion approach (see, e.g., [20]) which makes use of a global abstract answer type ans. (We omit the type annotation $\mathrm{int} \times (\mathrm{int} \to \mathrm{ans}) \to \mathrm{ans}$ on f, g):

$$\begin{aligned} A' &= \lambda(f, g, k : \mathrm{int} \to \mathrm{ans}).f(0, \lambda\_.\, g(0, \lambda\_.\, k\,0)) \\ B' &= \lambda(f, g, k : \mathrm{int} \to \mathrm{ans}).g(0, \lambda\_.\, f(0, \lambda\_.\, k\,0)) \end{aligned}$$

The following context C distinguishes between A' and B':

$$C = \lambda k.\, [\cdot]\,(\lambda(\_, \_).\, k\,1,\ \lambda(\_, \_).\, k\,2,\ k)$$

Substituting A' for the hole applies k to 1, while plugging in B' applies k to 2. The problem is that the values given for f and g take advantage of the explicit representation of continuations. They do something that our non-CPS functions cannot do: they ignore their own continuation and directly invoke a different one, thereby exposing the previously invisible difference in evaluation order within the bodies of A and B. Thus, this particular CPS-translation is not fully abstract.

The difference could be observed by source contexts if we added more powerful control operators to our source language, such as `call/cc` or even just exceptions. But we want a fully abstract translation without enriching the source language and weakening its abstractions to make more observations possible. The key idea is to take advantage of the target language type system in such a way that it rules out any target contexts that could make observations that are not possible at the source. One option, that has been studied extensively (e.g., Berdine *et al.* [10, 11]), is to ascribe linear types to continuations thus ensuring that they are used exactly once. As we will explain, in this paper, we eschew adding linear (or affine) types to the target and instead use polymorphism.

Contributions Our source language λ^{S} is the simply typed λ-calculus, while our target calculus λ^{T} is System F. (Section 3). We prove full abstraction for a CPS translation where each computation term is *locally* polymorphic in its answer type (Section 4). This translation has been studied before [48], but we are unaware of any work on proving it fully abstract. Although our target type system itself is not substructural, the locally polymorphic answer type is sufficient to enforce that continuations be used *at least once*—that is, continuations must be *relevant* in the terminology of substructural logics. Intuitively, enforcing relevance of continuations suffices in our setting because in a purely functional, terminating language such as λ^{T} it is impossible to distinguish between a single use and multiple uses of a continuation.

Our proof of full abstraction uses elementary operational techniques; there is no use of sophisticated machinery such as domain theory or game semantics. While the proof technique we used in prior work [4] relied on the source and target language being identical, our current proof technique works even if the two languages are different. To illustrate this point, we have picked a target (System F) that is more expressive than the source (STLC)—e.g., we can encode arithmetic operations in System F but not in STLC. In addition, like Morrisett *et al.*'s CPS target language [36], our λ^{T} requires terms to be in CPS form (in the spirit of compilers like SML/NJ [7, 45], Rabbit [47], Orbit [28], and more recently, Kennedy's SML.NET compiler [27], though our syntax is not as restrictive, e.g., we do not distinguish functions from continuations.)

Thus, neither language is a sub-language of the other. An interesting aspect of our proof technique is that it draws upon work on *language interoperability*. We define a *combined* language λ^{ST} that incorporates both λ^{S}, λ^{T}, and has two new *boundary* forms that let us interface terms of one language with the other (Section 5). This setup allows us to neatly decompose our proof into three parts (Section 6).

We discuss the addition of recursion to both the source and target in Section 8 but the details are beyond the scope of this paper. We have elided most proofs here. Detailed proofs may be found in the online technical appendix [5].

2. Main Ideas

Multi-language scenario and interoperability. Consider two source terms e_S and e'_S and their corresponding translation terms e_T and e'_T. To show full abstraction we need to establish equivalence *reflection* and equivalence *preservation*. Reflection is closely related to "compiler correctness" in the sense that we deem the translation fundamentally broken if non-equivalent source terms translate to equivalent target terms. The hard part of the proof, however, is to establish preservation: terms that are equivalent at source level should translate to equivalent target terms.

A natural way of proving equivalence preservation is to take an indirect approach: assume that e_T and e'_T are *not* equivalent and derive a contradiction. If e_T and e'_T are not contextually equivalent, there has to be a target context C_T that exposes the difference: $C_T[e_T]$ evaluates to true_T while $C_T[e'_T]$ evaluates to false_T. The idea is to show the existence of a *source* context C_S such that $C_S[e_S]$ evaluates to true_S while $C_S[e'_S]$ evaluates to false_S. In our previous work [4] we were able to construct C_S directly from C_T with the help of "wrapper" terms. This technique does not apply directly to the current setting of mutually incompatible source- and target languages.

However, it is possible to apply the technique—at least in an intuitive sense—if we take a page out of the work on interoperability [31] and define a combined language in which S- and T-terms can interoperate in a controlled way. The discriminating target context C_T gives rise to a discriminating context C_{ST} for e_S and e'_S. What remains to be done now is to show that C_{ST} can be converted to a context C_S that is equally discriminating.

Relevant continuations via parametricity. Our version of typed CPS-conversion differs from the "standard" account that uses a single, globally abstract answer type. Our answer type is individually abstract at each point where a continuation argument appears. In essence, each computation term of type $(\tau \to \mathrm{ans}) \to \mathrm{ans}$ becomes $\forall\alpha\,.\,(\tau \to \alpha) \to \alpha$. The polymorphic type variable α replaces the single answer type ans. As a result, the computation has less freedom in how it can use its continuation. In particular, to produce its own answer of type α, it has no choice but to invoke *its own* continuation (which ensures that the continuation is used at least once). It turns out that this typing of CPS code prevents any "bad" target terms (for example those whose source equivalent is a variant of `call/cc`) from being well typed. The technical underpinnings of this intuition is a *free theorem* [50] that applies to computation terms that are polymorphic in their answer type.

Back-translation. The remaining hurdle is to show that CPS-typed target contexts can be "back-translated" to corresponding source contexts. The difficulty lies in the fact that the type only governs the interface of a term without preventing the presence of arbitrary subterms, so for a term to have translation type does not immediately imply that it can be back-translated. However, as we will show, we are always able to eliminate occurrences of inconvenient subterms by partially reducing them. For instance, whenever

Types	σ	::=	$\mathsf{bool} \mid \sigma_1 \to \sigma_2$
Values	v	::=	$\mathsf{x} \mid \mathsf{true} \mid \mathsf{false} \mid \lambda \mathsf{x} : \sigma.\, \mathsf{e}$
Terms	e	::=	$\mathsf{v} \mid \mathsf{if}\ \mathsf{e}\ \mathsf{then}\ \mathsf{e}_1\ \mathsf{else}\ \mathsf{e}_2 \mid \mathsf{e}_1\ \mathsf{e}_2$
Eval. Contexts	E	::=	$[\cdot]_{\mathsf{S}} \mid \mathsf{if}\ \mathsf{E}\ \mathsf{then}\ \mathsf{e}_1\ \mathsf{else}\ \mathsf{e}_2 \mid \mathsf{E}\ \mathsf{e} \mid \mathsf{v}\ \mathsf{E}$

$\boxed{\mathsf{e} \longmapsto \mathsf{e}'}$

$$\mathsf{if\ true\ then\ e_1\ else\ e_2} \longmapsto \mathsf{e_1}$$
$$\mathsf{if\ false\ then\ e_1\ else\ e_2} \longmapsto \mathsf{e_2}$$
$$(\lambda \mathsf{x} : \sigma.\, \mathsf{e})\ \mathsf{v} \longmapsto \mathsf{e}[\mathsf{v}/\mathsf{x}]$$
$$\frac{\mathsf{e} \longmapsto \mathsf{e}'}{\mathsf{E}[\mathsf{e}] \longmapsto \mathsf{E}[\mathsf{e}']}$$

Figure 1. λ^{S}: Syntax and Dynamic Semantics

we have a term of the form $\mathbf{v}\ [\tau]\ \mathbf{v}_1$—application of a polymorphic function $\mathbf{v}$ to type τ and then to an argument $\mathbf{v}_1$—of translation type σ^+, either $\mathbf{v}$ will have a translation type $(\sigma_1 \to \sigma_2)^+$ (and therefore can be related to a source term of type $\sigma_1 \to \sigma_2$), or it will be a lambda term that can be applied to τ and $\mathbf{v}_1$, yielding a term $\mathbf{e}$ of type σ^+. The latter term can now be back-translated. This is well-founded because the result term is "smaller" in that it will reduce to a value in fewer steps. The metric for well-foundedness of our "back-translation" relation is a combination of length of reduction sequence, structure of the type of the term being back-translated, and the structure of the expression. We explain the details in Section 6.2.

3. The Source and Target Languages

We typeset the terms, types, and contexts of our source language λ^{S} using a blue sans-serif font, and those of our target language λ^{T} using a **bold red font with serifs**.

3.1 The Source Language (λ^{S})

Our source language λ^{S} is the call-by-value simply-typed λ-calculus with booleans. The syntax and dynamic semantics of λ^{S} are shown in Figure 1. We define a small-step operational semantics for λ^{S}, using evaluation contexts E to lift the primitive reductions to a standard left-to-right call-by-value semantics for the language.

λ^{S} typing judgments have the form $\Gamma \vdash \mathsf{e} : \sigma$, where the value environment Γ tracks the set of free term variables in scope, along with their types. The typing rules are entirely standard so we omit them here. (They appear later as part of Figure 3 when we define the CPS translation by induction on the structure of $\Gamma \vdash \mathsf{e} : \sigma$.)

λ^{S} Contextual Equivalence A λ^{S} context C is an expression with a single hole $[\cdot]_{\mathsf{S}}$ in it. Typing judgments for contexts have the form $\vdash \mathsf{C} : (\Gamma \vdash \sigma) \Rightarrow (\Gamma' \vdash \sigma')$, where $(\Gamma \vdash \sigma)$ indicates the type of the hole. Essentially, this judgment says that if e is an expression such that $\Gamma \vdash \mathsf{e} : \sigma$, then $\Gamma' \vdash \mathsf{C}[\mathsf{e}] : \sigma'$. The typing rule for a hole is as follows:

$$\frac{\Gamma \subseteq \Gamma'}{\vdash [\cdot]_{\mathsf{S}} : (\Gamma \vdash \sigma) \Rightarrow (\Gamma' \vdash \sigma)}$$

The other rules are straightforward (see our online appendix [5]).

We define contextual approximation $(\Gamma \vdash \mathsf{e}_1 \precsim^{ctx}_{\mathrm{S}} \mathsf{e}_2 : \sigma)$ to mean that, for any well-typed program context C with a hole of the type of e_1 and e_2, and result type bool, if $\mathsf{C}[\mathsf{e}_1]$ evaluates to the boolean value v then so does $\mathsf{C}[\mathsf{e}_2]$. Contextual equivalence $(\Gamma \vdash \mathsf{e}_1 \approx^{ctx}_{\mathrm{S}} \mathsf{e}_2 : \sigma)$ is then defined as contextual approximation in both directions.

Definition 3.1 (λ^{S} **Contextual Approximation & Equivalence**)
Let $\Gamma \vdash e_1 : \sigma$ *and* $\Gamma \vdash e_2 : \sigma$.

$$\Gamma \vdash e_1 \precsim^{ctx}_{\mathrm{S}} e_2 : \sigma \stackrel{\text{def}}{=} \forall C, v_1.\ \vdash C : (\Gamma \vdash \sigma) \Rightarrow (\cdot \vdash \mathsf{bool}) \ \wedge\ C[e_1] \Downarrow v_1 \implies \exists v_2.\ C[e_2] \Downarrow v_2 \ \wedge\ v_1 = v_2$$

$$\Gamma \vdash e_1 \approx^{ctx}_{\mathrm{S}} e_2 : \sigma \stackrel{\text{def}}{=} \Gamma \vdash e_1 \precsim^{ctx}_{\mathrm{S}} e_2 : \sigma \ \wedge\ \Gamma \vdash e_2 \precsim^{ctx}_{\mathrm{S}} e_1 : \sigma$$

λ^{S} ***CIU equivalence*** Note that evaluation contexts E (Figure 1) are a subset of general contexts C. Since only closed terms can be placed in an evaluation context, the type of the hole always has the form $\cdot \vdash \sigma$.

We define the notion of *ciu-equivalence* (uses of closed instantiations, first introduced by Mason and Talcott [30]) and show that it is a consequence of contextual equivalence. Two closed terms of type σ are ciu-equivalent if, in any evaluation context E with hole type σ and result type bool, they evaluate to the same value. This notion is often easier to work with than contextual equivalence since it cuts down on the number of contexts under consideration. The notion is extended to open terms by closing the terms with a value substitution γ that maps variables x to values v; we write $\vdash \gamma : \Gamma$ when $\mathrm{dom}(\gamma) = \mathrm{dom}(\Gamma)$ and for all $x \in \mathrm{dom}(\Gamma)$, $\vdash \gamma(x) : \Gamma(x)$.

Definition 3.2 (λ^{S} **CIU Approximation & Equivalence**)
Let $\Gamma \vdash e_1 : \sigma$ *and* $\Gamma \vdash e_2 : \sigma$.

$$\Gamma \vdash e_1 \precsim^{ciu}_{\mathrm{S}} e_2 : \sigma \stackrel{\text{def}}{=} \forall \mathsf{E}, \gamma, v_1.\ \vdash \mathsf{E} : (\cdot \vdash \sigma) \Rightarrow (\cdot \vdash \mathsf{bool}) \ \wedge\ \vdash \gamma : \Gamma \ \wedge\ \mathsf{E}[\gamma(e_1)] \Downarrow v_1 \implies \exists v_2.\ \mathsf{E}[\gamma(e_2)] \Downarrow v_2 \ \wedge\ v_1 = v_2$$

$$\Gamma \vdash e_1 \approx^{ciu}_{\mathrm{S}} e_2 : \sigma \stackrel{\text{def}}{=} \Gamma \vdash e_1 \precsim^{ciu}_{\mathrm{S}} e_2 : \sigma \ \wedge\ \Gamma \vdash e_2 \precsim^{ciu}_{\mathrm{S}} e_1 : \sigma$$

Lemma 3.3 (λ^{S}**: Contextual Approx Implies CIU Approx**)
If $\Gamma \vdash e_1 \precsim^{ctx}_{\mathrm{S}} e_2 : \sigma$ *then* $\Gamma \vdash e_1 \precsim^{ciu}_{\mathrm{S}} e_2 : \sigma$.

3.2 The Target Language (λ^{T})

Our CPS target language λ^{T} is call-by-value System F with booleans and pairs. The λ^{T} syntax and dynamic semantics are given in Figure 2 (top). Following Morrisett *et al.* [36], our CPS target language syntactically enforces continuation-passing style. λ^{T} code is nearly linear—consisting of a series of let bindings followed by a function call—with the exception of the $\mathbf{if}$ construct which forms a tree containing two subexpressions. Also following Morrisett *et al.*, we combine the types $\forall$ and $\rightarrow$ into $\forall\,[\alpha].\tau_1 \rightarrow \tau_2$ and have only one abstraction mechanism (λ) which binds both type and term variables. Finally, we have pairs in λ^{T} so we can express CPS conversion without the need to introduce more curried functions.

We define a small-step, call-by-value operational semantics. Notice that evaluation contexts in λ^{T} are redundant since the syntactic restriction to CPS results in a unique order of evaluation; we introduce them primarily to permit a more uniform treatment of λ^{S} and λ^{T} when they are incorporated into the multi-language semantics in Section 5.

λ^{T} typing judgments have the form $\Delta; \Gamma \vdash \mathbf{e} : \tau$, where the environments Δ and Γ are defined in Figure 2. 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 in environment Δ (written $\Delta \vdash \tau$ and defined as $\mathrm{ftv}(\tau) \subseteq \Delta$, where $\mathrm{ftv}(\tau)$ denotes the set of type variables that appear free in type τ). The typing rules are standard, so we only show a few rules in Figure 2 (middle).

Syntactic Sugar We abbreviate $\forall\,[\alpha].\tau_1 \rightarrow \tau_2$ as $\tau_1 \rightarrow \tau_2$ when $\alpha \notin (\mathrm{ftv}(\tau_1) \cup \mathrm{ftv}(\tau_2))$; we similarly abbreviate $\lambda\,[\alpha]\,(x:\tau_1).\,\mathbf{e}$ to $\lambda(x:\tau_1).\,\mathbf{e}$, when α does not appear free in τ_1 or $\mathbf{e}$; and we abbreviate $\mathbf{v}\,[\tau]\,\mathbf{v}_1$ to $\mathbf{v}\,\mathbf{v}_1$ when $\mathbf{v}$ has type $\tau_1 \rightarrow \tau_2$. We also use

Types $\tau ::= \mathbf{bool} \mid \tau_1 \times \tau_2 \mid \alpha \mid \forall\,[\alpha].\tau_1 \rightarrow \tau_2$
Values $\mathbf{v} ::= x \mid \mathbf{true} \mid \mathbf{false} \mid (\mathbf{v}_1, \mathbf{v}_2) \mid \lambda\,[\alpha]\,(x:\tau).\,\mathbf{e}$
Terms $\mathbf{e} ::= \mathbf{v} \mid \mathbf{if}\ \mathbf{v}\ \mathbf{then}\ \mathbf{e}_1\ \mathbf{else}\ \mathbf{e}_2 \mid \mathbf{let}\ x = \pi_i \mathbf{v}\ \mathbf{in}\ \mathbf{e} \mid \mathbf{v}_1\,[\tau]\,\mathbf{v}_2$
Eval. Ctxts $\mathbf{E} ::= [\cdot]_{\mathbf{T}}$

$\boxed{\mathbf{e} \longmapsto \mathbf{e}'}$

$$\begin{array}{rcl} \mathbf{if\ true\ then\ e_1\ else\ e_2} & \longmapsto & \mathbf{e_1} \\ \mathbf{if\ false\ then\ e_1\ else\ e_2} & \longmapsto & \mathbf{e_2} \\ \mathbf{let}\ x = \pi_i(\mathbf{v_1}, \mathbf{v_2})\ \mathbf{in\ e} & \longmapsto & \mathbf{e}[\mathbf{v}_i/x] \\ (\lambda\,[\alpha]\,(x:\tau_1).\,\mathbf{e})\,[\tau]\,\mathbf{v} & \longmapsto & \mathbf{e}[\tau/\alpha][\mathbf{v}/x] \end{array}$$

$$\frac{\mathbf{e} \longmapsto \mathbf{e}'}{\mathbf{E}[\mathbf{e}] \longmapsto \mathbf{E}[\mathbf{e}']}$$

Type Environments $\Delta ::= \cdot \mid \Delta, \alpha$
Value Environments $\Gamma ::= \cdot \mid \Gamma, x:\tau$

$\boxed{\Delta; \Gamma \vdash \mathbf{e} : \tau}$

$$\frac{x:\tau \in \Gamma}{\Delta;\Gamma \vdash x : \tau} \quad \cdots \quad \frac{\Delta;\Gamma \vdash \mathbf{v}_1 : \tau_1 \qquad \Delta;\Gamma \vdash \mathbf{v}_2 : \tau_2}{\Delta;\Gamma \vdash (\mathbf{v}_1, \mathbf{v}_2) : \tau_1 \times \tau_2}$$

$$\frac{\Delta;\Gamma \vdash \mathbf{v} : \tau_1 \times \tau_2 \qquad \Delta;\Gamma, x:\tau_i \vdash \mathbf{e} : \tau}{\Delta;\Gamma \vdash \mathbf{let}\ x = \pi_i \mathbf{v}\ \mathbf{in\ e} : \tau}$$

$$\frac{\Delta, \alpha; \Gamma, x:\tau_1 \vdash \mathbf{e} : \tau_2}{\Delta;\Gamma \vdash \lambda\,[\alpha]\,(x:\tau_1).\,\mathbf{e} : \forall\,[\alpha].\tau_1 \rightarrow \tau_2}$$

$$\frac{\Delta;\Gamma \vdash \mathbf{v}_1 : \forall\,[\alpha].\tau_2 \rightarrow \tau \qquad \Delta \vdash \tau' \qquad \Delta;\Gamma \vdash \mathbf{v}_2 : \tau_2[\tau'/\alpha]}{\Delta;\Gamma \vdash \mathbf{v}_1\,[\tau']\,\mathbf{v}_2 : \tau[\tau'/\alpha]}$$

Value Contexts $\mathbf{C^v} ::= [\cdot]^{\mathbf{v}}_{\mathbf{T}} \mid (\mathbf{C^v}, \mathbf{v}_2) \mid (\mathbf{v}_1, \mathbf{C^v}) \mid \lambda\,[\alpha]\,(x:\tau).\,\mathbf{C}$
Contexts $\mathbf{C} ::= [\cdot]_{\mathbf{T}} \mid \mathbf{C^v} \mid \mathbf{if\ C^v\ then\ e_1\ else\ e_2} \mid \mathbf{if\ e\ then\ C\ else\ e_2} \mid \mathbf{if\ e\ then\ e_1\ else\ C} \mid \mathbf{let}\ x = \pi_i \mathbf{C^v}\ \mathbf{in\ e} \mid \mathbf{let}\ x = \pi_i \mathbf{v}\ \mathbf{in\ C} \mid \mathbf{C^v}\,[\tau]\,\mathbf{v}_2 \mid \mathbf{v}_1\,[\tau]\,\mathbf{C^v}$

Figure 2. λ^{T}: Syntax, Dynamic + Static Semantics, Contexts

the following shorthand (and in the first three cases below, we have analogous shorthand for λ^{S}):

$$\begin{array}{rcll} \lambda x.\,\mathbf{e} & = & \lambda(x:\tau).\,\mathbf{e} & ;\ \tau \text{ inferred from context} \\ \mathbf{let}\ x = \mathbf{v}\ \mathbf{in\ e} & = & (\lambda x.\,\mathbf{e})\,\mathbf{v} & \\ \mathbf{id} & = & \lambda x.\,x & \\ \mathbf{let}\ (x_1, x_2) = \mathbf{z}\ \mathbf{in\ e} & = & \mathbf{let}\ x_1 = \pi_1 \mathbf{z}\ \mathbf{in}\ (\mathbf{let}\ x_2 = \pi_2 \mathbf{z}\ \mathbf{in\ e}) & \\ \lambda\,[\alpha]\,((x_1, x_2):\tau_1 \times \tau_2).\,\mathbf{e} & = & \lambda\,[\alpha]\,(\mathbf{z}:\tau_1 \times \tau_2).\,\mathbf{let}\ (x_1, x_2) = \mathbf{z}\ \mathbf{in\ e} & \end{array}$$

λ^{T} ***Contextual Equivalence and CIU Equivalence*** The syntax of λ^{T} contexts is given at the bottom of Figure 2. A λ^{T} context $\mathbf{C}$ is an expression with a single hole in it, but the hole may be either of the form $[\cdot]^{\mathbf{v}}_{\mathbf{T}}$, a hole that expects a value, or of the form $[\cdot]_{\mathbf{T}}$, a hole that expects any term. We write $\mathbf{C^v}$ (respectively, $\mathbf{C}$) for a context that *once filled*, regardless of the kind of hole in it, yields a value (respectively, a term). Typing rules for λ^{T} contexts are analogous to those for λ^{S} contexts, except that typing judgments have the form $\vdash \mathbf{C} : (\Delta;\Gamma \vdash \tau) \Rightarrow (\Delta';\Gamma' \vdash \tau')$. Hence, if $\mathbf{e}$ is a term such that $\Delta;\Gamma \vdash \mathbf{e} : \tau$, then $\Delta';\Gamma' \vdash \mathbf{C}[\mathbf{e}] : \tau'$. The typing rules for holes $[\cdot]^{\mathbf{v}}_{\mathbf{T}}$ and $[\cdot]_{\mathbf{T}}$ are identical so we show just one:

$$\frac{\Delta \subseteq \Delta' \qquad \Gamma \subseteq \Gamma'}{\vdash [\cdot]^{\mathbf{v}}_{\mathbf{T}} : (\Delta;\Gamma \vdash \tau) \Rightarrow (\Delta';\Gamma' \vdash \tau)}$$

The remaining rules are straightforward (see online appendix [5]). The definition of contextual approximation and equivalence for λ^{T} are analogous to those for λ^{S}, except that we now also have to account for the environment Δ. Also, we must make sure that the terms $\mathbf{C}[\mathbf{e}_1]$ and $\mathbf{C}[\mathbf{e}_2]$ are syntactically well formed λ^{T} terms. The

extra checks are needed because the hole in a context $\mathbf{C}$ may be of the form $[\cdot]_{\mathrm{T}}$ or $[\cdot]^{\mathbf{v}}_{\mathrm{T}}$. If it's of the form $[\cdot]_{\mathrm{T}}$, then any well-typed term $\mathbf{e}$ may be placed in $\mathbf{C}$. However, if it's of the form $[\cdot]^{\mathbf{v}}_{\mathrm{T}}$, then $\mathbf{C}[\mathbf{e}]$ will not be a well-formed term unless $\mathbf{e}$ is a value.

Definition 3.4 (λ^{T} **Contextual Approximation & Equivalence**)
Let $\Delta;\Gamma \vdash \mathbf{e}_1 : \boldsymbol{\tau}$ *and* $\Delta;\Gamma \vdash \mathbf{e}_2 : \boldsymbol{\tau}$.

$$\Delta;\Gamma \vdash \mathbf{e}_1 \precsim^{ctx}_{\mathrm{T}} \mathbf{e}_2 : \boldsymbol{\tau} \stackrel{\text{def}}{=} \forall \mathbf{C}, \mathbf{v}_1.\ \vdash \mathbf{C} : (\Delta;\Gamma \vdash \boldsymbol{\tau}) \Rightarrow (\cdot;\cdot \vdash \mathbf{bool}) \wedge \cdot;\cdot \vdash \mathbf{C}[\mathbf{e}_1] : \mathbf{bool} \wedge \cdot;\cdot \vdash \mathbf{C}[\mathbf{e}_2] : \mathbf{bool} \wedge \mathbf{C}[\mathbf{e}_1] \Downarrow \mathbf{v}_1 \implies \exists \mathbf{v}_2.\ \mathbf{C}[\mathbf{e}_2] \Downarrow \mathbf{v}_2 \wedge \mathbf{v}_1 = \mathbf{v}_2$$

$$\Delta;\Gamma \vdash \mathbf{e}_1 \approx^{ctx}_{\mathrm{T}} \mathbf{e}_2 : \boldsymbol{\tau} \stackrel{\text{def}}{=} \Delta;\Gamma \vdash \mathbf{e}_1 \precsim^{ctx}_{\mathrm{T}} \mathbf{e}_2 : \boldsymbol{\tau} \wedge \Delta;\Gamma \vdash \mathbf{e}_2 \precsim^{ctx}_{\mathrm{T}} \mathbf{e}_1 : \boldsymbol{\tau}$$

λ^{T} evaluation contexts are essentially degenerate. Nonetheless we define ciu-equivalence for λ^{T} since the notion will later be more convenient to work with than contextual equivalence. As before, closed terms of closed type $\boldsymbol{\tau}$ are ciu-equivalent if, in any evaluation context $\mathbf{E}$ with hole type $\boldsymbol{\tau}$ and result type $\mathbf{bool}$, they evaluate to the same value. The notion is extended to open terms by closing the type $\boldsymbol{\tau}$ and the terms with substitutions δ and γ. The type substitution δ is a finite map from type variables $\boldsymbol{\alpha}$ to closed types $\boldsymbol{\tau}$; we write $\delta \models \Delta$ whenever $\mathrm{dom}(\delta) = \Delta$. The value substitution γ now maps variables $\mathbf{x}$ to values $\mathbf{v}$ and $\vdash \gamma : \Gamma$ means that γ maps variables to values that are well-typed according to Γ.

Definition 3.5 (λ^{T} **CIU Approximation & Equivalence**)
Let $\Delta;\Gamma \vdash \mathbf{e}_1 : \boldsymbol{\tau}$ *and* $\Delta;\Gamma \vdash \mathbf{e}_2 : \boldsymbol{\tau}$.

$$\Delta;\Gamma \vdash \mathbf{e}_1 \precsim^{ciu}_{\mathrm{T}} \mathbf{e}_2 : \boldsymbol{\tau} \stackrel{\text{def}}{=} \forall \mathbf{E}, \delta, \gamma, \mathbf{v}_1.\ \vdash \mathbf{E} : (\cdot;\cdot \vdash \delta(\boldsymbol{\tau})) \Rightarrow (\cdot;\cdot \vdash \mathbf{bool}) \wedge \delta \models \Delta \wedge \vdash \gamma : \delta(\Gamma) \wedge \mathbf{E}[\delta(\gamma(\mathbf{e}_1))] \Downarrow \mathbf{v}_1 \implies \exists \mathbf{v}_2.\ \mathbf{E}[\delta(\gamma(\mathbf{e}_2))] \Downarrow \mathbf{v}_2 \wedge \mathbf{v}_1 = \mathbf{v}_2$$

$$\Delta;\Gamma \vdash \mathbf{e}_1 \approx^{ciu}_{\mathrm{T}} \mathbf{e}_2 : \boldsymbol{\tau} \stackrel{\text{def}}{=} \Delta;\Gamma \vdash \mathbf{e}_1 \precsim^{ciu}_{\mathrm{T}} \mathbf{e}_2 : \boldsymbol{\tau} \wedge \Delta;\Gamma \vdash \mathbf{e}_2 \precsim^{ciu}_{\mathrm{T}} \mathbf{e}_1 : \boldsymbol{\tau}$$

Lemma 3.6 (λ^{T}**: Contextual Approx Implies CIU Approx**)
If $\Delta;\Gamma \vdash \mathbf{e}_1 \precsim^{ctx}_{\mathrm{T}} \mathbf{e}_2 : \boldsymbol{\tau}$ *then* $\Delta;\Gamma \vdash \mathbf{e}_1 \precsim^{ciu}_{\mathrm{T}} \mathbf{e}_2 : \boldsymbol{\tau}$.

$$\begin{array}{lllll}
\textit{Types} & \varphi & ::= & \sigma \mid \boldsymbol{\tau} & \\
\textit{Terms} & \mathbf{e} & ::= & \ldots \mid {}^{\sigma}\mathcal{ST}\,\mathbf{e} & \\
 & \mathbf{e} & ::= & \ldots \mid \mathbf{let}\ \mathbf{x} = (\mathcal{TS}\,{}^{\sigma}\,\mathbf{e})\ \mathbf{in}\ \mathbf{e} & \\
 & e & ::= & \mathbf{e} \mid \mathbf{e} & \\
\textit{Values} & v & ::= & \mathbf{v} \mid \mathbf{v} & \\
\textit{Eval. Contexts} & \mathbf{E} & ::= & \ldots \mid {}^{\sigma}\mathcal{ST}\,\mathbf{E} & \\
 & \mathbf{E} & ::= & \ldots \mid \mathbf{let}\ \mathbf{x} = (\mathcal{TS}\,{}^{\sigma}\,\mathbf{E})\ \mathbf{in}\ \mathbf{e} & \\
 & E & ::= & \mathbf{E} \mid \mathbf{E} &
\end{array}$$

$\boxed{e \longmapsto e'}$

$$\begin{array}{rcl}
{}^{\mathbf{bool}}\mathcal{ST}\ \mathbf{true} & \longmapsto & \mathbf{true} \\
{}^{\mathbf{bool}}\mathcal{ST}\ \mathbf{false} & \longmapsto & \mathbf{false} \\
{}^{\sigma_1 \to \sigma_2}\mathcal{ST}\ \mathbf{v} & \longmapsto & \lambda \mathbf{x} : \sigma_1.\ {}^{\sigma_2}\mathcal{ST}\ (\mathbf{let}\ \mathbf{z} = (\mathcal{TS}\,{}^{\sigma_1}\,\mathbf{x})\ \mathbf{in}\ (\mathbf{v}\ [\sigma_2{}^{+}]\ (\mathbf{z}, \mathbf{id})))
\end{array}$$

$$\begin{array}{rcl}
\mathbf{let}\ \mathbf{y} = (\mathcal{TS}\,{}^{\mathbf{bool}}\,\mathbf{true})\ \mathbf{in}\ \mathbf{e} & \longmapsto & \mathbf{e}[\mathbf{true}/\mathbf{y}] \\
\mathbf{let}\ \mathbf{y} = (\mathcal{TS}\,{}^{\mathbf{bool}}\,\mathbf{false})\ \mathbf{in}\ \mathbf{e} & \longmapsto & \mathbf{e}[\mathbf{false}/\mathbf{y}] \\
\mathbf{let}\ \mathbf{y} = (\mathcal{TS}\,{}^{\sigma_1 \to \sigma_2}\,\mathbf{v})\ \mathbf{in}\ \mathbf{e} & \longmapsto & \mathbf{e}[\mathbf{v}/\mathbf{y}] \\
\multicolumn{3}{r}{\text{where } \mathbf{v} = \boldsymbol{\lambda}\,[\boldsymbol{\alpha}]\,((\mathbf{x}, \mathbf{k}) : \sigma_1{}^{+} \times (\sigma_2{}^{+} \to \boldsymbol{\alpha}))\,.\ \mathbf{let}\ \mathbf{z} = (\mathcal{TS}\,{}^{\sigma_2}\,(\mathbf{v}\ ({}^{\sigma_1}\mathcal{ST}\ \mathbf{x})))\ \mathbf{in}\ \mathbf{k}\ \mathbf{z}}
\end{array}$$

$$\frac{e \longmapsto e'}{E[e] \longmapsto E[e']}$$

$$\begin{array}{lllll}
\textit{Type Environments} & \Delta & ::= & \cdot \mid \Delta, \boldsymbol{\alpha} & \\
\textit{Value Environments} & \Gamma & ::= & \cdot \mid \Gamma, \mathbf{x} : \sigma \mid \Gamma, \mathbf{x} : \boldsymbol{\tau} &
\end{array}$$

$\boxed{\Delta;\Gamma \vdash e : \varphi}$

$$\cdots \quad \frac{\Delta;\Gamma \vdash \mathbf{e} : \sigma^{+}}{\Delta;\Gamma \vdash {}^{\sigma}\mathcal{ST}\ \mathbf{e} : \sigma} \qquad \frac{\Delta;\Gamma \vdash \mathbf{e} : \sigma \qquad \Delta;\Gamma, \mathbf{x} : \sigma^{+} \vdash \mathbf{e} : \boldsymbol{\tau}}{\Delta;\Gamma \vdash \mathbf{let}\ \mathbf{x} = (\mathcal{TS}\,{}^{\sigma}\,\mathbf{e})\ \mathbf{in}\ \mathbf{e} : \boldsymbol{\tau}}$$

$$\begin{array}{lll}
\textit{Contexts} & \mathbf{C} ::= \ldots \mid {}^{\sigma}\mathcal{ST}\,\mathbf{C} & \\
 & \mathbf{C} ::= \ldots \mid \mathbf{let}\ \mathbf{x} = (\mathcal{TS}\,{}^{\sigma}\,\mathbf{C})\ \mathbf{in}\ \mathbf{e} \mid \mathbf{let}\ \mathbf{x} = (\mathcal{TS}\,{}^{\sigma}\,\mathbf{e})\ \mathbf{in}\ \mathbf{C} & \\
 & C ::= \mathbf{C} \mid \mathbf{C} &
\end{array}$$

Figure 4. λ^{ST}: Syntax, Dynamic + Static Semantics, Contexts

4. Typed CPS Translation

CPS translation maps source values of type σ to target values of type σ^{+}. The type translation is given in Figure 3 (top).

As mentioned earlier, our version of typed CPS-conversion differs from the "standard" account. Instead of using a single, globally abstract answer type, we make the answer type individually abstract at each point where a continuation argument appears. As shown in Figure 3 (top left), each function of type $\sigma_1 \to \sigma_2$ acquires a type parameter $\boldsymbol{\alpha}$ and an additional term parameter of type ${\sigma_2}^{+} \to \boldsymbol{\alpha}$ representing the continuation. The polymorphic type variable $\boldsymbol{\alpha}$ acts as the *answer* type. The consequence of this is that a function of CPS type has less freedom in how it can use continuations. In particular, to produce its own answer of type $\boldsymbol{\alpha}$, it has no choice but to invoke *its own* continuation. Of course, the function can invoke its continuation multiple times, but in a purely functional setting it is impossible to tell the difference between one use of the continuation and multiple uses. As discussed earlier, this typing of CPS code prevents any "bad" target terms from being well typed. The polymorphic type of each continuation lets us take advantage of a *free theorem* [50] that captures the above intuitions and plays a key role in our proof; see discussion of Lemma 6.10 in Section 6.3.

Figure 3 (bottom) shows the rules for CPS translation in combination with declarative typing rules for the source language λ^{S}. To this end, the typing judgment $\Gamma \vdash \mathsf{e} : \sigma$ is extended to a translation judgment $\Gamma \vdash \mathsf{e} : \sigma \rightsquigarrow \mathbf{v}$, where $\mathbf{v}$ describes a CPS *computation* of type $\forall\,[\boldsymbol{\alpha}].(\sigma^{+} \to \boldsymbol{\alpha}) \to \boldsymbol{\alpha}$ (which we abbreviate as $\sigma^{\div}$). A computation is a suspended term awaiting its continuation. It is polymorphic in the answer type of the continuation and, like functions, it will be forced to invoke its own continuation to produce its answer of type $\boldsymbol{\alpha}$.

Note that for $\mathbf{v}$ to have the type $\sigma^{\div}$, it has to be considered under the translated environment Γ^{+} (defined in Figure 3, top right). Therefore, notice that variables x in the source term are replaced by variables $\mathbf{x}$ in the target term. For instance, in the variable translation rule, the source variable x changes to $\mathbf{x}$ in the target. Also, in the function rule, note that $\mathbf{v}$ in the premise may contain a free occurrence of $\mathbf{x} : {\sigma_1}^{+}$. This $\mathbf{x}$ is captured in the conclusion by a binding occurrence of $\mathbf{x}$. We could parametrize our translation with a mapping from source variables to target variables and use that to perform renaming. To avoid this unnecessary complication, we work with a fixed a-priori mapping and adopt the convention that for each x, y, z, etc., there exists a unique $\mathbf{x}$, $\mathbf{y}$, $\mathbf{z}$, respectively, that we can use in the translated term.

The rules are designed with clarity, not optimality, in mind. A translation term $\mathbf{v}$ will usually contain many "administrative" redexes that can easily be eliminated with subsequent optimization.

5. Multi-Language Semantics

In this section, we present the language λ^{ST} designed for interoperability of terms from the source and target languages λ^{S} and λ^{T} and give a definition of contextual equivalence (written $\approx^{ctx}_{\mathrm{ST}}$) for the language.

The λ^{ST} multi-language system, presented in Figure 4, embeds the source and target languages λ^{S} and λ^{T} so that both languages have natural access to foreign values (i.e., values from the other

$\boxed{\sigma^+}$

$$(\mathsf{bool})^+ = \mathbf{bool} \qquad (\sigma_1 \to \sigma_2)^+ = \forall\,[\alpha].(\sigma_1{}^+ \times (\sigma_2{}^+ \to \alpha)) \to \alpha$$

$\boxed{\Gamma^+}$

$$(\cdot)^+ = \cdot \qquad (\Gamma, x : \sigma)^+ = \Gamma^+, \mathbf{x} : \sigma^+$$

$\boxed{\Gamma \vdash \mathsf{e} : \sigma \rightsquigarrow \mathbf{v}}$ where $\cdot ; \Gamma^+ \vdash \mathbf{v} : \sigma^\div$ and we define $\sigma^\div = \forall\,[\alpha].(\sigma^+ \to \alpha) \to \alpha$

$$\frac{}{\Gamma \vdash \mathsf{true} : \mathsf{bool} \rightsquigarrow \lambda\,[\alpha]\,(\mathbf{k} : \mathsf{bool}^+ \to \alpha).\ \mathbf{k}\ \mathbf{true}} \qquad \frac{}{\Gamma \vdash \mathsf{false} : \mathsf{bool} \rightsquigarrow \lambda\,[\alpha]\,(\mathbf{k} : \mathsf{bool}^+ \to \alpha).\ \mathbf{k}\ \mathbf{false}}$$

$$\frac{\Gamma \vdash \mathsf{e} : \mathsf{bool} \rightsquigarrow \mathbf{v} \qquad \Gamma \vdash \mathsf{e}_1 : \sigma \rightsquigarrow \mathbf{v}_1 \qquad \Gamma \vdash \mathsf{e}_2 : \sigma \rightsquigarrow \mathbf{v}_2}{\Gamma \vdash \mathsf{if\ e\ then\ e_1\ else\ e_2} : \sigma \rightsquigarrow \lambda\,[\alpha]\,(\mathbf{k} : \sigma^+ \to \alpha).\ \mathbf{v}\ [\alpha]\ (\lambda(\mathbf{x} : \mathbf{bool}).\ \mathbf{if\ x\ then}\ (\mathbf{v}_1\ [\alpha]\ \mathbf{k})\ \mathbf{else}\ (\mathbf{v}_2\ [\alpha]\ \mathbf{k}))}$$

$$\frac{x : \sigma \in \Gamma}{\Gamma \vdash x : \sigma \rightsquigarrow \lambda\,[\alpha]\,(\mathbf{k} : \sigma^+ \to \alpha).\ \mathbf{k}\ \mathbf{x}} \qquad \frac{\Gamma, x : \sigma_1 \vdash \mathsf{e} : \sigma_2 \rightsquigarrow \mathbf{v}}{\Gamma \vdash \lambda x : \sigma_1.\,\mathsf{e} : \sigma_1 \to \sigma_2 \rightsquigarrow \lambda\,[\alpha]\,(\mathbf{k} : (\sigma_1 \to \sigma_2)^+ \to \alpha).\ \mathbf{k}\ (\lambda\,[\beta]\,((\mathbf{x}, \mathbf{k'}) : \sigma_1{}^+ \times (\sigma_2{}^+ \to \beta)).\ (\mathbf{v}\ [\beta]\ \mathbf{k'}))}$$

$$\frac{\Gamma \vdash \mathsf{e}_1 : \sigma_2 \to \sigma \rightsquigarrow \mathbf{v}_1 \qquad \Gamma \vdash \mathsf{e}_2 : \sigma_2 \rightsquigarrow \mathbf{v}_2}{\Gamma \vdash \mathsf{e}_1\ \mathsf{e}_2 : \sigma \rightsquigarrow \lambda\,[\alpha]\,(\mathbf{k} : \sigma^+ \to \alpha).\ \mathbf{v}_1\ [\alpha]\ (\lambda(\mathbf{x}_1 : (\sigma_2 \to \sigma)^+).\ \mathbf{v}_2\ [\alpha]\ (\lambda(\mathbf{x}_2 : \sigma_2{}^+).\ \mathbf{x}_1\ [\alpha]\ (\mathbf{x}_2, \mathbf{k})))}$$

Figure 3. CPS: Type and Environment Translation (top); Term Translation (bottom)

language). They receive foreign boolean values as native values, and can call foreign functions as native functions. The design is inspired by Matthews and Findler's multi-language system for (pared down) ML and Scheme [31], but crafted with the CPS translation in mind.

To the original core languages, we add new syntax, evaluation contexts, and reduction rules that define syntactic boundaries, written ${}^{\sigma}\mathcal{ST}$ and $\mathcal{TS}^{\sigma}$ to allow cross-language communication. The term ${}^{\sigma}\mathcal{ST}\ \mathbf{e}$ (target inside, source outside) allows a term $\mathbf{e}$ of target type σ^+ to be used as a term of source type σ, while $\mathcal{TS}^{\sigma}\ \mathsf{e}$ (source inside, target outside) allows a term of source type σ to be used as a term of target type σ^+. Since code in our CPS target language is (nearly) linear, we let-bind $\mathcal{TS}^{\sigma}\ \mathsf{e}$ instead of simply adding $\mathcal{TS}^{\sigma}\ \mathsf{e}$ to the grammar for terms. (Here, we do not require e to be a value since, informally, e lives on the source side of the boundary and the source language does not mandate the linear code structure of the target.) In the interest of brevity, we will often abbreviate $\mathbf{let\ z} = (\mathcal{TS}^{\sigma}\ \mathsf{e})\ \mathbf{in\ C}[\mathbf{z}]$ to just $\mathbf{C}[\mathcal{TS}^{\sigma}\ \mathsf{e}]$ even if $\mathbf{C}$ requires a value.

We define reduction rules for boundaries annotated bool that convert boolean values from one language to the other. To convert functions across languages, we use native proxy functions. We represent a target function $\mathbf{v}$ in the source at type $\sigma_1 \to \sigma_2$ by a new function that takes an argument of type σ_1, converts it to its equivalent in the target, passes that and the identity continuation as an argument to the original target function $\mathbf{v}$ (instantiated with the result type $\sigma_2{}^+$), and converts the result back to source at type σ_2. Converting source functions to target functions is a little more involved. We represent a source function v in the target at type $(\sigma_1 \to \sigma_2)^+$ by a new function that takes type parameter $\boldsymbol{\alpha}$, an argument $\mathbf{x} : \sigma_1{}^+$ and a continuation $\mathbf{k} : \sigma_2{}^+ \to \boldsymbol{\alpha}$, converts the argument $\mathbf{x}$ to its equivalent in the source, passes that to the original source function v, converts the result back to target at type $\sigma_2{}^+$, and finally passes that to the continuation $\mathbf{k}$. In both cases, notice that the direction of the conversion (and the boundary used) reverses for function arguments.

Typing judgments for λ^{ST} have the form $\Delta; \Gamma \vdash e : \varphi$, where the environments Δ and Γ are defined in Figure 4. Note that the environment Γ now tracks both source variables of type σ and target variables of type $\boldsymbol{\tau}$. The typing rules include all the λ^{S} typing rules, but augmented with the additional environment Δ; all the λ^{T} typing rules, unchanged; and rules for the two boundary constructs, shown in Figure 4.

λ^{ST} Contextual Equivalence and CIU Equivalence The grammar for contexts from λ^{S} and λ^{T} is augmented to define λ^{ST} contexts as shown in Figure 4 (bottom). The typing rules for contexts are straightforward (see online appendix [5]), largely following the ideas discussed before for λ^{T}. The definition of contextual equivalence for λ^{ST} is similar to that for λ^{T} and is given below. In the definition below, we could equivalently have chosen the context C's result type to be $\mathbf{bool}$ instead of bool. As is usually the case for contextual equivalence in a terminating language, any base type of the language would suffice.

Definition 5.1 (Contextual Approximation & Equivalence)
Let $\Delta; \Gamma \vdash e_1 : \varphi$ *and* $\Delta; \Gamma \vdash e_2 : \varphi$.

$$\begin{array}{l}\Delta; \Gamma \vdash e_1 \precsim^{ctx}_{\mathrm{ST}} e_2 : \varphi \stackrel{\mathrm{def}}{=} \\ \quad \forall C, v_1.\ \vdash C : (\Delta; \Gamma \vdash \varphi) \Rightarrow (\cdot; \cdot \vdash \mathsf{bool}) \ \wedge\ \cdot; \cdot \vdash C[e_1] : \mathsf{bool} \ \wedge \\ \qquad\qquad \cdot; \cdot \vdash C[e_2] : \mathsf{bool} \ \wedge\ C[e_1] \Downarrow v_1 \implies \\ \qquad\qquad \exists v_2.\ C[e_2] \Downarrow v_2 \ \wedge\ v_1 = v_2\end{array}$$

$$\Delta; \Gamma \vdash e_1 \approx^{ctx}_{\mathrm{ST}} e_2 : \varphi \stackrel{\mathrm{def}}{=} \Delta; \Gamma \vdash e_1 \precsim^{ctx}_{\mathrm{ST}} e_2 : \varphi \wedge \Delta; \Gamma \vdash e_2 \precsim^{ctx}_{\mathrm{ST}} e_1 : \varphi$$

We define ciu-equivalence for λ^{ST} below. It is analogous to the definition of ciu-equivalence for λ^{T}. As before, the substitution δ maps type variables $\boldsymbol{\alpha}$ to closed types $\boldsymbol{\tau}$. The substitution γ now maps both source variables (to values v) and target variables (to values $\mathbf{v}$) in Γ.

Definition 5.2 (λ^{ST} CIU Approximation & Equivalence)
Let $\Delta; \Gamma \vdash e_1 : \varphi$ *and* $\Delta; \Gamma \vdash e_2 : \varphi$.

$$\begin{array}{l}\Delta; \Gamma \vdash e_1 \precsim^{ciu}_{\mathrm{ST}} e_2 : \varphi \stackrel{\mathrm{def}}{=} \\ \quad \forall E, \delta, \gamma, v_1.\ \vdash E : (\cdot; \cdot \vdash \delta(\varphi)) \Rightarrow (\cdot; \cdot \vdash \mathsf{bool}) \ \wedge \\ \qquad\qquad \delta \models \Delta \ \wedge\ \vdash \gamma : \delta(\Gamma) \ \wedge\ E[\delta(\gamma(e_1))] \Downarrow v_1 \implies \\ \qquad\qquad \exists v_2.\ E[\delta(\gamma(e_2))] \Downarrow v_2 \ \wedge\ v_1 = v_2\end{array}$$

$$\Delta; \Gamma \vdash e_1 \approx^{ciu}_{\mathrm{ST}} e_2 : \varphi \stackrel{\mathrm{def}}{=} \Delta; \Gamma \vdash e_1 \precsim^{ciu}_{\mathrm{ST}} e_2 : \varphi \wedge \Delta; \Gamma \vdash e_2 \precsim^{ciu}_{\mathrm{ST}} e_1 : \varphi$$

Lemma 5.3 (λ^{ST}: Contextual Approx Implies CIU Approx)
If $\Delta; \Gamma \vdash e_1 \precsim^{ctx}_{\mathrm{ST}} e_2 : \varphi$ *then* $\Delta; \Gamma \vdash e_1 \precsim^{ciu}_{\mathrm{ST}} e_2 : \varphi$.

6. Equivalence Preservation

Having defined λ^{ST} for source-language interoperability, the proof of equivalence preservation can be decomposed into three parts:

1. if $\Gamma \vdash \mathsf{e}_1 \approx^{ctx}_{\mathrm{S}} \mathsf{e}_2 : \sigma$ then $\Gamma \vdash \mathsf{e}_1 \approx^{ctx}_{\mathrm{ST}} \mathsf{e}_2 : \sigma$ (Section 6.2);
2. if $\Gamma \vdash \mathsf{e}_1 \approx^{ctx}_{\mathrm{ST}} \mathsf{e}_2 : \sigma$ then $\cdot; \Gamma^+ \vdash \mathbf{v}_1 \approx^{ctx}_{\mathrm{ST}} \mathbf{v}_2 : \sigma^\div$, where $\Gamma \vdash \mathsf{e}_1 : \sigma \rightsquigarrow \mathbf{v}_1$ and $\Gamma \vdash \mathsf{e}_2 : \sigma \rightsquigarrow \mathbf{v}_2$ (Section 6.3); and

$$\mathrm{Atom}[\varphi_1,\varphi_2] = \{\,(e_1,e_2) \mid \cdot;\cdot \vdash e_1 : \varphi_1 \ \wedge\ \cdot;\cdot \vdash e_2 : \varphi_2\,\}$$

$$\mathrm{Rel}[\tau_1,\tau_2] = \{\,R \in \mathscr{P}(\mathrm{Atom}^{\mathrm{val}}[\tau_1,\tau_2]) \mid \forall (v_1,v_2)\in R.\ \forall v_2'.\ v_2 \precsim^{ciu}_{\mathrm{ST}} v_2' : \tau_2 \implies (v_1,v_2') \in R\,\}$$

Shorthand: $\mathrm{Atom}[\varphi]\rho = \mathrm{Atom}[\rho_1(\varphi),\rho_2(\varphi)]$

$$\mathcal{V}[\![\mathsf{bool}]\!]\rho = \{\,(\mathsf{v},\mathsf{v}) \in \mathrm{Atom}[\mathsf{bool}]\rho \mid \mathsf{v} = \mathsf{true} \ \vee\ \mathsf{v} = \mathsf{false}\,\}$$

$$\mathcal{V}[\![\sigma\to\sigma']\!]\rho = \{\,(\lambda x{:}\sigma.\,e_1, \lambda x{:}\sigma.\,e_2) \in \mathrm{Atom}[\sigma\to\sigma']\rho \mid \forall (v_1,v_2)\in\mathcal{V}[\![\sigma]\!]\rho.\ (e_1[v_1/x], e_2[v_2/x]) \in \mathcal{E}[\![\sigma']\!]\rho\,\}$$

$$\mathcal{V}[\![\alpha]\!]\rho = R \qquad \text{where } \rho(\alpha) = (\tau_1,\tau_2,R)$$

$$\mathcal{V}[\![\mathbf{bool}]\!]\rho = \{\,(\mathbf{v},\mathbf{v}) \in \mathrm{Atom}[\mathbf{bool}]\rho \mid \mathbf{v} = \mathbf{true} \ \vee\ \mathbf{v} = \mathbf{false}\,\}$$

$$\mathcal{V}[\![\tau\times\tau']\!]\rho = \{\,((\mathbf{v}_1,\mathbf{v}_1'),(\mathbf{v}_2,\mathbf{v}_2')) \in \mathrm{Atom}[\tau\times\tau']\rho \mid (\mathbf{v}_1,\mathbf{v}_2)\in\mathcal{V}[\![\tau]\!]\rho \ \wedge\ (\mathbf{v}_1',\mathbf{v}_2')\in\mathcal{V}[\![\tau']\!]\rho\,\}$$

$$\mathcal{V}[\![\forall\,[\alpha].\tau\to\tau']\!]\rho = \{\,(\Lambda\,[\alpha]\,(\mathbf{x}:\rho_1(\tau)).\,\mathbf{e}_1, \Lambda\,[\alpha]\,(\mathbf{x}:\rho_2(\tau)).\,\mathbf{e}_2) \in \mathrm{Atom}[\forall\,[\alpha].\tau\to\tau']\rho \mid \forall\tau_1,\tau_2,R\in\mathrm{Rel}[\tau_1,\tau_2].\ \forall(\mathbf{v}_1,\mathbf{v}_2)\in\mathcal{V}[\![\tau]\!]\rho[\alpha\mapsto(\tau_1,\tau_2,R)].\ (\mathbf{e}_1[\tau_1/\alpha][\mathbf{v}_1/\mathbf{x}], \mathbf{e}_2[\tau_2/\alpha][\mathbf{v}_2/\mathbf{x}]) \in \mathcal{E}[\![\tau']\!]\rho[\alpha\mapsto(\tau_1,\tau_2,R)]\,\}$$

$$\mathcal{E}[\![\varphi]\!]\rho = \{\,(e_1,e_2) \in \mathrm{Atom}[\varphi]\rho \mid \forall v_1.\ e_1 \longmapsto^* v_1 \implies \exists v_2.\ e_2 \longmapsto^* v_2 \ \wedge\ (v_1,v_2)\in\mathcal{V}[\![\varphi]\!]\rho\,\}$$

$$\mathcal{D}[\![\cdot]\!] = \{\,\emptyset\,\} \qquad \mathcal{D}[\![\Delta,\alpha]\!] = \{\,\rho[\alpha\mapsto(\tau_1,\tau_2,R)] \mid \rho\in\mathcal{D}[\![\Delta]\!] \ \wedge\ R\in\mathrm{Rel}[\tau_1,\tau_2]\,\}$$

$$\mathcal{G}[\![\cdot]\!]\rho = \{\,(\emptyset,\emptyset)\,\} \qquad \mathcal{G}[\![\Gamma,x:\varphi]\!]\rho = \{\,(\gamma_1[x\mapsto v_1],\gamma_2[x\mapsto v_2]) \mid (\gamma_1,\gamma_2)\in\mathcal{G}[\![\Gamma]\!]\rho \ \wedge\ (v_1,v_2)\in\mathcal{V}[\![\varphi]\!]\rho\,\}$$

$$\Delta;\Gamma\vdash e_1 \precsim^{log}_{\mathrm{ST}} e_2 : \varphi \stackrel{\mathrm{def}}{=} \Delta;\Gamma\vdash e_1:\varphi \ \wedge\ \Delta;\Gamma\vdash e_2:\varphi \ \wedge\ \forall\rho,\gamma_1,\gamma_2.\ \rho\in\mathcal{D}[\![\Delta]\!]\wedge(\gamma_1,\gamma_2)\in\mathcal{G}[\![\Gamma]\!]\rho \implies (\rho_1(\gamma_1(e_1)),\rho_2(\gamma_2(e_2)))\in\mathcal{E}[\![\varphi]\!]\rho$$

$$\Delta;\Gamma\vdash e_1 \approx^{log}_{\mathrm{ST}} e_2 : \varphi \stackrel{\mathrm{def}}{=} \Delta;\Gamma\vdash e_1 \precsim^{log}_{\mathrm{ST}} e_2:\varphi \ \wedge\ \Delta;\Gamma\vdash e_2 \precsim^{log}_{\mathrm{ST}} e_1:\varphi$$

Figure 5. Combined Language (λ^{ST}): Logical Relation

3. if $\cdot;\Gamma^+ \vdash \mathbf{v}_1 \approx^{ctx}_{\mathrm{ST}} \mathbf{v}_2 : \sigma^\div$ then $\cdot;\Gamma^+ \vdash \mathbf{v}_1 \approx^{ctx}_{\mathrm{T}} \mathbf{v}_2 : \sigma^\div$ (Section 6.4).

We shall see that parts (1) and (2) are the nontrivial parts of the proof. Informally, part (1) says that embedding λ^{S} into λ^{ST} preserves λ^{S} equivalences, while part (2) says that the translation preserves equivalences within λ^{ST}. Once we have proved (1), (2), and (3), our main result is immediate:

Theorem 6.1 (CPS Translation is Equivalence Preserving)
If $\Gamma\vdash \mathsf{e}_1 : \sigma \rightsquigarrow \mathbf{v}_1$, $\Gamma\vdash \mathsf{e}_2 : \sigma \rightsquigarrow \mathbf{v}_2$, *and* $\Gamma \vdash \mathsf{e}_1 \approx^{ctx}_{\mathrm{S}} \mathsf{e}_2 : \sigma$, *then* $\cdot;\Gamma^+ \vdash \mathbf{v}_1 \approx^{ctx}_{\mathrm{T}} \mathbf{v}_2 : \sigma^\div$.

We start in Section 6.1 by setting up some machinery for our proof. We define a logical relation for λ^{ST} and prove that it coincides with contextual equivalence. Proving contextual equivalence directly can be hard or even intractable due to the quantification over all contexts C in the definition of $\approx^{ctx}_{\mathrm{ST}}$. The logical relation provides us with a convenient method for carrying out proofs of contextual equivalence. Next, in Sections 6.2-6.4, we present the three parts of the proof.

6.1 Logical Relation for λ^{ST} and Two Key Lemmas

The basic idea of logical relations is to define an equivalence (or approximation) relation on program terms by induction on the structure of their types. For instance, we would say that two functions are logically related at the type $\sigma_1\to\sigma_2$ iff, when applied to arguments that are logically related at σ_1, they yield results that are logically related at σ_2. To take the example of product types, we would say that two pairs are logically related at the type $\tau_1\times\tau_2$ iff their first and second components are pairwise related at the types τ_1 and τ_2, respectively.

Figure 5 presents the definition of the logical relation for λ^{ST}. The big picture is that we define a relation $\mathcal{V}[\![\varphi]\!]$ that relates closed values at type φ and a relation $\mathcal{E}[\![\varphi]\!]$ that relates closed terms at type φ, and then generalize the definition of relatedness to open terms (written $\Delta;\Gamma \vdash e_1 \precsim^{log}_{\mathrm{ST}} e_2 : \varphi$). So far these relations define logical approximation and are intended to capture the notion of contextual approximation; we define logical equivalence (written $\Delta;\Gamma\vdash e_1 \approx^{log}_{\mathrm{ST}} e_2 : \varphi$) as logical approximation in both directions.

In more detail, the value relation $\mathcal{V}[\![\varphi]\!]$ is parametrized by a parameter ρ that provides relational interpretations R for the free type variables in φ. We must make sure that these relations R satisfy certain requirements (enforced by requiring that R belong to $\mathrm{Rel}[\tau_1,\tau_2]$, which we explain momentarily). The first requirement is that R should relate only well-typed closed values. To this end, we first define $\mathrm{Atom}[\varphi_1,\varphi_2]$ to be the set of all pairs of well-typed closed terms e_1 and e_2 of types φ_1 and φ_2, respectively. We write $\mathrm{Atom}^{\mathrm{val}}$ to restrict the above set to pairs of values. The second requirement is that R must be *equivalence-respecting*. The latter means that R must be closed under equivalence (or more precisely under approximation): if $(v_1,v_2)\in R$ (where $v_2 : \tau_2$) and $v_2 \precsim^{ciu}_{\mathrm{ST}} v_2' : \tau_2$, then $(v_1,v_2')\in R$. Both of these requirements are enforced by $\mathrm{Rel}[\tau_1,\tau_2]$, which we define as the set of all relations R that contain values of types τ_1 and τ_1 with the additional requirement that these relations must be equivalence-respecting. (Note that the equivalence-respecting requirement is needed to prove that our logical relation is *complete* with respect to contextual equivalence.)

The parameter ρ is a finite map from type variables α to triples (τ_1,τ_2,R), where τ_1 and τ_2 are closed types and $R\in\mathrm{Rel}[\tau_1,\tau_2]$. We define abbreviations for projecting the type components of the triple as follows. If $\rho(\alpha) = (\tau_1,\tau_2,R)$, then $\rho_1(\alpha)=\tau_1$ and $\rho_2(\alpha)=\tau_2$.

The rest of the definition of the logical relation is essentially standard. All value relations $\mathcal{V}[\![\varphi]\!]\rho$ consist of pairs (v_1,v_2) where $\cdot;\cdot\vdash v_1 : \rho_1(\varphi)$ and $\cdot;\cdot\vdash v_2 : \rho_2(\varphi)$ (and similarly for term relations $\mathcal{E}[\![\varphi]\!]\rho$). Two values are related at the type α if they are in the relation R in $\rho(\alpha)$. Two values are related at the type bool (similarly, $\mathbf{bool}$) if they are equal. Two functions are related at $\sigma\to\sigma'$ if, when applied to arguments related at σ, they beta reduce to terms related at σ'—i.e., the latter must be terms that belong to the term relation $\mathcal{E}[\![\sigma']\!]$. The term relation $\mathcal{E}[\![\varphi]\!]\rho$ relates terms e_1 and e_2 if, when e_1 evaluates to v_1, then e_2 evaluates to some v_2 such that v_1 and v_2 are related in $\mathcal{V}[\![\varphi]\!]\rho$. Finally, the relation $\mathcal{V}[\![\forall\,[\alpha].\tau\to\tau']\!]\rho$, relates polymorphic functions. As expected, it considers arbitrary types τ_1, τ_2, together with an arbitrary relation $R\in\mathrm{Rel}[\tau_1,\tau_2]$, and two arguments $\mathbf{v}_1$ and $\mathbf{v}_2$ related at τ (with ρ extended to map α to (τ_1,τ_2,R), since α may appear free in τ). Then, the two polymorphic functions are considered related at type $\forall\,[\alpha].\tau\to\tau'$ if, when applied, respectively, to the type arguments τ_1 and τ_2, and the value arguments $\mathbf{v}_1$ and $\mathbf{v}_2$, they beta reduce to terms that are related at type τ' (again, with $\rho[\alpha\mapsto(\tau_1,\tau_2,R)]$).

The definitions of logical approximation and equivalence for open terms (at the bottom of Figure 5) rely on the relational semantics ascribed to the contexts Δ and Γ. We say that ρ belongs to the relational interpretation of Δ if $\mathrm{dom}(\rho)=\Delta$, and whenever

$\rho(\alpha) = (\tau_1, \tau_2, R)$, R is a well-formed relational interpretation (i.e., $R \in \mathrm{Rel}[\tau_1, \tau_2]$). We say the value substitutions γ_1 and γ_2 are related at Γ if they map variables in $\mathrm{dom}(\Gamma)$ to related values.

The definition of the logical relation for open terms, $\Delta; \Gamma \vdash e_1 \lesssim^{log}_{\mathrm{ST}} e_2 : \varphi$ (pronounced "e_1 logically approximates e_2"), is also standard. It says that given a relational interpretation ρ for Δ and value substitutions γ_1, γ_2 related at Γ, the closed terms $\rho_1(\gamma_1(e_1))$ and $\rho_2(\gamma_2(e_2))$ are related at the type φ. Finally, we say that e_1 and e_2 are logically equivalent, $\Delta; \Gamma \vdash e_1 \approx^{log}_{\mathrm{ST}} e_2 : \varphi$, if they logically approximate each other.

Properties of the Logical Relation We prove the fundamental property of logical relations, which says that if a term is well typed, then it is related to itself. This follows from the proofs of a series of compatibility lemmas (e.g., see [38]). The proofs of most of these lemmas are standard, exactly as for any logical relation for System F. The two that are interesting are the compatibility lemmas for boundaries, which we state below. These require the following "bridge" lemma. (Further details and proofs are given in the online technical appendix [5].)

Lemma 6.2 (Bridge Lemma)

1. *If* $(\mathbf{e}_1, \mathbf{e}_2) \in \mathcal{E}\,[\![\sigma^+]\!]\,\emptyset$ *then* $({}^{\sigma}\mathcal{ST}\,\mathbf{e}_1, {}^{\sigma}\mathcal{ST}\,\mathbf{e}_2) \in \mathcal{E}\,[\![\sigma]\!]\,\emptyset$.
2. *If* $(\mathbf{e}_1, \mathbf{e}_2) \in \mathcal{E}\,[\![\sigma]\!]\,\emptyset$ *and* $(\lambda(\mathbf{x}:\sigma^+).\,\mathbf{e}_1', \lambda(\mathbf{x}:\sigma^+).\,\mathbf{e}_2') \in \mathcal{V}\,[\![\sigma^+ \to \tau]\!]\,\rho$, *then* $(\mathbf{let}\ \mathbf{x} = (\mathcal{TS}\,{}^{\sigma}\,\mathbf{e}_1)\ \mathbf{in}\ \mathbf{e}_1', \mathbf{let}\ \mathbf{x} = (\mathcal{TS}\,{}^{\sigma}\,\mathbf{e}_2)\ \mathbf{in}\ \mathbf{e}_2') \in \mathcal{E}\,[\![\tau]\!]\,\rho$.

Lemma 6.3 (Compatibility $\mathcal{ST}$)
If $\Delta; \Gamma \vdash \mathbf{e}_1 \lesssim^{log}_{\mathrm{ST}} \mathbf{e}_2 : \sigma^+$ *then* $\Delta; \Gamma \vdash {}^{\sigma}\mathcal{ST}\,\mathbf{e}_1 \lesssim^{log}_{\mathrm{ST}} {}^{\sigma}\mathcal{ST}\,\mathbf{e}_2 : \sigma$.

Lemma 6.4 (Compatibility $\mathcal{TS}$)
If $\Delta; \Gamma \vdash \mathbf{e}_1 \lesssim^{log}_{\mathrm{ST}} \mathbf{e}_2 : \sigma$ *and* $\Delta; \Gamma, \mathbf{x} : \sigma^+ \vdash \mathbf{e}_1 \lesssim^{log}_{\mathrm{ST}} \mathbf{e}_2 : \tau$ *then* $\Delta; \Gamma \vdash \mathbf{let}\ \mathbf{x} = (\mathcal{TS}\,{}^{\sigma}\,\mathbf{e}_1)\ \mathbf{in}\ \mathbf{e}_1 \lesssim^{log}_{\mathrm{ST}} \mathbf{let}\ \mathbf{x} = (\mathcal{TS}\,{}^{\sigma}\,\mathbf{e}_2)\ \mathbf{in}\ \mathbf{e}_2 : \tau$.

Theorem 6.5 (λ^{ST} Fundamental Property)
If $\Delta; \Gamma \vdash e : \varphi$ *then* $\Delta; \Gamma \vdash e \lesssim^{log}_{\mathrm{ST}} e : \varphi$.

Proof By induction on the derivation $\Delta; \Gamma \vdash e : \varphi$. Each case follows from the corresponding compatibility lemma. □

Next, we prove that the λ^{ST} logical relation is *sound* and *complete* with respect to contextual equivalence. The key thing here is that the following lemmas (along with the property that $\lesssim^{ctx}_{\mathrm{ST}}$ implies $\lesssim^{ciu}_{\mathrm{ST}}$, Lemma 5.3), together establish that logical approximation $\lesssim^{log}_{\mathrm{ST}}$, ciu-approximation $\lesssim^{ciu}_{\mathrm{ST}}$, and contextual approximation $\lesssim^{ctx}_{\mathrm{ST}}$ all coincide. Therefore, in subsequent sections, when proving contextual equivalence properties, we are free to switch to whichever definition is most convenient to work with for proving the property at hand.

Theorem 6.6 (λ^{ST}: Soundness w.r.t. Contextual Approx)
If $\Delta; \Gamma \vdash e_1 \lesssim^{log}_{\mathrm{ST}} e_2 : \varphi$ *then* $\Delta; \Gamma \vdash e_1 \lesssim^{ctx}_{\mathrm{ST}} e_2 : \varphi$.

Lemma 6.7 (λ^{ST}: CIU Approx Implies Logical Approx)
If $\Delta; \Gamma \vdash e_1 \lesssim^{ciu}_{\mathrm{ST}} e_2 : \varphi$ *then* $\Delta; \Gamma \vdash e_1 \lesssim^{log}_{\mathrm{ST}} e_2 : \varphi$.

Theorem 6.8 (λ^{ST}: Completeness w.r.t. Contextual Approx)
If $\Delta; \Gamma \vdash e_1 \lesssim^{ctx}_{\mathrm{ST}} e_2 : \varphi$ *then* $\Delta; \Gamma \vdash e_1 \lesssim^{log}_{\mathrm{ST}} e_2 : \varphi$.

Proof Immediate from Lemmas 5.3 and 6.7. □

Two Key Lemmas We use our logical relation to establish two key properties that we will need repeatedly when proving parts (1) and (2).

The first is *boundary cancellation*, which essentially says that if you embed $\mathbf{e}$ into the source using $\mathcal{ST}$, and then embed that into the target using $\mathcal{TS}$, the resulting term is contextually equivalent to the original. Analogously, embedding $\mathbf{e}$ into the target via $\mathcal{TS}$ and then embedding the latter into the source via $\mathcal{ST}$, also results in a term that is contextually equivalent to the original.

Lemma 6.9 (Boundary Cancellation)

- *Let* $\Delta; \Gamma \vdash \mathbf{e} : \sigma$. *Then* $\Delta; \Gamma \vdash \mathbf{e} \approx^{log}_{\mathrm{ST}} {}^{\sigma}\mathcal{ST}\,(\mathcal{TS}\,{}^{\sigma}\,\mathbf{e}) : \sigma$.
- *Let* $\Delta; \Gamma \vdash \mathbf{e} : \sigma^+$. *Then* $\Delta; \Gamma \vdash \mathbf{e} \approx^{log}_{\mathrm{ST}} \mathcal{TS}\,{}^{\sigma}\,({}^{\sigma}\mathcal{ST}\,\mathbf{e}) : \sigma^+$.

Proof By induction on the structure of σ. □

The second key property is a *free theorem* [50] regarding terms of (computation) type $\forall\,[\alpha].(\sigma^+ \to \alpha) \to \alpha$. (We prove an analogous free theorem for terms of type $\forall\,[\alpha].(\sigma_1{}^+ \times (\sigma_2{}^+ \to \alpha)) \to \alpha$, but we will not show that here.) This free theorem captures the essence of what we gain from switching from a CPS type translation that makes use of a global answer type, to one that makes each continuation's answer type individually abstract: namely, that a computation (or function) of the above type *must* invoke its continuation at least once, and that it does not matter (in our purely functional setting) if it invokes it more than once. The free theorem can be proved using our logical relation; a similar theorem is given in Wadler [50]. We take a notational liberty in the statement of this theorem, which we discuss next.

Lemma 6.10 (Free Theorem: Continuation Shuffling)
Let $\Delta; \Gamma \vdash \mathbf{v}_f : \forall\,[\alpha].(\tau \to \alpha) \to \alpha$, *and* $\Delta; \Gamma \vdash \mathbf{v}_k : \tau \to \tau_k$.
Then $\Delta; \Gamma \vdash \mathbf{v}_f\ [\tau_k]\ \mathbf{v}_k \approx^{log}_{\mathrm{ST}} \mathbf{v}_k\ (\mathbf{v}_f\ [\tau]\ \mathbf{id}) : \tau_k$.

Notice that $\mathbf{v}_k\ (\mathbf{v}_f\ [\tau]\ \mathbf{id})$ is not a syntactically well-formed term in λ^{ST}! We use this essentially as shorthand to avoid a much longer (and less intuitive) statement of the theorem. Strictly speaking, the conclusion of the above lemma should be written as follows:

$$\begin{aligned}\text{Then: } \forall \rho \in \mathcal{D}\,[\![\Delta]\!]\,.\,\forall(\gamma_1,\gamma_2) \in \mathcal{G}\,[\![\Gamma]\!]\,\rho.\\ &\text{if } \rho_2(\gamma_2(\mathbf{v}_f\ [\tau]\ \mathbf{id})) \longmapsto^* \mathbf{v} \text{ then}\\ &\quad (\rho_1(\gamma_1(\mathbf{v}_f\ [\tau_k]\ \mathbf{v}_k)), (\rho_2(\gamma_2(\mathbf{v}_k)))\ \mathbf{v}) \in \mathcal{E}\,[\![\tau_k]\!]\,\rho\\ &\text{and}\\ &\text{if } \rho_1(\gamma_1(\mathbf{v}_f\ [\tau]\ \mathbf{id})) \longmapsto^* \mathbf{v} \text{ then}\\ &\quad ((\rho_1(\gamma_1(\mathbf{v}_k)))\ \mathbf{v}, \rho_2(\gamma_2(\mathbf{v}_f\ [\tau_k]\ \mathbf{v}_k))) \in \mathcal{E}\,[\![\tau_k]\!]\,\rho\end{aligned}$$

The intuition here is that we close off the expression $\mathbf{v}_f\ [\tau]\ \mathbf{id}$ with appropriate type and term substitutions and evaluate it to get a value $\mathbf{v}$ that we then pass to the (appropriately closed) continuation $\mathbf{v}_k$. The two clauses are required because our underlying relation $\mathcal{E}\,[\![\cdot]\!]$ is an approximation while what we want here is equivalence.

6.2 Proving Part (1)

We start with the top layer of the proof of equivalence preservation. Our goal here is to prove that if $\Gamma \vdash \mathbf{e}_1 \approx^{ctx}_{\mathrm{S}} \mathbf{e}_2 : \sigma$ then $\cdot; \Gamma \vdash \mathbf{e}_1 \approx^{ctx}_{\mathrm{ST}} \mathbf{e}_2 : \sigma$. As discussed in Section 2, given an arbitrary λ^{ST} context C with a hole of type $(\cdot; \Gamma \vdash \sigma)$, we need to "back-translate" C to an equivalent λ^{S} context C. The fact that this can be done at all may seem surprising, since λ^{ST} is a more expressive language than λ^{S}. Specifically, λ^{ST} includes System F in which we can encode, e.g., Church numerals, and operations like addition and multiplication; but natural numbers, addition and multiplication cannot be encoded in our source language λ^{S}. As another example, λ^{ST} contains pairs (from λ^{T}), but pairs cannot be encoded in our source language λ^{S}. Thus, we consider this "back-translation" method and its accompanying insights to be a significant contribution of this work.

"Back-translating" λ^{ST} to λ^{S} As discussed above, we wish to "back-translate" an arbitrary λ^{ST} context C with a hole of type $(\cdot; \Gamma \vdash \sigma)$. Such a context can simply be treated as an expression $\lambda \mathsf{x} : \sigma.\, C[\mathsf{x}]$ which has type $\sigma \to \mathsf{bool}$. This type is significant. Informally, the type represents the interface for this component and, in this case, it tells us that the component behaves like a term of source type (if we have our translation right, that is) and expects to interact with terms that behave similarly (which, technically, will mean terms of some source type σ or translation type σ^+). Thus, we have to be able to "back-translate" λ^{ST} terms of source type.

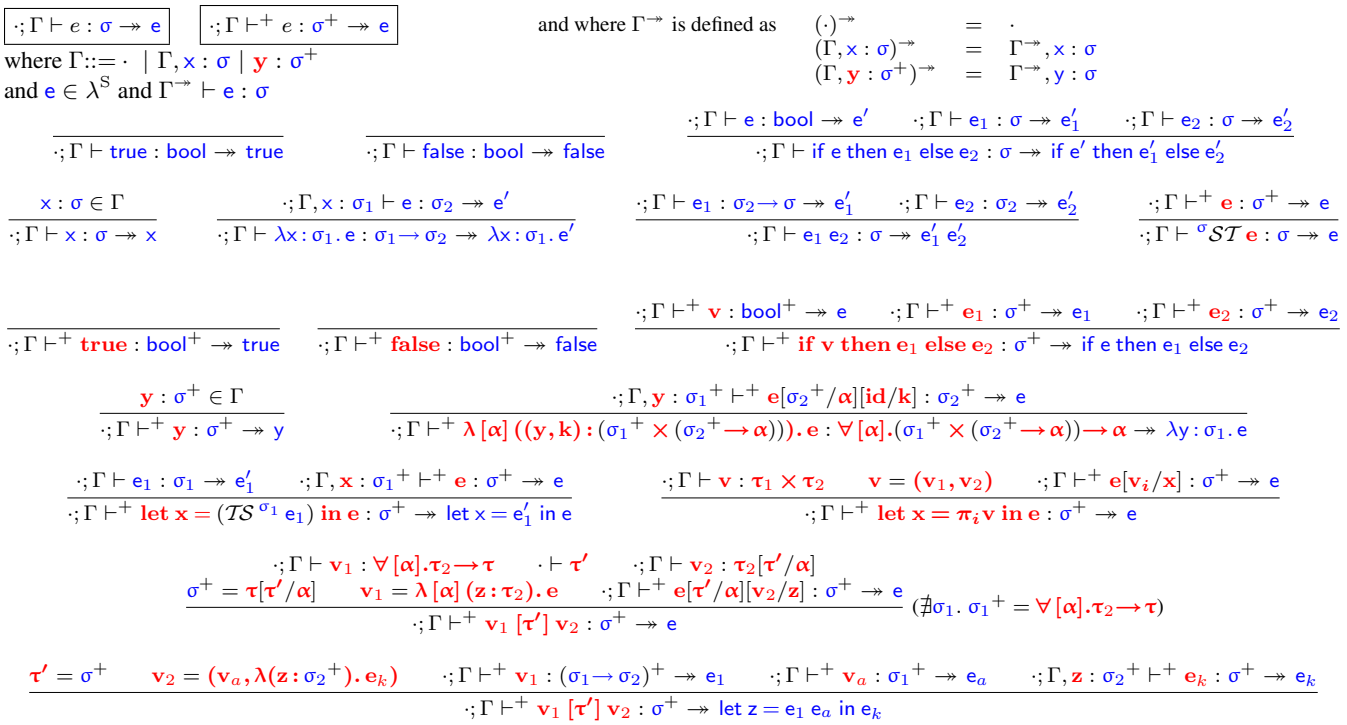

Figure 6. "Back-translation": Relating λ^{ST} terms to λ^{S} terms

Translating all the λ^{S} terms embedded in λ^{ST} is straightforward—they remain unchanged—until we get to a boundary term ${}^{\sigma}\mathcal{ST}\ \mathbf{e}$. At this point, we have to be able to translate $\mathbf{e}$ which has type σ^{+}. As discussed before, $\mathbf{e}$ may contain subterms that are not "back-translatable". However, the whole term $\mathbf{e}$ has type σ^{+} so it should be back-translatable. Intuitively, the idea here is to partially evaluate $\mathbf{e}$ till you have a term whose subterms are all back-translatable (of type σ or σ^{+}). Of course, the result of partial evaluation will be equivalent to the original $\mathbf{e}$. The remaining issue is: can we always partially evaluate till we get to such a point? We will show that this is always the case.

With that in mind, we set up two judgments that "back-translate" e to some $\mathsf{e} \in \lambda^{\mathrm{S}}$ (see Figure 6). They have the form $\cdot;\Gamma \vdash e : \sigma \twoheadrightarrow \mathsf{e}$ (for translating σ terms) and $\cdot;\Gamma \vdash^{+} e : \sigma^{+} \twoheadrightarrow \mathsf{e}$ (for translating σ^{+} terms). Here Γ may only contain mappings of the form $\mathsf{x} : \sigma$ or $\mathbf{y} : \sigma^{+}$—that is, Γ may *only* contain variables of source type or translation type. (This is an important restriction as we shall see.) $\Gamma^{\twoheadrightarrow}$ denotes the environment Γ with all mappings of the form $\mathbf{y} : \sigma^{+}$ replaced by $\mathsf{y} : \sigma$. $\Gamma^{\twoheadrightarrow}$ is the environment used to type-check e.

The "back-translation" rules are given in Figure 6. The rules for translating λ^{ST} terms $\mathsf{e} : \sigma$ are straightforward, defined by induction on the structure of the term. The only interesting case is the boundary ${}^{\sigma}\mathcal{ST}\ \mathbf{e}$. This is where we switch to the other judgment ($\vdash^{+}$) which translates terms $\mathbf{e}$ that have type σ^{+}. We translate target boolean values by converting them to equivalent source booleans and target $\mathbf{if}$ expressions are easy to translate because all of the subterms are of translation type.

When translating $\boldsymbol{\lambda}$ terms of type $\sigma_1 \to {\sigma_2}^{+}$, which have a type parameter $\boldsymbol{\alpha}$ and two value parameters $\mathbf{y} : {\sigma_1}^{+}$ and $\mathbf{k} : {\sigma_2}^{+} \to \boldsymbol{\alpha}$, we can only add $\mathbf{y} : {\sigma_1}^{+}$ to the environment Γ; since $\mathbf{k}$ is not of source type or translation type it cannot be added to Γ, and we can never add type variables to the type environment Δ (it always remains empty). Hence, we substitute ${\sigma_2}^{+}$ for $\boldsymbol{\alpha}$ in the body of the function and the identity continuation $\mathbf{id}$ for $\mathbf{k}$. As a result, our premise has a term of translation type ${\sigma_2}^{+}$ that we can continue to back-translate. The reason this rule is well founded is because the type of the term (${\sigma_2}^{+}$) in the premise is smaller than the type of the term in the conclusion.

Translating the (let form for) boundary $\mathcal{TS}^{\sigma}\ \mathsf{e}_1$ is straightforward, since the subterms have either source type or translation type.

The term $\mathbf{let\ x} = \boldsymbol{\pi}_i \mathbf{v}\ \mathbf{in\ e}$ is more interesting. Here $\mathbf{v}$ must have type $\tau_1 \times \tau_2$. But since the latter is not a source type σ or a translation type σ^{+}, $\mathbf{v}$ cannot be a variable! (This is why the restriction on the codomain of Γ is critical.) Therefore, $\mathbf{v}$ must be a pair, which means partial evaluation is possible. We project the i-th component and substitute it for $\mathbf{x}$ in the let body $\mathbf{e}$. The resulting term has type σ^{+} so we continue to back-translate. Note that this rule is well founded because $\mathbf{e}[\mathbf{v}_i/\mathbf{x}]$ will reduce in fewer steps than $\mathbf{let\ x} = \boldsymbol{\pi}_i \mathbf{v}\ \mathbf{in\ e}$.

The two rules for $\mathbf{v}_1\ [\tau']\ \mathbf{v}_2$ are more involved but follow similar reasoning. In the first of these rules the function $\mathbf{v}_1$ is not of translation type. That means that it cannot be a variable. This permits partial evaluation (by applying $\mathbf{v}_1$ to τ' and $\mathbf{v}_2$), which yields a term that can be back-translated (since it has type σ^{+}) and will reduce in fewer steps. In the second rule (last rule in Figure 6), $\mathbf{v}_1$ is of type $(\sigma_1 \to \sigma_2)^{+}$ (so it may be a variable), but that means that $\mathbf{v}_2$ must be a pair so it cannot be a variable. Therefore, we take apart the argument $\mathbf{v}_a$ and the continuation in that pair. The continuation also is not of translation type, so it cannot be variable. We then separately back-translate these and reassemble to get the final back-translation for $\mathbf{v}_1\ [\tau']\ \mathbf{v}_2$. This rule is well founded since it only requires back-translation of subterms of the original term.

Our rules are exhaustive, in the sense that all possible terms of type σ and σ^{+} have been covered.

Next we show that for every term of type σ and σ^{+}, it is possible to construct a finite back-translation derivation, and that the back-translation e is equivalent to the original e. Note that for

the equivalence statement, on the left-hand side we have to replace all target $\mathbf{y}$ variables with $\mathcal{TS}^{\sigma}\,\mathsf{y}$ since $\Gamma^{\twoheadrightarrow}$ contains $\mathsf{y} : \sigma$ in place of $\mathbf{y} : \sigma^{+}$.

Lemma 6.11 (From λ^{ST} term : σ / σ^{+} to equivalent λ^{S} term)
Let $\Gamma ::= \cdot \mid \Gamma, \mathsf{x} : \sigma \mid \mathbf{y} : \sigma^{+}$

1. *If* $\cdot; \Gamma \vdash e : \sigma$ *then there exists* $\mathsf{e} \in \lambda^{\mathrm{S}}$ *s.t.* $\cdot; \Gamma \vdash e : \sigma \twoheadrightarrow \mathsf{e}$ *and* $\cdot; \Gamma^{\twoheadrightarrow} \vdash e[(\overline{\mathcal{TS}^{\Gamma^{\twoheadrightarrow}(\mathsf{y})}\,\mathsf{y})/\mathbf{y}}] \approx^{log}_{\mathrm{ST}} \mathsf{e} : \sigma$.
2. *If* $\cdot; \Gamma \vdash e : \sigma^{+}$ *then there exists* $\mathsf{e} \in \lambda^{\mathrm{S}}$ *s.t.* $\cdot; \Gamma \vdash^{+} e : \sigma^{+} \twoheadrightarrow \mathsf{e}$ *and* $\cdot; \Gamma^{\twoheadrightarrow} \vdash {}^{\sigma}\mathcal{ST}\,(e[(\overline{\mathcal{TS}^{\Gamma^{\twoheadrightarrow}(\mathsf{y})}\,\mathsf{y})/\mathbf{y}}]) \approx^{log}_{\mathrm{ST}} \mathsf{e} : \sigma$.

Proof (1) and (2) are proved by simultaneous induction since the σ and σ^{+} translation rules are mutually dependent. We then proceed by induction on the length of the reduction sequence for e, nested induction on the type σ, and innermost induction on the structure of the term e. □

Wrapping Up Proof of Part (1) Our desired lemma, that $\Gamma \vdash \mathsf{e}_1 \approx^{ctx}_{\mathrm{S}} \mathsf{e}_2 : \sigma$ implies $\cdot; \Gamma \vdash \mathsf{e}_1 \approx^{ctx}_{\mathrm{ST}} \mathsf{e}_2 : \sigma$, follows as a corollary from the lemma below.

Lemma 6.12 (Ciu-equiv in λ^{S} implies ciu-equiv in λ^{ST})
Let Γ *be a* λ^{S} *environment, and let* e_1 *and* e_2 *be* λ^{S} *terms.*
If $\Gamma \vdash \mathsf{e}_1 \precsim^{ciu}_{\mathrm{S}} \mathsf{e}_2 : \sigma$ *then* $\cdot; \Gamma \vdash \mathsf{e}_1 \precsim^{ciu}_{\mathrm{ST}} \mathsf{e}_2 : \sigma$.

Proof Suppose $E : (\cdot; \cdot \vdash \sigma) \Rightarrow (\cdot; \cdot \vdash \mathsf{bool})$, and $\gamma_{\mathrm{st}} : \Gamma$ and $E[\gamma_{\mathrm{st}}(\mathsf{e}_1)] \Downarrow \mathsf{v}$ where $\mathsf{v} : \mathsf{bool}$. Show: $E[\gamma_{\mathrm{st}}(\mathsf{e}_2)] \Downarrow \mathsf{v}$.
We back-translate E (or, to be precise, $\lambda \mathsf{x} : \sigma.\, E[\mathsf{x}]$) and γ_{st} to e_E and γ_{s}. By Lemma 6.11 these are equivalent to the original E and γ_{st}. Hence, $E[\gamma_{\mathrm{st}}(\mathsf{e}_1)] \approx^{ctx}_{\mathrm{ST}} \mathsf{e}_E(\gamma_{\mathrm{s}}(\mathsf{e}_1)) : \mathsf{bool}$. Hence, the latter evaluates to v. Now, we instantate the premise with e_E (after morphing it into a valid evaluation context), and γ_{s}. Hence, $\mathsf{e}_E(\gamma_{\mathrm{s}}(\mathsf{e}_2))$ evaluates to v. Since $\mathsf{e}_E(\gamma_{\mathrm{s}}(\mathsf{e}_2)) \approx^{ctx}_{\mathrm{ST}} E[\gamma_{\mathrm{st}}(\mathsf{e}_2)] : \mathsf{bool}$, the latter evaluates to v. □

6.3 Proving Part (2)

We now tackle the middle layer of the proof of equivalence preservation. Our goal in this section is to prove that if e_1 and e_2 translate to $\mathbf{v}_1$ and $\mathbf{v}_2$, then $\mathsf{e}_1 \approx^{ctx}_{\mathrm{ST}} \mathsf{e}_2$ at type σ implies that $\mathbf{v}_1 \approx^{ctx}_{\mathrm{ST}} \mathbf{v}_2$ at type $\sigma^{\div}$.

Below, we prove that our CPS translation is both semantics preserving (typically referred to as compiler correctness) and semantics reflecting. We use the logical relation for our multi-language system to do these proofs. Our statements of semantics preservation and reflection are novel in that they rely on our multi-language semantics. As we discuss in Section 8, as compared to work on semantics-preserving compilation that uses cross-language logical relations [8, 9, 17] (which relate source terms to target terms), it seems simpler to understand how to set up the definitions and machinery for such proofs using a multi-language system and logical relation.

Finally, we wrap up by showing the main result of this section (Lemma 6.17), namely the proof that translation preserves equivalence (within λ^{ST}), which follows easily from the fact that CPS translation preserves and reflects semantics.

Translation Preserves and Reflects Semantics With the boundary cancellation and continuation-shuffling lemmas in hand, we can prove that our CPS translation preserves and reflects semantics. In the statement of these lemmas, we will have a λ^{S} term e on one side and its translation, a λ^{T} term $\mathbf{v}$, on the other side. Wherever e contains a variable $\mathsf{x} : \sigma$, $\mathbf{v}$ will have the variable $\mathbf{x} : \sigma^{+}$. Therefore, we will need related (source and target) substitutions γ_{S} and γ_{T} to obtain closed terms.

Definition 6.13 (Related Source-Target Substitutions)
Let Γ *be a finite map from variables* x *to types* σ. *Let* γ_{S} *be a finite map from variables* x *to (closed) values* v. *Let* γ_{T} *be a finite map from variables* $\mathbf{x}$ *to (closed) values* $\mathbf{v}$.
We define $\Gamma \vdash \gamma_{\mathrm{S}} \lesssim \gamma_{\mathrm{T}}$ *as follows:*

$$\cdot \vdash \emptyset \lesssim \emptyset \quad \text{iff} \quad \textit{(unconditionally)}$$
$$\Gamma, \mathsf{x} : \sigma \vdash \gamma_{\mathrm{S}}, \mathsf{x} \mapsto \mathsf{v} \lesssim \gamma_{\mathrm{T}}, \mathbf{x} \mapsto \mathbf{v} \quad \text{iff} \quad \Gamma \vdash \gamma_{\mathrm{S}} \lesssim \gamma_{\mathrm{T}} \;\wedge\; \vdash \mathsf{v} \precsim^{log}_{\mathrm{ST}} {}^{\sigma}\mathcal{ST}\,\mathbf{v} : \sigma$$

We define $\Gamma \vdash \gamma_{\mathrm{S}} \gtrsim \gamma_{\mathrm{T}}$ *as follows:*

$$\cdot \vdash \emptyset \gtrsim \emptyset \quad \text{iff} \quad \textit{(unconditionally)}$$
$$\Gamma, \mathsf{x} : \sigma \vdash \gamma_{\mathrm{S}}, \mathsf{x} \mapsto \mathsf{v} \gtrsim \gamma_{\mathrm{T}}, \mathbf{x} \mapsto \mathbf{v} \quad \text{iff} \quad \Gamma \vdash \gamma_{\mathrm{S}} \gtrsim \gamma_{\mathrm{T}} \;\wedge\; \vdash {}^{\sigma}\mathcal{ST}\,\mathbf{v} \precsim^{log}_{\mathrm{ST}} \mathsf{v} : \sigma$$

We define $\Gamma \vdash \gamma_{\mathrm{S}} \simeq \gamma_{\mathrm{T}}$ *as follows:*

$$\Gamma \vdash \gamma_{\mathrm{S}} \simeq \gamma_{\mathrm{T}} \quad \text{iff} \quad \Gamma \vdash \gamma_{\mathrm{S}} \lesssim \gamma_{\mathrm{T}} \wedge \Gamma \vdash \gamma_{\mathrm{S}} \gtrsim \gamma_{\mathrm{T}}$$

Notice that in the definition of $\Gamma \vdash \gamma_{\mathrm{S}} \gtrsim \gamma_{\mathrm{T}}$, we require $\vdash {}^{\sigma}\mathcal{ST}\,\mathbf{v} \precsim^{log}_{\mathrm{ST}} \mathsf{v} : \sigma$. Using boundary cancellation and the compatibility lemmas for boundaries, we can conclude that this is equivalent to $\vdash \mathbf{v} \precsim^{log}_{\mathrm{ST}} \mathcal{TS}^{\sigma}\,\mathsf{v} : \sigma^{+}$. (This observation might make it slightly easier to understand the statement of Lemma 6.15.)

Informally, the following lemma says that if e evaluates to some value v_1, then its CPS translation, when applied to the identity continuation will evaluate to some $\mathbf{v}_2 : \sigma^{+}$ that can be converted to a source value v_2 such that v_1 and v_2 are related at σ.

Lemma 6.14 (CPS is semantics preserving)
If $\Gamma \vdash \mathsf{e} : \sigma \rightsquigarrow \mathbf{v}$ *and* $\Gamma \vdash \gamma_{\mathrm{S}} \lesssim \gamma_{\mathrm{T}}$
then $\vdash \gamma_{\mathrm{S}}(\mathsf{e}) \precsim^{log}_{\mathrm{ST}} {}^{\sigma}\mathcal{ST}\,(\gamma_{\mathrm{T}}\,(\mathbf{v})\;\,[\sigma^{+}]\;\mathbf{id}) : \sigma$.

Proof By induction on the structure of e. □

The statement that CPS translation is semantics reflecting is a bit more involved. Informally, the following lemma says that suppose that the computation $\mathbf{v}$ (which is the CPS translation of e), when applied to a type $\boldsymbol{\tau}_\alpha$ and some continuation $\mathbf{v}_k : \sigma^{+} \to \boldsymbol{\tau}_\alpha$, evaluates to the value $\mathbf{v}_1 : \boldsymbol{\tau}_\alpha$. Then if we convert e to a target value $\mathbf{v}' : \sigma^{+}$, and then invoke the continuation $\mathbf{v}_k$ (or to be precise, a continuation that's related to $\mathbf{v}_k$) with $\mathbf{v}'$, this will result in a value $\mathbf{v}_2$ such that $\mathbf{v}_1$ and $\mathbf{v}_2$ are related at the type $\boldsymbol{\tau}_\alpha$.

Lemma 6.15 (CPS is semantics reflecting)
If $\Gamma \vdash \mathsf{e} : \sigma \rightsquigarrow \mathbf{v}$ *and* $\Gamma \vdash \gamma_{\mathrm{S}} \gtrsim \gamma_{\mathrm{T}}$ *then*
$\vdash \gamma_{\mathrm{T}}(\mathbf{v}) \precsim^{log}_{\mathrm{ST}} \boldsymbol{\lambda}\,[\boldsymbol{\alpha}]\,(\mathbf{k} : \sigma^{+} \to \boldsymbol{\alpha}).\ \mathbf{let}\ \mathbf{z} = (\mathcal{TS}^{\sigma}\,\gamma_{\mathrm{S}}(\mathsf{e}))\ \mathbf{in}\ \mathbf{k}\,\mathbf{z} : \sigma^{\div}$.

Proof By induction on the structure of e. □

The following is a corollary of semantics preservation and reflection. Notice that by boundary cancellation, the conclusion is equivalent to $\vdash \mathcal{TS}^{\sigma}\,(\gamma_{\mathrm{S}}(\mathsf{e})) \approx^{log}_{\mathrm{ST}} (\gamma_{\mathrm{T}}\,(\mathbf{v})\;\,[\sigma^{+}]\;\mathbf{id}) : \sigma^{+}$. (This explains why we call it "translation is equivalent to embedding.")

Corollary 6.16 (Translation is Equivalent to Embedding)
Let $\Gamma \vdash \mathsf{e} : \sigma \rightsquigarrow \mathbf{v}$ *and* $\Gamma \vdash \gamma_{\mathrm{S}} \simeq \gamma_{\mathrm{T}}$.
Then $\vdash \gamma_{\mathrm{S}}(\mathsf{e}) \approx^{log}_{\mathrm{ST}} {}^{\sigma}\mathcal{ST}\,(\gamma_{\mathrm{T}}\,(\mathbf{v})\;\,[\sigma^{+}]\;\mathbf{id}) : \sigma$.

Lemma 6.17 (Translation Preserves Equivalence in λ^{ST})
Let e_1 *and* e_2 *be* λ^{S} *terms. If* $\Gamma \vdash \mathsf{e}_1 : \sigma \rightsquigarrow \mathbf{v}_1$, $\Gamma \vdash \mathsf{e}_2 : \sigma \rightsquigarrow \mathbf{v}_2$, *and* $\cdot; \Gamma \vdash \mathsf{e}_1 \precsim^{log}_{\mathrm{ST}} \mathsf{e}_2 : \sigma$, *then* $\cdot; \Gamma \vdash \mathbf{v}_1 \precsim^{log}_{\mathrm{ST}} \mathbf{v}_2 : \sigma^{\div}$.

Proof Follows from Lemmas 6.14 and 6.15 and the transitivity of $\precsim^{log}_{\mathrm{ST}}$. □

6.4 Proving Part (3)

For the final (bottom) layer of our proof of equivalence preservation, we must show that if $\cdot; \Gamma^{+} \vdash \mathbf{v}_1 \approx^{ctx}_{\mathrm{ST}} \mathbf{v}_2 : \sigma^{\div}$ then $\cdot; \Gamma^{+} \vdash \mathbf{v}_1 \approx^{ctx}_{\mathrm{T}} \mathbf{v}_2 : \sigma^{\div}$. The latter is immediate from the following more general lemma.

Lemma 6.18 (Equivalence in λ^{ST} implies equivalence in λ^{T})
Let $\mathbf{e}_1$ *and* $\mathbf{e}_2$ *be* λ^{T} *terms.*
If $\Delta; \Gamma \vdash \mathbf{e}_1 \precsim^{ctx}_{\mathrm{ST}} \mathbf{e}_2 : \boldsymbol{\tau}$ *then* $\Delta; \Gamma \vdash \mathbf{e}_1 \precsim^{ctx}_{\mathrm{T}} \mathbf{e}_2 : \boldsymbol{\tau}$.

The proof is straightforward, intuitively, because λ^{T} contexts are a subset of λ^{ST} contexts. Given an arbitrary λ^{T} context $\mathbf{C}$ of the appropriate type, we must show that if $\mathbf{C}[\mathsf{e}_1]$ evaluates to $\mathbf{v}$ (which will be of type $\mathbf{bool}$) then so does $\mathbf{C}[\mathsf{e}_2]$. We can instantiate the premise with the context ${}^{\mathsf{bool}}\mathcal{ST}\,[\mathbf{C}[\cdot]]$. The rest easily follows from the fact that there is a one-to-one correspondence between the evaluation of a λ^{T} expression in λ^{ST} and in λ^{T}, and from noting that the reduction rule for ${}^{\mathsf{bool}}\mathcal{ST}\cdot$ simply converts $\mathbf{true}$ to true and $\mathbf{false}$ to false.

7. Equivalence Reflection

Equivalence reflection is a direct consequence of semantics preservation. Semantics preservation states that source programs (closed λ^{S} terms of base type, i.e., bool) and their translations in λ^{T} behave analogously:

Lemma 7.1 (Semantics preservation)
Let $\cdot \vdash \mathsf{e} : \mathsf{bool} \rightsquigarrow \mathbf{v}$. *If* $\mathsf{e} \longmapsto^* \mathsf{true}$ *then* $\mathbf{v}\ [\mathbf{bool}]\ \mathbf{id} \longmapsto^* \mathbf{true}$ *and if* $\mathsf{e} \longmapsto^* \mathsf{false}$ *then* $\mathbf{v}\ [\mathbf{bool}]\ \mathbf{id} \longmapsto^* \mathbf{false}$.

Proof Immediate from Lemma 6.16. □

The next observation we need concerns the structural behavior of the CPS translation:

Lemma 7.2 (Context translation)
Let $\Gamma \vdash \mathsf{C}[\mathsf{e}_1] : \sigma$ *and* $\Gamma \vdash \mathsf{C}[\mathsf{e}_2] : \sigma$. *Then there exist* $\mathbf{C}$, $\mathbf{v}_1$, $\mathbf{v}_2$ *such that* $\Gamma \vdash \mathsf{C}[\mathsf{e}_1] : \sigma \rightsquigarrow \mathbf{C}[\mathbf{v}_1]$ *and* $\Gamma \vdash \mathsf{C}[\mathsf{e}_2] : \sigma \rightsquigarrow \mathbf{C}[\mathbf{v}_2]$.

Proof By induction on the structure of C, using the definition of the CPS translation relation. □

Equivalence reflection now follows almost immediately from Lemmas 7.1 and 7.2:

Theorem 7.3 (Equivalence reflection)
Let $\Gamma \vdash \mathsf{e}_1 : \sigma \rightsquigarrow \mathbf{v}_1$ *and* $\Gamma \vdash \mathsf{e}_2 : \sigma \rightsquigarrow \mathbf{v}_2$. *If* $\cdot; \Gamma^+ \vdash \mathbf{v}_1 \approx^{ctx}_{\mathrm{T}} \mathbf{v}_2 : \sigma^\div$ *then* $\Gamma \vdash \mathsf{e}_1 \approx^{ctx}_{\mathrm{S}} \mathsf{e}_2 : \sigma$.

Proof Indirect: Suppose the conclusion does not hold, which means there exists some C such that $\cdot \vdash \mathsf{C}[\mathsf{e}_1] : \mathsf{bool}$ and $\cdot \vdash \mathsf{C}[\mathsf{e}_2] : \mathsf{bool}$ where (w.l.o.g.) $\mathsf{C}[\mathsf{e}_1] \longmapsto^* \mathsf{true}$ while $\mathsf{C}[\mathsf{e}_2] \longmapsto^* \mathsf{false}$. By Lemma 7.2 there must exist a $\mathbf{C}$ such that $\cdot \vdash \mathsf{C}[\mathsf{e}_1] : \mathsf{bool} \rightsquigarrow \mathbf{C}[\mathbf{v}_1]$ and $\cdot \vdash \mathsf{C}[\mathsf{e}_2] : \mathsf{bool} \rightsquigarrow \mathbf{C}[\mathbf{v}_2]$. At this point Lemma 7.1 tells us that $(\mathbf{C}[\mathbf{v}_1])\ [\mathbf{bool}]\ \mathbf{id} \longmapsto^* \mathbf{true}$ while $(\mathbf{C}[\mathbf{v}_2])\ [\mathbf{bool}]\ \mathbf{id} \longmapsto^* \mathbf{false}$. Thus, $(\mathbf{C}[\cdot])\ [\mathbf{bool}]\ \mathbf{id}$ is a context that discriminates between $\mathbf{v}_1$ and $\mathbf{v}_2$, which is a contradiction. □

8. Discussion and Future Work

Supporting Additional Language Features For this paper, we chose simply-typed λ-calculus and System F as our source and target languages so that we could highlight the main ideas underlying our proof technique, in particular, the use of multi-language semantics and how back-translation can leverage partial evaluation given the right type translation. We now sketch how to extend our theorem and its proof to source and target langauges with more advanced features.

If the source and target language have *non-terminating* terms, then we are faced with two difficulties: ensuring well-foundedness of the back-translation (which affects Lemma 6.11) and proving the continuation-shuffling lemma (Lemma 6.10).

To address the first problem, ensuring well-foundedness of the back-translation relation, we add a new case wherever back-translation performs computation steps: if the term to be partially evaluated is non-terminating (i.e., contextually equivalent to a term that diverges), simply make its translation a non-terminating source term of appropriate type. Notice that this definition does not yield an algorithm for back-translation but merely a relation. However, since back-translation is merely a proof device, it does not need to be an algorithm. We explain this is greater detail in the technical appendix [5] (see §1.1): we add divergent terms to both λ^{S} and λ^{T} and present the changes required to the back-translation rules.

The second problem, proving the continuation-shuffling lemma, is more difficult to deal with: in the presence of non-termination, Lemma 6.10 cannot be proved by parametricity alone; parametricity only gives us an approximation relation and not an equivalence relation. However, there is an alternative, syntactic proof for the continuation-shuffling lemma in this case. The details are beyond the scope of the current work; we will present that result in a future paper.

Recursive types give rise to non-termination, so everything said above applies here as well. In particular, though, recursive types make it somewhat trickier to define the semantics of our multi-language boundary terms. However, the technique for defining "wrapper" terms in our full abstraction proof for closure conversion [4] can be adapted to deal with this issue. Also, in the presence of recursive types we will need to switch to a step-indexed logical relation [3] for λ^{ST} to ensure well-foundedness of the logical relation. Thus, all parts of the proof that make use of the λ^{ST} logical relation will become more involved. Again, we plan to report on this result in a future paper.

Adding *polymorphism* to the source language is not difficult to deal with. The main idea is for boundary terms to not "peek under" abstraction barriers but to simply leave abstract things abstract. We have successfully applied this idea before in our work on fully abstract closure conversion [4] and do not foresee any difficulties when adapting it to the case of CPS translation.

Our longer-term goal is to show that a compiler from *System F* (with recursion, and later, with state) *to typed assembly language* is equivalence preserving. Thus far we have considered two phases of compilation, namely CPS (this paper) and typed closure conversion [4], where the type translations were precisely the key to ensuring that the target terms that compiled code interacts with are sufficiently well behaved. Thus, the type translations were the key to proving that the translations were equivalence preserving. To formulate an equivalence-preserving translation from CPS-and-closure-converted code down to assembly, we will need a target-level type system rich enough to be able to express invariants about local state, separation of state, and so on. We believe that an assembly language based on Hoare Type Theory [37] can provide the features needed for imposing the necessary well-behavedness constraints on target contexts.

Finally, the reader may have wondered if we can prove that our CPS translation is fully abstract when the target language is System F *without the syntactic restriction that enforces continuation-passing style*. In the online technical appendix (see §1.3) we show that this is, in fact, the case. The main change to the proof is that the back-translation from this unrestricted System F to STLC becomes more complicated, intuitively, since a target language without the CPS restriction contains many more terms than λ^{T}.

Compiler Correctness Proofs Let us compare the statement of "CPS is Semantics Preserving" (Lemma 6.14) to roughly what one would expect in a semantics-preservation (compiler correctness) proof that uses a cross-language logical relation (e.g., [8, 9, 17]) relating source terms of type σ to target terms of type σ^+ (though, relating to $\sigma^\div$ is a valid choice as well). In the absence of a combined language, the statement there cannot mention $\mathcal{ST}$. Instead it says that v_1 must be related to $\mathbf{v}_1$ in the cross-language relation at the type σ (which would mean that $\mathbf{v}_1$ would have type σ^+).

That means that to understand the definition of "relatedness" at each type, one would have to have a precise understanding of the cross-language relation. In the presence of features like recursive types (and eventually state), these logical relations get rather complicated, for instance using step-indexing and/or biorthogonal-

ity (e.g., [8, 9]), or advanced denotational models. Thus, it can be difficult for someone who simply wants to understand the statement of the semantics-preservation theorem, and not the mathematical machinery underlying the logical relation, to decipher exactly what that statement says. On the other hand, with the multi-language approach, to understand the statement of the theorem, one simply has to examine the reduction rules for boundaries. The details of the λ^{ST} logical relation are not important at this level; all one needs to know is that it is sound with respect to contextual equivalence.

Finally, defining the multi-language logical relation seems more straightforward than the cross-language logical relation. The multi-language logical relation is simply a combination of the logical relations for source and target; the desired relationship between source and target semantics is specified separately by defining the dynamic semantics for boundaries. The cross-language logical relation, meanwhile, must simultaneously provide a proof method and specify the desired relationship between source and target terms.

9. Related Work

Fully Abstract Denotational Models The earliest work on full abstraction was done in the context of denotational models of languages (e.g., [2, 16, 25, 33]). The denotation function can be thought of as a translation from a syntactic/operational calculus into a mathematical domain. The goal there is to ensure that the denotational semantics does not expose differences that are not operationally observable [35]. As typified by parallel OR in PCF [39], in a lot of this work, full abstraction is achieved by adding certain behaviors that are possible in the denotational semantics (target) to the operational semantics of the language being modeled (source). Our work differs somewhat from that on denotational models in that we are interested in proving full abstraction of *translations* (mostly compilers); here the target language also comes with an operational semantics. In particular, we focus on type-directed and type-preserving compilation and critically require a sufficiently "clever" type translation such that the types of compiled terms can impose well-behavedness constraints on any target-level term that might interact with the result of the translation, thus ensuring that target contexts cannot violate source-level abstractions.

Translation to Continuation-Passing Style CPS has been studied extensively in the literature. Here we only discuss work that seems most closely related to ours.

A number of papers have investigated full abstraction for CPS translation, but there is no prior full abstraction result for a CPS translation that uses a locally polymorphic answer type. Meyer and Riecke [32] pointed out that standard (untyped) CPS translation does not preserve observational equivalence and conjectured as to how this could be repaired.

Several papers have looked at the use of linear types to ensure that a function calls its continuation exactly once. Zdancewic and Myers [51] present a security-typed CPS target language with higher-order, imperative features where linear continuations are needed to ensure noninterference. They also give a translation from a security-typed, direct-style language into the afore-mentioned target language using linear typing of CPS and prove that the translation preserves well-typedness. Berdine *et al.* [10, 11] show that continuations are used linearly in a variety of situations, including procedure call/return, exception raising/handling, labelled jumps (`goto` statements) and process switching (coroutines). Neither Zdancewic and Myers [51] nor Berdine *et al.* [10, 11] present any full abstraction results for their CPS translation.

Berdine, O'Hearn, and Thielecke [12] use affine typing of CPS to extract the range of CPS for a call-by-value λ-calculus. Specifically, they restrict the grammar of types in the target to only those forms exercised by the CPS translation. They then define the grammar for target terms so that there is one syntactic category for terms of each type. This allows them to give a precise characterization of the range of CPS by showing that all terms in the target come from some term in the source—that is, they prove a "no junk" lemma (also known as full completeness) that states that for each term M in the target, there exists a term N in the source such that M is $\beta\eta$-equivalent to the CPS translation of N. The proof of this "no junk" lemma requires back-translating M to get N. The critical difference between Berdine *et al.*'s work [12] and ours that their target language is *exactly as expressive* as the source (i.e., every target term can be back-translated), while our target language is *more expressive* than the source (i.e., there exist well-typed target terms that cannot be back-translated, specifically, terms of type $\boldsymbol{\tau}$ where $\boldsymbol{\tau}$ is not a translation type.) Consequently, our proof framework allows for one target language to serve as the target for different source languages and compilers, and allows components written in these different source languages to interoperate at the target level (assuming the compilation strategies rely on similar representation invariants at the target level). As a concrete example, we could define a CPS translation from a second source language, System F, to our target language λ^{T}. Now source code written in λ^{S} can interoperate with source code written in System F *after* CPS translation to λ^{T}, as long as we have a well-typed program after "linking" the two compiled components in λ^{T}. This would not be possible if we had resorted to a strategy like Berdine *et al.*'s where back-translation requires that the target language be no more expressive than the source.

Sabry and Felleisen [43] study equational completeness of CPS based on $\beta\eta$-equality rather than observational equivalence. They present a CPS transformation and an inverse mapping (or "back-translation"). From the CPS transformation, they extract the precise language of CPS terms closed under $\beta\eta$-equality, arriving at almost exactly the same syntax (modulo "administrative" redexes) as Berdine *et al.* [12]. Sabry and Felleisen analyze the syntax of the output of CPS while Berdine *et al.* analyze the types of the output of CPS to arrive as almost exactly the same target language syntax. Hence, Berdine *et al.*'s back-translation is essentially Sabry and Felleisen's inverse mapping (from CPS to direct style).

Hasegawa [21] has proved a full completeness result for the linear CPS transformation in the setting of a simply typed λ-calculus using syntactic methods based on long $\beta\eta$-normal forms. Like Sabry and Felleisen [43], he considers only $\beta\eta$-equivalence, not observational equivalence.

Using categorical game semantics, Laird [29] showed that for call-by-value PCF one can recover full abstraction of CPS translation by imposing an *affine* typing discipline on continuations, essentially employing the idea of "linearly-used continuations" presented by Berdine *et al.* [11]. We feel that with proofs based on game semantics, it is hard for non-experts to understand even the *statements* of the main lemmas required for the proof. Therefore, a primary objective of our work has been to come up with an operational proof technique that's simpler to understand and could (plausibly) be used throughout all the stages of a compiler. The proof techniques described in this paper, combined with recent advances in step-indexed logical relations [3, 6] seem like they would scale when applied to richer languages and successive compilation phases.

Thielecke [48] seems to have been the first to study CPS transformation with a locally polymorphic answer type, though his work focuses on the role of answer type polymorphism in a language with control effects. He uses parametricity reasoning to observe the connection between linear typing of CPS and answer type polymorphism in a pure call-by-value setting, showing that functions without control effects do not impose any constraints on the answer type and so can have a locally polymorphic answer type. He essen-

tially proves the equivalent of our continuation-shuffling lemma, a property that he calls naturality. He has also studied answer type polymorphism for the call-by-name CPS transform and used it to show that the latter satisfies the eta-law [49]. He does not, however, discuss full abstraction of CPS translations.

Danvy [18] presents a translation from CPS programs into direct-style (DS) programs in an untyped setting. His technique relies on syntactically characterizing CPS terms that can be translated back to DS. Specifically, to be back-translatable, a CPS term must satisfy certain *occurrence conditions* (see Danvy [18], Fig. 2) that ensure that the CPS term encodes a call-by-value left-to-right evaluation order, checked essentially by parsing the CPS term using a stack that holds the formal parameters of continuations. We, on the other hand, use types to *semantically* characterize terms that can be translated back to our direct-style λ^{S}, without requiring that all of their subterms also have back-translatable types. As a result, we can back-translate more terms. Finally, our CPS grammar does not distinguish between ordinary λ's and continuation λ's as Danvy's does, while he does not make use of partial evaluation as we do, i.e., to deal with subterms that are not, on their own, back-translatable.

Several researchers have investigated back-and-forth translations between direct-style and CPS semantics following Meyer and Wand's [34] work on *retractions*. An embedding-retraction pair (i, j) is a pair of functions such that $j \circ i$ is the identity function. Meyer and Wand work with the typed λ-calculus and use the "standard" type translation for CPS that we showed is not fully abstract in Section 1. They write σ' to denote their type translation of σ; let $\sigma^* = (\sigma' \to \mathrm{ans}) \to \mathrm{ans}$. They show that there exist embedding-retraction pairs (i_σ, j_σ) and (I_σ, J_σ)—definable in call-by-name λ-calculus (CBN)—where $i_\sigma : \sigma \to \sigma'$, $j_\sigma : \sigma' \to \sigma$, $I_\sigma : \sigma' \to \sigma^*$, and $J_\sigma : \sigma^* \to \sigma'$. Their main result is the Retraction Theorem which says that $(j_\sigma \circ J_\sigma)$ is the inverse of the CPS transform, i.e., $M =_{CBN} j_\sigma(J_\sigma(\overline{M}))$, where $\overline{M}$ is the CPS transform of M. The boundary terms $\mathcal{TS}^{\sigma}$ and ${}^{\sigma}\mathcal{ST}$ in our multi-language semantics are similar in spirit to i_σ and j_σ, respectively; the property that $(j_\sigma \circ i_\sigma) = \mathsf{id}$ is analogous to (part 1) of our boundary cancellation property (Lemma 6.9); and their Retraction Theorem, which says that the retraction of a translation is equivalent to the original term, is analogous to our "translation is equivalent to embedding" lemma (Corollary 6.16). [1] But there are also important technical differences since our target language (System F) is more expressive than theirs (STLC), and due to the fact that we make use of a multi-language semantics. In particular, our boundary terms are "built-in" with an appropriate operational semantics; thus our embedding-retraction pairs do not have to be *definable* in the target language as is the case with retractions, and this makes our multi-language technique more general. With retractions, the source language is generally assumed to be a strict subset of the target and the Retraction Theorem proves equivalence with respect to the (larger) target language. Note that our λ^{S} is not a strict subset of λ^{T} since the latter syntactically enforces continuation-passing style. Meyer and Wand's Retraction Theorem is essentially analogous to our proof of Part (2). There is no analog to our Part (1) and no proof of full abstraction. In particular, Meyer and Riecke [32] subsequently showed that if we replace CBN $\beta\eta$-equational reasoning with call-by-value observational equivalence, then the embedding-retraction pairs defined in Meyer-Wand no longer suffice. In fact, Meyer and Riecke failed to prove a Retraction Theorem in this setting.

[1] In technical terms, our previous work on typed closure conversion seems closer to the work on retractions since there we do not use a multi-language semantics; instead the source and target are the same language and in this language we define wrapper functions $\mathcal{W}^+$ and $\mathcal{W}^-$ that are analogous to i and j, respectively, and we prove a theorem similar to Meyer and Wand's (except that it's for closure conversion, not CPS conversion).

Later Riecke [41] and Riecke and Viswanathan [41] investigated a semantic variation of the retraction approach with the goal of isolating side-effects in sequential programs. Also, Filinski [19] generalized Meyer and Wand's Retraction Theorem to a CPS transform for the monadic metalanguage. Like Meyer-Wand, Filinski's technique does not generalize to a language with divergence.

Berger, Honda, and Yoshida have studied fully abstract translations from various languages (with recursion [13], polymorphism [14, 15], control [23], state [22], and concurrency [22]) to linear or affinely typed—and in some cases polymorphic [14, 15]—π-calculus. They prove their translations fully abstract in the case of recursion (with source language PCF) [13], polymorphism (with source language System F) [14, 15], and control [23], but only speculate about full abstraction in the case of state and concurrency [22]. Since the usual translation of the λ-calculus into the π-calculus can be seen as a form of CPS translation, it may be useful to further investigate the connections between translations into the π-calculus and translations to continuation-passing style. Like us, Berger *et al.* rely on typing in the π-calculus to ensure that the translations are fully abstract. Unlike us, they rely on game semantics to prove their translations fully abstract. In this paper, we have shown that terms of translation type are *back-translatable*. Analogously, Berger *et al.* show that terms of translation type are *definable*. Definability says that for every π-calculus term P of translation type $\sigma^\bullet$, there exists a well-typed source term $M : \sigma$ (where they write $\sigma^\bullet$ to denote the translation of a source type σ). Like us, they note that one reason definability is difficult to establish is because subterms of P may not be of translation type, which means that the proof cannot be carried out simply by induction on typing derivations. Our strategy for dealing with subterms that are not of translation type was to show that it is always possible to perform some partial evaluation that gets rid of the problematic subterm, leaving only subterms that *are* of translation type. Their strategy is to show that every finite target term P of translation type can be represented by a *finite innocent function* that can be turned into a *finite canonical form*, which in turn is easily transformed into some source term M such that P is equivalent to the translation of M. Thus, they use the notion of *innocence* [24] from game semantics to establish, in essence, that translation types at the target level are inhabited by only well-behaved computations. They are, thus, able to perform induction on the size of the corresponding innocent functions. This approach is similar to that of Laird's [29] whose proof of full abstraction also relies on game semantics. Our proof method is more elementary as it relies on operational/syntactic techniques (coupled with typing) for back-translatability; expertise in game semantics is not required to follow the details.

Full Abstraction of Other Translations Most work on proving that translations preserve equivalence has typically resorted to adding precisely those target behaviors that are problematic to the source language. For instance, Riecke [42] investigates fully abstract translations between CBN, CBV, and lazy PCF, using denotational models of the languages that include the parallel conditional. This is needed to make the models fully abstract. Also, Sanjabi and Ong [44] investigate a translation from a core calculus of additive aspects to a target language with higher-order store in the style of ML references. After showing that their original translation is not fully abstract, they weaken the source language by endowing it with the power to construct "bad labels"—the analogue of the bad references at the target that were responsible for the failure of full abstraction.

Shikuma and Igarashi [46] prove full abstraction of a translation from STLC with seal and unseal operators to STLC with base types for each sealing authority. They use a syntactic proof method, but their back-translation is only applicable to terms *all* of whose *sub-*

terms are of translation type. Our back-translation is more general precisely because it does not impose this restriction.

Acknowledgments

The first author would like to thank Greg Morrisett for suggesting the problem of fully abstract compilation to her in Spring 2005. We thank Kyle Ross who helped us with this work in Spring 2010. We are also grateful to Amr Sabry and several anonymous reviewers for their many helpful suggestions on earlier versions of this paper.

References

[1] M. Abadi. Protection in programming-language translations. In *ICALP*, 1998.

[2] S. Abramsky, R. Jagadeesan, and P. Malacaria. Full abstraction for PCF. *Inf. Comput.*, 163(2):409–470, 2000.

[3] A. Ahmed. Step-indexed syntactic logical relations for recursive and quantified types. In *ESOP*, 2006.

[4] A. Ahmed and M. Blume. Typed closure conversion preserves observational equivalence. In *ICFP*, 2008.

[5] A. Ahmed and M. Blume. An equivalence-preserving CPS translation via multi-language semantics (technical appendix). Available at `http://www.cs.indiana.edu/~amal/papers/epc/`, July 2011.

[6] A. Ahmed, D. Dreyer, and A. Rossberg. State-dependent representation independence. In *POPL*, 2009.

[7] A. W. Appel. *Compiling with Continuations*. Cambridge University Press, 1992.

[8] N. Benton and C.-K. Hur. Biorthogonality, step-indexing and compiler correctness. In *ICFP*, 2009.

[9] N. Benton and C.-K. Hur. Realizability and compositional compiler correctness for a polymorphic language. Technical Report MSR-TR-2010-62, Microsoft Research, Apr. 2010.

[10] J. Berdine. Linear and affine typing of continuation-passing style. Technical Report RR-04-04, Queen Mary, Univ. of London, Jan. 2004.

[11] J. Berdine, P. O'Hearn, U. Reddy, and H. Thielecke. Linear continuation-passing. *Higher Order Symbol. Comput.*, 15(2-3):181–208, 2002.

[12] J. Berdine, P. O'Hearn, and H. Thielecke. Extracting the range of cps from affine typing: Extended abstract. In *Workshop on Linear Logic*, 2002.

[13] M. Berger, K. Honda, and N. Yoshida. Sequentiality and the π-calculus. In *TLCA*, 2001.

[14] M. Berger, K. Honda, and N. Yoshida. Genericity and the π-calculus. In *FOSSACS*, 2003.

[15] M. Berger, K. Honda, and N. Yoshida. Genericity and the π-calculus. *Acta Informatica*, 42:83–141, November 2005.

[16] R. Cartwright and M. Felleisen. Observable sequentiality and full abstraction. In *POPL*, 1992.

[17] A. Chlipala. A certified type-preserving compiler from lambda calculus to assembly language. In *PLDI*, 2007.

[18] O. Danvy. Back to direct style. *Science of Computer Programming*, 22(3):183–195, 1994.

[19] A. Filinski. Representing monads. In *POPL*, 1994.

[20] R. Harper and M. Lillibridge. Explicit polymorphism and CPS conversion. In *POPL*, 1993.

[21] M. Hasegawa. Linearly used effects: Monadic and CPS transformations into the linear lambda calculus. In *FLOPS*, 2002.

[22] K. Honda and N. Yoshida. A uniform type structure for secure information flow. In *POPL*, 2002.

[23] K. Honda, N. Yoshida, and M. Berger. Control in the π-calculus. In *Fourth ACM-SIGPLAN Continuations Workshop (CW '04)*, Jan. 2004.

[24] J. M. E. Hyland and C. H. L. Ong. On full abstraction for PCF: I, II, and III. *Information and Computation*, 163(2):285–408, 2000.

[25] A. Jeffrey. A fully abstract semantics for a concurrent functional language with monadic types. In *LICS*, 1995.

[26] A. Kennedy. Securing the .NET programming model. *Theoretical Computer Science*, 364(3):311–317, 2006.

[27] A. Kennedy. Compiling with continuations, continued. In *ICFP*, 2007.

[28] D. A. Kranz, R. A. Kelsey, J. A. Rees, P. Hudak, and J. Philbin. ORBIT: an optimizing compiler for Scheme. In *Proceedings of the ACM Symposium on Compiler Construction*, June 1986.

[29] J. Laird. Game semantics and linear CPS interpretation. *Theor. Comput. Sci.*, 333(1-2):199–224, 2005.

[30] I. A. Mason and C. L. Talcott. Equivalence in functional languages with effects. *J. Functional Programming*, 1(3):287–327, 1991.

[31] J. Matthews and R. B. Findler. Operational semantics for multi-language programs. In *POPL*, Nice, France, pages 3–10, Jan. 2007.

[32] A. Meyer and J. G. Riecke. Continuations may be unreasonable. In *Conf. on LISP and functional programming, LFP*, 1988.

[33] A. R. Meyer and K. Sieber. Towards fully abstract semantics for local variables. In *POPL*, 1988.

[34] A. R. Meyer and M. Wand. Continuation semantics in typed lambda-calculi. In R. Parikh, editor, *Logics of Programs (Brooklyn, June, 1985)*, volume 193 of *LNCS*, pages 219–224. Springer-Verlag, 1985.

[35] R. Milner. Fully abstract models of typed lambda calculi. *Theoretical Computer Science*, 4(1), 1977.

[36] G. Morrisett, D. Walker, K. Crary, and N. Glew. From System F to typed assembly language. *Transactions on Programming Languages and Systems*, 21(3):527–568, May 1999.

[37] A. Nanevski, A. Ahmed, G. Morrisett, and L. Birkedal. Abstract predicates and mutable adts in hoare type theory. In *ESOP*, 2007.

[38] A. M. Pitts. Typed operational reasoning. In B. C. Pierce, editor, *Advanced Topics in Types and Programming Languages*. MIT Press, 2005.

[39] G. D. Plotkin. LCF considered as a programming language. *Theoretical Computer Science*, 5:223–255, 1977.

[40] J. C. Reynolds. Types, abstraction, and parametric polymorphism. *Information Processing*, pages 513–523, 1983.

[41] J. Riecke and R. Viswanathan. Isolating side effects in sequential languages. In *POPL*, 1995.

[42] J. G. Riecke. Fully abstract translations between functional languages. In *POPL*, 1991.

[43] A. Sabry and M. Felleisen. Reasoning about programs in continuation-passing style. In *Conf. on LISP and functional programming, LFP*, 1992.

[44] S. B. Sanjabi and C.-H. L. Ong. Fully abstract semantics of additive aspects by translation. In *Proceedings of the 6th international conference on Aspect-oriented software development (AOSD)*, 2007.

[45] Z. Shao and A. W. Appel. A type-based compiler for Standard ML. In *PLDI*, 1995.

[46] N. Shikuma and A. Igarashi. Proving noninterference by a fully complete translation to the simply typed lambda-calculus. *Logical Methods in Computer Science*, 4(3:10):1–31, 2008.

[47] G. L. Steele. RABBIT: A compiler for SCHEME. Technical Report AI-TR-474, MIT, May 1978.

[48] H. Thielecke. From control effects to typed continuation passing. In *POPL*, 2003.

[49] H. Thielecke. Answer type polymorphism in call-by-name continuation passing. In *ESOP*, 2004.

[50] P. Wadler. Theorems for free! In *ACM Symposium on Functional Programming Languages and Computer Architecture (FPCA)*, 1989.

[51] S. Zdancewic and A. C. Myers. Secure information flow and CPS. In *ESOP*, 2001.