Fully Abstract Compilation via Universal Embedding

Max New
Northeastern University
maxnew@ccs.neu.edu

William J. Bowman
Northeastern University
wjb@williamjbowman.com

Amal Ahmed
Northeastern University
amal@ccs.neu.edu

Abstract

A *fully abstract* compiler guarantees that two source components are observationally equivalent in the source language if and only if their translations are observationally equivalent in the target. Full abstraction implies the translation is *secure*: target-language attackers can make no more observations of a compiled component than a source-language attacker interacting with the original source component. Proving full abstraction for realistic compilers is challenging because realistic target languages contain features (such as control effects) unavailable in the source, while proofs of full abstraction require showing that every target context to which a compiled component may be linked can be *back-translated* to a behaviorally equivalent source context.

We prove the first full abstraction result for a translation whose target language contains exceptions, but the source does not. Our translation—specifically, closure conversion of simply typed λ-calculus with recursive types—uses types at the target level to ensure that a compiled component is never linked with attackers that have more distinguishing power than source-level attackers. We present a new back-translation technique based on a deep embedding of the target language into the source language at a dynamic type. Then boundaries are inserted that mediate terms between the untyped embedding and the strongly-typed source. This technique allows back-translating non-terminating programs, target features that are untypeable in the source, and well-bracketed effects.

We use a blue sans-serif font to typeset our source language and a **bold red serif font** to typeset the target. The paper will be much easier to read if viewed/printed in color.

1. Introduction

When building secure software systems, programmers rely on language-provided abstractions and on the assumption that any attacker—i.e., any code that their software component might be linked with—will be bound by the rules of the programming language. However, after the component is compiled, it may be linked with arbitrary target-level attackers that violate source-level abstractions, thus invalidating source-level security guarantees. Target attackers may be able to do things impossible in the source, such as read the compiled component's private data, modify the component's control flow, and even modify code implementing the component's methods.

To guarantee that target attackers respect source-language rules, a compiler must be *fully abstract*—that is, it should *preserve* and *reflect* observational equivalence [1, 20, 6, 15, 28, 10, 13]. We use the standard notion of observational equivalence, also known as *contextual equivalence*: two components are contextually equivalent if they are indistinguishable in any valid (appropriately typed) program context. Fully abstract compilation ensures that when a source component e compiles to a target component **e** a valid target-language context **C** (attacker) does not have the power to observe anything more from interacting with **e** than a source-language context C interacting with e. Note that ensuring fully abstract compilation is only important when compiling *components* (not whole programs) since it is a property that ensures a component is protected from an attacker context (but whole programs have no context).

Achieving Secure Compilation There are three complementary techniques that one could use to achieve secure compilation. The first and most basic is to change the source program's notion of program equivalence to be exactly the equivalence of compiled code. In this way all compilers can be seen to be secure, but at the cost of introducing extra-linguistic reasoning. A good example of this is programming in low-level languages like C where one reasons about programs by understanding their compilation.

The second way is to introduce mediating constructs such as dynamic checks or encodings that coerce low-level attackers into behaving like high-level code. This comes at the cost of runtime overhead in time and space. This is the approach taken in much of the literature (e.g., [15, 3, 27, 28, 11]).

The final way is to statically ensure that low-level program interfaces are secure, i.e., low-level programs operating at the specified interface must be in some way verified to act like high-level programs. This comes at the cost of disallowing linking with unverified/untyped code.

In this paper we focus on the final approach. In particular, we use types to statically verify that the target-language contexts that a compiled component is linked with have no more observational power than source-language contexts. Of course, taking this statically-enforced secure compilation approach in a realistic compiler will require designing *statically sound* (e.g., typed) intermediate languages for the compiler, but note that eventually all type information will be erased before runtime. Since fully abstract compilation is only important when compiling components, as noted above, we only need to perform type-preserving compilation till the IR in the compiler where linking happens, yielding a whole program. Having type information available at link time is important in order to rule out linking with contexts that can attack the component in a way that no source context could have. However, after that point in the compilation pipeline, we can erase types and do whole-program compiler correctness—i.e., there is no further need for types or for proving full abstraction for the remaining passes of the compiler.

There are currently significant performance costs to existing dynamically-enforced secure compilers, and reducing these costs requires reliance on specialized hardware support—for instance, in the form of protected memory architectures (PMAs) as shown by Patrignani et al. [28]. Our work focuses on type-enforced secure compilation to ensure that dynamic checks *can* be avoided, without reliance on any specialized hardware, when low-level code *is* verified and to do this we need to ensure the compiler is secure when linking against a certain class of low-level programs.

Moreover, even if one were only to link against unverified low-level code, in order to minimize dynamic enforcement overhead we *should* use statically sound *intermediate* compiler passes. For example, recent work by Devriese et al. [11] showed full abstraction for a dynamically-protected type erasure pass for simply typed lambda calculus. The protection mechanism introduces deep checks to ensure safety for the compiler which would have to be compounded by further checks when compiling to lower-level languages. If multiple passes of the compiler are introducing deep semantic checks then the compiler will produce highly inefficient code. Thus, to minimize the cost of dynamic checks one should preserve high level information (e.g., via types) as long as possible in a secure compiler.

Correct and Secure Compilation We start with specifications of correctness and security of our compiler. We give these specifications in terms of relations that we explain intuitively here, and formally define later.

We write $\Gamma \vdash \mathtt{e} : \sigma \rightsquigarrow_e \mathbf{e}$ when a source component $\mathtt{e}$ of type σ (under environment Γ) compiles to the target component $\mathbf{e}$ of translation type $\sigma^{\div}$ (under the translated environment Γ^{+}). We use the relations $\approx_{\mathtt{S}}^{ctx}$ and $\approx_{\mathbf{T}}^{ctx}$ to denote source and target contextual equivalence, respectively.

Theorem 1.1 (Semantics Preservation)
If $\Gamma \vdash \mathtt{e} : \sigma \rightsquigarrow_e \mathbf{e}$, *then* $\Gamma \vdash \mathtt{e} \; {}^{\mathcal{S}}\!\simeq^{\mathcal{T}} \mathbf{e} : \sigma$.

Semantics preservation, Theorem 1.1, states that the translation of a source *component* correctly preserves the behavior of the original source component, for a suitable specification ${}^{\mathcal{S}}\!\simeq^{\mathcal{T}}$ of related behavior. We specify related behavior by a cross-language equivalence that can be seen as a specification for the calling convention and encoding of source values. In other words, it specifies when a source component $\mathtt{e} : \sigma$ is behaviorally equivalent to a target component $\mathbf{e} : \sigma^{\div}$ (where $\mathbf{e}$ may or may not be an output of the compiler). We define this cross-language equivalence using a multi-language contextual equivalence in §3. The multi-language specifies interoperability between the source and target languages of our compiler [6, 29]. Theorem 1.1 gives us compositional compiler correctness, *i.e.*, correctness for components instead of whole programs [6, 29, 37, 26, 39]; in the special case that $\Gamma = \cdot$, we recover whole-program correctness.

Theorem 1.2 (Equivalence Reflection)
If $\Gamma \vdash \mathtt{e}_1 : \sigma \rightsquigarrow_e \mathbf{e_1}$, $\Gamma \vdash \mathtt{e}_2 : \sigma \rightsquigarrow_e \mathbf{e_2}$ *and* $\cdot; \Gamma^{+} \vdash \mathbf{e_1} \approx_{\mathbf{T}}^{ctx} \mathbf{e_2} : \sigma^{\div}$, *then* $\Gamma \vdash \mathtt{e}_1 \approx_{\mathtt{S}}^{ctx} \mathtt{e}_2 : \sigma$.

Equivalence reflection, Theorem 1.2, states that if compiling two different source components results in two target components that are contextually equivalent in the target language, then the original source components must be contextually equivalent in the source language. This is perhaps clearer if viewed as its contrapositive: if two source components *are distinguishable*, then so are their translations. Viewed in this way it can be seen as a weak form of semantics preservation that can be stated independent of the specification ${}^{\mathcal{S}}\!\simeq^{\mathcal{T}}$. We prove this as a corollary of the stronger semantics preservation theorem in §4.

Theorem 1.3 (Equivalence Preservation)
If $\Gamma \vdash \mathtt{e}_1 : \sigma \rightsquigarrow_e \mathbf{e_1}$, $\Gamma \vdash \mathtt{e}_2 : \sigma \rightsquigarrow_e \mathbf{e_2}$ *and* $\Gamma \vdash \mathtt{e}_1 \approx_{\mathtt{S}}^{ctx} \mathtt{e}_2 : \sigma$, *then* $\cdot; \Gamma^{+} \vdash \mathbf{e_1} \approx_{\mathbf{T}}^{ctx} \mathbf{e_2} : \sigma^{\div}$.

Equivalence preservation, Theorem 1.3, ensures a *secure* compiler. It states that two observationally equivalent source components will be compiled to observationally equivalent target components. Viewed as a modularity property, this says that the abstraction barriers of the source language are preserved by compilation. Viewed as a security property, it says that attackers in the target language can be emulated by attackers in the source, so any reasoning about security can be done completely in the source language. Proving equivalence preservation requires back-translating target contexts to the source to show that they can make no more observations than source contexts. We prove this at the end of §5.

While equivalence preservation and reflection may at first glance seem to subsume the semantics-preservation theorem, note that neither mentions any notion of cross-language equivalence. Just proving full abstraction cannot guarantee that you use the correct calling convention or encoding. For instance, a compiler that compiled `true` to **`false`**, `false` to **`true`** and swapped branches of `if`s preserves and reflects equivalences, but would not be semantics-preserving if the specification says that `true` should be encoded as **`true`**. However, equivalence reflection tells you that semantics have been "morally preserved," in that distinctions between source programs were not lost.

Contributions The key to proving Theorem 1.3 is showing that for every well-behaved target context—i.e., context of translation type—there exists a behaviorally equivalent source context. This essentially requires *back-translating* a more expressive target language to a less expressive source language. Prior full abstraction results targeting typed languages performed *back-translation by partial evaluation* [6, 10, 35], a technique that does not seem to scale to nonterminating languages, which we discuss in §7.

We demonstrate a new back-translation technique, *universal embedding*. This technique is to first translate the target language into the source at a universal—i.e., dynamic—source type and then mediate between the universal type and each source type with functions that perform the encoding at run-time. Whereas previous work back-translates to types that precisely encode only target programs, we instead back-translate to a type that is overly large: it contains all the target behaviors but many other undesirable behaviors. This enables us to back translate when the target is capable of finer type distinctions than the source, exemplified in our paper by the polymorphism of the target language.

This universal embedding demonstrates equivalence between an embedded interpreter and a multi-language semantics, a novel result of independent interest.

We apply this technique to prove full abstraction of typed closure conversion from a simply-typed λ-calculus with recursive types to a polymorphic λ-calculus with recursive types and exceptions. Our technique allows back-translating non-terminating programs, target-language features that are untypeable in the source, and well-bracketed/delimited effects. Ours is the first full-abstraction result that targets a *typed*, *non-terminating* target language with features unavailable in the source.

We provide complete definitions and proofs in the technical report, submitted as supplementary material.

2. Closure Conversion

Our source and target languages are both call-by-value. They are also in monadic normal form—constructors and eliminators are only applied to syntactic values [9]—meant to represent compiler intermediate languages.

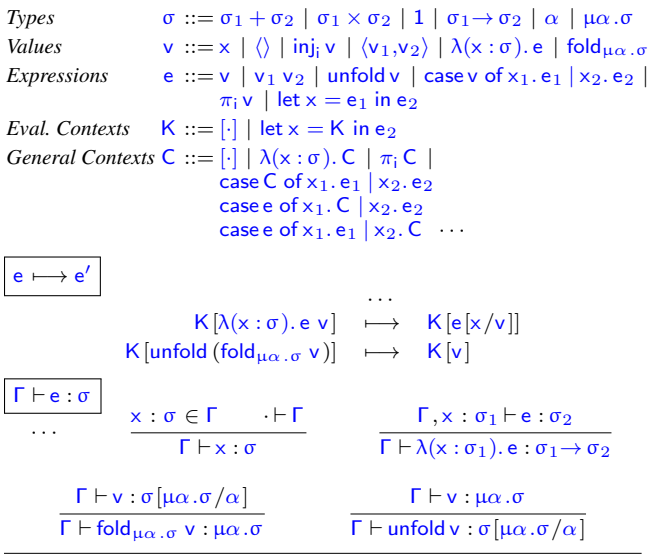

Figure 1. λ^{S}: Syntax + Semantics (excerpts)

Source Language Our source language λ^{S} is a simply-typed lambda calculus with unit, sums, pairs, and recursive types in a monadic normal form. Figure 1 presents the syntax and excerpts of the semantics. We present the dynamic semantics ($e \longmapsto e'$) using evaluation contexts K [14] to define a standard left-to-right call-by-value semantics. Since our language is in a normal form, the only non-trivial evaluation contexts are let-bindings. We elide most of the reduction rules and typing rules ($\Gamma \vdash e : \sigma$) as they are completely standard. The typing environment Γ maps term variables x to their types σ.

Figure 1 also presents an excerpt of the syntax for general contexts which are expressions with a single hole in them. We omit some of the details caused by the monadic syntax; for instance, some contexts can only be plugged with values. Context typing ($\vdash C : (\Gamma \vdash \sigma) \Rightarrow (\Gamma' \vdash \sigma')$), ensures that for any expression e such that $\Gamma \vdash e : \sigma$, we can conclude that $\Gamma' \vdash C[e] : \sigma'$.

We define contextual equivalence ($\Gamma \vdash e_1 \approx^{ctx}_{\mathrm{S}} e_2 : \sigma$) for λ^{S} as follows. Informally, two components e_1 and e_2 are contextually equivalent if either can be replaced by the other in any appropriately typed program context C without affecting the program's observable behavior. As it is a simple, functional language, we take termination (written $e \Downarrow$) as our notion of observable behavior. We write $e_1 \Updownarrow e_2$ when $e_1 \Downarrow$ if and only if $e_2 \Downarrow$.

Definition 2.1 (λ^{S} **Contextual Equivalence**)

$$\Gamma \vdash e_1 \approx^{ctx}_{\mathrm{S}} e_2 : \sigma \stackrel{\text{def}}{=} \Gamma \vdash e_1 : \sigma \ \wedge\ \Gamma \vdash e_2 : \sigma \ \wedge\ \forall \sigma', C.\ \vdash C : (\Gamma \vdash \sigma) \Rightarrow (\cdot \vdash \sigma') \implies (C[e_1] \Updownarrow C[e_2])$$

Target Language Our target language λ^{T} is a polymorphic λ-calculus with the empty type, sums, n-ary tuples, existential types, recursive types, and exceptions tracked by a modal type system. Figure 2 presents the syntax and excerpts of the dynamic and static semantics.

The target language has three syntactic categories for terms: $\mathbf{v}$ is a value, $\mathbf{e}$ is a computation that may have effects and $\mathbf{r}$ is a result, i.e. a normalized computation: either a returned value $\mathbf{return\ v}$ or a raised exception $\mathbf{raise\ v}$.

The let-form of the λ^{S} is subsumed in λ^{T} by a combined let and try-catch form called **handle** in the style of Benton and Kennedy [8]. On a successful computation, *i.e.*, a **return**, it continues with the left branch:

$$\mathbf{handle\ return\ v\ with\ (x.e_1)\ (y.e_2)} \longmapsto \mathbf{e_1}[\mathbf{v}/\mathbf{x}]$$

On an exception it continues with the right branch:

$$\mathbf{handle\ raise\ v\ with\ (x.e_1)\ (y.e_2)} \longmapsto \mathbf{e_2}[\mathbf{v}/\mathbf{y}]$$

We define let-forms as syntactic sugar for a handle that immediately re-raises any exception it encounters. We similarly define a more traditional try-catch by doing the opposite:

$$\mathbf{let\ x = e\ in\ e'} \stackrel{\text{def}}{=} \mathbf{handle\ e\ with\ (x.e')\ (y.raise\ y)}$$
$$\mathbf{catch\ y = e\ in\ e'} \stackrel{\text{def}}{=} \mathbf{handle\ e\ with\ (x.return\ x)\ (y.e')}$$

We use a modal type system to track exceptions: $\boldsymbol{\tau}$ is a value type (for values $\mathbf{v}$) and $\boldsymbol{\theta}$ is a computation type (for computations $\mathbf{e}$). If $\mathbf{e}$ has type $\boldsymbol{\theta} = \mathbf{E}\,\boldsymbol{\tau}_{\mathbf{exn}}\,\boldsymbol{\tau}$ then type soundness for this language means that if $\mathbf{e}$ reduces to a normal form it will either be a $\mathbf{return\ v}$ where $\mathbf{v}$ has type $\boldsymbol{\tau}$, or a $\mathbf{raise\ v'}$ where $\mathbf{v'}$ has type $\boldsymbol{\tau}_{\mathbf{exn}}$. Crucially for our compiler, we can use the empty type $\mathbf{0}$ as the exception type to enforce that a computation does not throw an exception.

Context typing and contextual equivalence are defined analogously to λ^{S}.

Definition 2.2 (λ^{T} **Contextual Equivalence**)

$$\mathbf{\Delta; \Gamma \vdash e_1 \approx^{ctx}_{\mathrm{T}} e_2 : \theta} \stackrel{\text{def}}{=} \mathbf{\Delta; \Gamma \vdash e_1 : \theta} \ \wedge\ \mathbf{\Delta; \Gamma \vdash e_2 : \theta} \ \wedge\ \forall \boldsymbol{\theta'}, \mathbf{C}.\ \vdash \mathbf{C} : (\mathbf{\Delta; \Gamma \vdash \theta}) \Rightarrow (\cdot; \cdot \vdash \boldsymbol{\theta'}) \implies (\mathbf{C}[\mathbf{e_1}] \Updownarrow \mathbf{C}[\mathbf{e_2}])$$

Closure Conversion Closure conversion is a standard internal compiler pass. It translates functions with references to free variables, i.e. variables from the local environment, to be "closed" so that all variables references are bound by the functions parameters. This supports easily heap allocating functions later. Closure conversion collects the values of free variables used in a function definition into a *closure environment* that is stored with the function, where the function itself is modified to take the environment as an additional input. A naïve type translation, for instance, one that represents closure environments as tuples, does not even preserve well-typedness—*i.e.*, two terms that have the same type in the source language may not have the same type in the target language. To see why, consider two functions $\mathrm{e}_1 = \lambda \mathrm{x}.\,\mathrm{x}$ and $\mathrm{e}_2 = \lambda \mathrm{x}.\,\mathrm{z}$ of type $\mathrm{bool} \to \mathrm{bool}$, where z is a free variable of type bool. The function part of the translation of e_1 would have type $\langle\langle\rangle, \mathrm{bool}\rangle \to \mathrm{bool}$ while the translation of e_2 would have type $\langle\langle \mathrm{bool}\rangle, \mathrm{bool}\rangle \to \mathrm{bool}$.

To preserve equivalence to a target language with exceptions, we must ensure target contexts cannot use exceptions to make additional observations of translated terms. For instance, consider $\mathrm{e}_1 = \lambda \mathrm{f}.\,(\mathrm{f\ true; f\ false; }\langle\rangle)$ and $\mathrm{e}_2 = \lambda \mathrm{f}.\,(\mathrm{f\ false; f\ true; }\langle\rangle)$. In a language with just non-termination, these terms are contextually equivalent. However, if the argument f can raises an exception, a context can distinguish these terms. The context $\mathrm{catch\ y} = ([\cdot]\ (\lambda \mathrm{x}.\,\mathrm{raise\ x}))\ \mathrm{in\ y}$ returns true when given e_1 and false when given e_2.

Minamide et al. [25] introduced a type translation that uses existential types to keep the closure environment abstract. Our closure conversion pass is a straightforward extension of this typed closure conversion to accommodate a modal type system. Figure 4 presents the type translation which is split into the value type translation σ^{+} and computation type translation $\sigma^{\div}$. A value of type σ is translated to a value of some value type $\boldsymbol{\tau} = \sigma^{+}$. Non-trivial expressions of type σ are translated to some computation type $\boldsymbol{\theta} = \sigma^{\div}$, where $\sigma^{\div} = \mathbf{E}\,\mathbf{0}\,\sigma^{+}$, indicating that if this computation terminates it will result in a value of type σ^{+}. The value type translation σ^{+} is defined by structural recursion in all cases except for functions. A function of type $\sigma_1 \to \sigma_2$ is compiled to a closure, *i.e.*, a pair of the function and its environment: $\exists \boldsymbol{\alpha}.\,\langle(\langle \boldsymbol{\alpha}, \sigma_1{}^{+}\rangle \to \sigma_2{}^{\div}), \boldsymbol{\alpha}\rangle$. The type of the environment is existentially quantified so that functions of the same type but with different environments are translated to functions of the same type. Parametricity of the language should then ensure that (standard) typed closure conversion is equivalence preserving. The use of existential types ensures that the function component of a closure can only ever be called with the environment it is packaged with, and ensures the environment can only be used as an argument

$$
\begin{array}{llll}
\textit{Value Types} & \tau & ::= & \tau_1 + \tau_2 \mid 1 \mid \langle \overline{\tau} \rangle \mid \alpha \mid \forall\,[\alpha].\tau \to \theta \mid \mu\alpha.\tau \mid \exists\alpha.\tau \mid 0 \\
\textit{Computation Types} & \theta & ::= & \mathbf{E}\,\tau_1\,\tau_2 \\
\textit{Values} & \mathrm{v} & ::= & \mathrm{x} \mid \mathbf{inj}_1\,\mathrm{v}_1 \mid \mathbf{inj}_2\,\mathrm{v}_2 \mid \langle \overline{\mathrm{v}} \rangle \mid \lambda\,[\alpha]\,(\mathrm{x}:\tau).\,\mathrm{e} \mid \mathbf{fold}_{\mu\alpha.\tau}\,\mathrm{v} \mid \mathbf{pack}(\tau,\mathrm{v}) \\
\textit{Results} & \mathrm{r} & ::= & \mathbf{return}\,\mathrm{v} \mid \mathbf{raise}\,\mathrm{v} \\
\textit{Computations} & \mathrm{e} & ::= & \mathrm{r} \mid \mathrm{v}.\mathbf{i} \mid \mathbf{unfold}\,\mathrm{v} \mid \mathbf{handle}\,\mathrm{e}\,\mathbf{with}\,(\mathrm{x}.\mathrm{e}_1)\,(\mathrm{y}.\mathrm{e}_2) \mid \mathrm{v}_1\,[\tau]\,\mathrm{v}_2 \\
 & & & \quad \mid \mathbf{case}\,\mathrm{v}\,\mathbf{of}\,\mathrm{x}_1.\mathrm{e}_1 \mid \mathrm{x}_2.\mathrm{e}_2 \mid \mathbf{unpack}(\alpha,\mathrm{x}) = \mathrm{v}\,\mathbf{in}\,\mathrm{e} \\
\textit{Evaluation Contexts} & \mathrm{K} & ::= & [\cdot] \mid \mathbf{handle}\,\mathrm{K}\,\mathbf{with}\,(\mathrm{x}.\mathrm{e}_1)\,(\mathrm{y}.\mathrm{e}_2)
\end{array}
$$

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

$$
\begin{array}{rcl}
 & \cdots & \\
\mathrm{K}\,[(\lambda\,[\alpha]\,(\mathrm{x}:\tau).\,\mathrm{e})\,[\tau']\,\mathrm{v}] & \longmapsto & \mathrm{K}\,\mathrm{e}[\tau'/\alpha][\mathrm{v}/\mathrm{x}] \\
\mathrm{K}\,[\mathbf{unpack}(\alpha,\mathrm{x}) = (\mathbf{pack}(\tau,\mathrm{v}))\,\mathbf{in}\,\mathrm{e}] & \longmapsto & \mathrm{K}\,[\mathrm{e}[\tau/\alpha][\mathrm{v}/\mathrm{x}]] \\
\mathrm{K}\,[\mathbf{handle}\,(\mathbf{return}\,\mathrm{v})\,\mathbf{with}\,(\mathrm{x}.\mathrm{e}_1)\,(\mathrm{y}.\mathrm{e}_2)] & \longmapsto & \mathrm{K}\,[\mathrm{e}_1[\mathrm{v}/\mathrm{x}]] \\
\mathrm{K}\,[\mathbf{handle}\,(\mathbf{raise}\,\mathrm{v})\,\mathbf{with}\,(\mathrm{x}.\mathrm{e}_1)\,(\mathrm{y}.\mathrm{e}_2)] & \longmapsto & \mathrm{K}\,[\mathrm{e}_2[\mathrm{v}/\mathrm{y}]]
\end{array}
$$

$\boxed{\Delta \vdash \tau}$

$$
\cdots \quad \frac{\alpha \in \Delta}{\Delta \vdash \alpha} \qquad \frac{\Delta,\alpha \vdash \tau_1 \qquad \Delta,\alpha \vdash \tau_2}{\Delta \vdash \forall\,[\alpha].\tau_1 \to \tau_2} \qquad \frac{\Delta,\alpha \vdash \tau}{\Delta \vdash \mu\alpha.\tau} \qquad \frac{\Delta,\alpha \vdash \tau}{\Delta \vdash \exists\alpha.\tau}
$$

$\boxed{\Delta;\Gamma \vdash \mathrm{v} : \tau}$

$$
\cdots \quad \frac{\Delta;\Gamma \vdash \mathrm{v_i} : \tau_i}{\Delta;\Gamma \vdash \langle \overline{\mathrm{v}} \rangle : \langle \overline{\tau} \rangle} \qquad \frac{\Delta \vdash \Gamma \qquad \alpha;\mathrm{x} : \tau \vdash \mathrm{e} : \theta}{\Delta;\Gamma \vdash \lambda\,[\alpha]\,(\mathrm{x}:\tau).\,\mathrm{e} : \forall\,[\alpha].\tau \to \theta} \qquad \frac{\Delta;\Gamma \vdash \mathrm{v} : \tau[\tau'/\alpha] \qquad \Delta \vdash \tau'}{\Delta;\Gamma \vdash \mathbf{pack}(\tau',\mathrm{v}) : \exists\alpha.\tau}
$$

$\boxed{\Delta;\Gamma \vdash \mathrm{e} : \theta}$

$$
\cdots \quad \frac{\Delta;\Gamma \vdash \mathrm{v} : \tau \qquad \Delta \vdash \tau_{\mathbf{exn}}}{\Delta;\Gamma \vdash \mathbf{return}\,\mathrm{v} : \mathbf{E}\,\tau_{\mathbf{exn}}\,\tau} \qquad \frac{\Delta;\Gamma \vdash \mathrm{v} : \tau_{\mathbf{exn}} \qquad \Delta \vdash \tau}{\Delta;\Gamma \vdash \mathbf{raise}\,\mathrm{v} : \mathbf{E}\,\tau_{\mathbf{exn}}\,\tau} \qquad \frac{\Delta;\Gamma \vdash \mathrm{v} : \exists\alpha.\tau \qquad \Delta,\alpha;\Gamma,\mathrm{x}:\tau \vdash \mathrm{e} : \mathbf{E}\,\tau_{\mathbf{exn}}\,\tau' \qquad \Delta \vdash \tau' \qquad \Delta \vdash \tau_{\mathbf{exn}}}{\Delta;\Gamma \vdash \mathbf{unpack}(\alpha,\mathrm{x}) = \mathrm{v}\,\mathbf{in}\,\mathrm{e} : \mathbf{E}\,\tau_{\mathbf{exn}}\,\tau'}
$$

$$
\frac{\Delta;\Gamma \vdash \mathrm{e} : \mathbf{E}\,\tau'_{\mathbf{exn}}\,\tau' \qquad \Delta;\Gamma,\mathrm{x}:\tau' \vdash \mathrm{e}_1 : \mathbf{E}\,\tau_{\mathbf{exn}}\,\tau \qquad \Delta;\Gamma,\mathrm{y}:\tau'_{\mathbf{exn}} \vdash \mathrm{e}_2 : \mathbf{E}\,\tau_{\mathbf{exn}}\,\tau}{\Delta;\Gamma \vdash \mathbf{handle}\,\mathrm{e}\,\mathbf{with}\,(\mathrm{x}.\mathrm{e}_1)\,(\mathrm{y}.\mathrm{e}_2) : \mathbf{E}\,\tau_{\mathbf{exn}}\,\tau}
$$

Figure 2. λ^{T}: Syntax + Semantics (excerpts)

$\boxed{\Gamma \vdash \mathrm{v} : \sigma \rightsquigarrow_v \mathrm{v}}$

$$
\cdots \quad \frac{\mathrm{x} : \sigma \in \Gamma}{\Gamma \vdash \mathrm{x} : \sigma \rightsquigarrow_v \mathrm{x}}
$$

$$
\frac{(\mathrm{y}_1,\ldots,\mathrm{y}_n) = \mathrm{fv}(\lambda(\mathrm{x}:\sigma').\,\mathrm{e}) \qquad \Gamma \vdash \mathrm{y}_i : \sigma_i \qquad \tau_{\mathbf{env}} = \langle {\sigma_1}^+,\ldots,{\sigma_n}^+ \rangle \qquad \Gamma,\mathrm{x}:\sigma \vdash \mathrm{e} : \sigma' \rightsquigarrow_e \mathrm{e}}{\Gamma \vdash \lambda(\mathrm{x}:\sigma).\,\mathrm{e} : \sigma \to \sigma' \rightsquigarrow_v \mathbf{pack}(\tau_{\mathbf{env}}, \langle \lambda(\mathrm{z} : \langle \tau_{\mathbf{env}}, \sigma^+ \rangle).\ \mathbf{let}\,\mathrm{x}_{\mathbf{env}} = \mathbf{return}\,\mathrm{z}.1\,\mathbf{in}\ \mathbf{let}\,\mathrm{y_i} = \mathbf{return}\,\mathrm{x}_{\mathbf{env}}.\mathrm{i}\,\mathbf{in}\ \cdots\ \mathbf{let}\,\mathrm{x} = \mathbf{return}\,\mathrm{z}.2\,\mathbf{in}\,\mathrm{e}, \langle \mathrm{y}_1,\ldots,\mathrm{y}_n \rangle \rangle)
$$

$\boxed{\Gamma \vdash \mathrm{e} : \sigma \rightsquigarrow_e \mathrm{e}}$

$$
\cdots \quad \frac{\Gamma \vdash \mathrm{v} : \sigma \rightsquigarrow_v \mathrm{v}}{\Gamma \vdash \mathrm{v} : \sigma \rightsquigarrow_e \mathbf{return}\,\mathrm{v}} \qquad \frac{\Gamma \vdash \mathrm{e}_1 : \sigma_1 \rightsquigarrow_e \mathrm{e}_1 \qquad \Gamma,\mathrm{x} \vdash \mathrm{e}_2 : \sigma_2 \rightsquigarrow_e \mathrm{e}_2}{\Gamma \vdash \mathsf{let}\,\mathrm{x} = \mathrm{e}_1\,\mathsf{in}\,\mathrm{e}_2 : \sigma_2 \rightsquigarrow_e \mathbf{let}\,\mathrm{x} = \mathrm{e}_1\,\mathbf{in}\,\mathrm{e}_2}
$$

$$
\frac{\Gamma \vdash \mathrm{v}_1 : \sigma_1 \to \sigma_2 \rightsquigarrow_v \mathrm{v}_1 \qquad \Gamma \vdash \mathrm{v}_2 : \sigma_1 \rightsquigarrow_v \mathrm{v}_2}{\Gamma \vdash \mathrm{v}_1\,\mathrm{v}_2 : \sigma_2 \rightsquigarrow_e \mathbf{unpack}(\alpha,\mathrm{z}) = \mathrm{v}_1\,\mathbf{in}\ \mathbf{let}\,\mathrm{y}_1 = \mathbf{return}\,\mathrm{z}.1\,\mathbf{in}\ \mathbf{let}\,\mathrm{y}_2 = \mathbf{return}\,\mathrm{z}.2\,\mathbf{in}\ \mathrm{y}_1\,\langle \mathrm{y}_2, \mathrm{v}_2 \rangle}
$$

Figure 3. Closure Conversion: Term Translation (excerpts)

$$
\begin{array}{rcl}
\sigma^{\div} & = & \mathbf{E}\,0\,\sigma^+ \\
1^+ & = & 1 \\
\alpha^+ & = & \alpha \\
(\sigma_1 + \sigma_2)^+ & = & {\sigma_1}^+ + {\sigma_2}^+ \\
(\sigma_1 \times \sigma_2)^+ & = & \langle {\sigma_1}^+, {\sigma_2}^+ \rangle \\
(\mu\alpha.\sigma)^+ & = & \mu\alpha.\sigma^+ \\
(\sigma_1 \to \sigma_2)^+ & = & \exists\alpha.\,\langle (\langle \alpha, {\sigma_1}^+ \rangle \to {\sigma_2}^{\div}), \alpha \rangle
\end{array}
\qquad
\begin{array}{rcl}
(\cdot)^+ & = & \cdot \\
(\Gamma, \mathrm{x} : \sigma)^+ & = & \Gamma^+, \mathrm{x} : \sigma^+
\end{array}
$$

Figure 4. Closure Conversion: Type Translation

to the function it is pacakged with. Furthermore the output type of the function is ${\sigma_2}^{\div} = \mathbf{E}\,0\,{\sigma_2}^+$, guaranteeing that when the function is called, it does not raise an exception. This is key to ensuring that a target context cannot provide a closure as input to a compiled function (that itself expects a function argument at source level) such that the function inside the closure raises an uncaught exception. Thus, target contexts cannot use exceptions to make additional observations.

The term translation is given in Figure 3. We define a value translation $\Gamma \vdash \mathrm{v} : \sigma \rightsquigarrow_v \mathrm{v}$ and an expression translation $\Gamma \vdash \mathrm{e} : \sigma \rightsquigarrow_e \mathrm{e}$. Note that since we translate open terms, we translate a free variable x to $\mathbf{x}$—the same variable name but in a different color (language). In the expression translation, we translate values by first translating the value according to the value translation, then wrapping the translated value in a $\mathbf{return}$. We translate functions to an existential package, where all free variables referenced in the body of the function are put in the environment part of the package, and the function part of the package first introduces the free variables via let before executing the translated function body. We translate function application by first unpacking the closure and then applying the underlying function to the pair of the argument and the environment. We verify that this translation is type preserving.

Theorem 2.3 (Closure Conversion is Type Preserving)

1. *If* $\Gamma \vdash \mathrm{v} : \sigma$ *and* $\Gamma \vdash \mathrm{v} : \sigma \rightsquigarrow_v \mathrm{v}$, *then* $\cdot;\Gamma^+ \vdash \mathrm{v} : \sigma^+$.
2. *If* $\Gamma \vdash \mathrm{e} : \sigma$ *and* $\Gamma \vdash \mathrm{e} : \sigma \rightsquigarrow_e \mathrm{e}$, *then* $\cdot;\Gamma^+ \vdash \mathrm{e} : \sigma^{\div}$.

3. Multi-Language Semantics

We use multi-language semantics [23, 6, 29] to define interoperability between source and target components. The multi-language λ^{ST} provides a natural specification for semantics preservation, given in §3.1.

$$
\begin{array}{llll}
\textit{Value Types} & \tau & ::= \sigma \mid \boldsymbol{\tau} & v ::= \mathsf{v} \mid \mathbf{v} \\
\textit{Comp. Types} & \theta & ::= \sigma \mid \boldsymbol{\theta} & r ::= \mathsf{v} \mid \mathbf{r} \\
\textit{All Types} & \varphi & ::= \tau \mid \theta & e ::= \mathsf{e} \mid \mathbf{e} \\
 & \Delta & ::= \boldsymbol{\Delta} & \mathsf{e} ::= \ldots \mid {}^{\sigma}\mathcal{ST}\,\mathbf{e} \\
 & \Gamma & ::= \cdot \mid \Gamma, \mathsf{x} : \sigma \mid \Gamma, \mathbf{x} : \boldsymbol{\tau} & \mathbf{e} ::= \ldots \mid \mathcal{TS}^{\sigma}\,\mathsf{e} \\
 & \mathsf{K} & ::= \ldots \mid {}^{\sigma}\mathcal{ST}\,\mathbf{K} & \mathsf{C} ::= \ldots \mid {}^{\sigma}\mathcal{ST}\,\mathbf{C} \\
 & \mathbf{K} & ::= \ldots \mid \mathcal{TS}^{\sigma}\,\mathsf{K} & \mathbf{C} ::= \ldots \mid \mathcal{TS}^{\sigma}\,\mathsf{C} \\
 & K & ::= \mathsf{K} \mid \mathbf{K} & C ::= \mathsf{C} \mid \mathbf{C}
\end{array}
$$

$$
\boxed{\Delta;\Gamma \vdash e : \theta} \quad \ldots \quad
\frac{\Delta;\Gamma \vdash \mathbf{e} : \sigma^{\div}}{\Delta;\Gamma \vdash {}^{\sigma}\mathcal{ST}\,\mathbf{e} : \sigma} \qquad
\frac{\Delta;\Gamma \vdash \mathsf{e} : \sigma}{\Delta;\Gamma \vdash \mathcal{TS}^{\sigma}\,\mathsf{e} : \sigma^{\div}}
$$

Figure 5. λ^{ST}: Syntax and Static Semantics (excerpts)

In Figure 5, we define the syntax for λ^{ST} by extending the syntax of λ^{S} and λ^{T}. For instance, we extend the source and target terms to include boundary terms, and extend expression contexts to include the boundary contexts. Boundary terms allow injecting a term from one language into the other language: ${}^{\sigma}\mathcal{ST}\,\mathbf{e}$ injects target $\mathbf{e}$ into source; $\mathcal{TS}^{\sigma}\,\mathsf{e}$ injects source e into target. We define syntactic categories e for all terms, φ for all types, τ for value types, and so on.

We also define typing rules for boundaries in Figure 5. The two boundaries mediate between the types σ and $\sigma^{\div}$. This ensures that source (resp. target) components can only be plugged into target (resp. source) contexts if the component's type and context's expectation are compatible according to our type translation. Other typing rules for the multi-language are inherited from the source and target languages with the environments for all the rules changed to $\Delta;\Gamma$.

Henceforth, when we refer to a "source term" (e) in λ^{ST}, we mean a multi-language term that has type σ. Analogously, a "target term" ($\mathbf{e}$) in λ^{ST}, is a multi-language term of type $\boldsymbol{\theta}$.

The operational semantics for the multi-language, defined in Figure 6, inherits reductions from the source and target languages. Note that we evaluate under a boundary $\mathcal{TS}$ till we reach a value v, and under a boundary $\mathcal{ST}$ until we reach a result $\mathbf{r}$. The reduction rules for boundary terms essentially perform our translation during runtime directed by the type σ on the boundary. Unlike our compiler which translates based on program syntax, this boundary translation is entirely extensional and only wraps values. For instance, $\mathcal{TS}^{1}\,\langle\rangle$ evaluates to $\mathbf{return}\ \langle\rangle$. Since $\mathcal{TS}^{\sigma}\,\mathsf{v}$ boundaries are computations, the translation of v must be injected into the exception monad in the target language. For either boundary around a fold, the value is unfolded in one language, then that value is translated and refolded in the other.

For a boundary around a source function, we evaluate to a closure with an empty environment, since the source function must be closed (because terms are closed at runtime). In that closure, we generate a target function that wraps its argument with a boundary and calls the original source function. For a boundary around a target closure, we evaluate to a source function that unpacks the closure then calls the target function with the environment and the source argument wrapped in a boundary term.

Contextual equivalence in the multi-language is analogous to the definition in the source and target languages.

Definition 3.1 (λ^{ST} Contextual Equivalence)

$$
\Delta;\Gamma \vdash e_1 \approx^{ctx}_{\mathrm{ST}} e_2 : \theta \stackrel{\mathrm{def}}{=} \Delta;\Gamma \vdash e_1 : \theta \ \wedge\ \Delta;\Gamma \vdash e_2 : \theta \ \wedge\ \forall \theta', C.\, \vdash C : (\Delta;\Gamma \vdash \theta) \Rightarrow (\cdot;\cdot \vdash \theta') \implies C[e_1] \Updownarrow C[e_2]
$$

Finally, we state Lemma 3.2, which provides a simple correctness property of our multi-language semantics, and implies Corollary 3.3. The latter is important as it justifies stating cross-language theorems with an arbitrary choice of placing the boundary on the source or target term.

Lemma 3.2 (Boundary Cancellation)

1. *If* $\Delta;\Gamma \vdash \mathsf{e} : \sigma$, *then* $\Delta;\Gamma \vdash \mathsf{e} \approx^{ctx}_{\mathrm{ST}} {}^{\sigma}\mathcal{ST}\,\mathcal{TS}^{\sigma}\,\mathsf{e} : \sigma$.
2. *If* $\Delta;\Gamma \vdash \mathbf{e} : \sigma^{\div}$, *then* $\Delta;\Gamma \vdash \mathbf{e} \approx^{ctx}_{\mathrm{ST}} \mathcal{TS}^{\sigma}\,{}^{\sigma}\mathcal{ST}\,\mathbf{e} : \sigma^{\div}$.

Corollary 3.3

$\Delta;\Gamma \vdash \mathsf{e} \approx^{ctx}_{\mathrm{ST}} {}^{\sigma}\mathcal{ST}\,\mathbf{e} : \sigma$ *iff* $\Delta;\Gamma \vdash \mathcal{TS}^{\sigma}\,\mathsf{e} \approx^{ctx}_{\mathrm{ST}} \mathbf{e} : \sigma^{\div}$.

3.1 Cross-Language Equivalence

Now that we have defined an interoperation semantics for the two languages, we have a completely natural way to define when a target term accurately simulates the behavior of a source term. This gives us a clear and concise definition of compiler correctness.

First we consider closed terms. Since our multi-language semantics defines embedding of a source component in a target context, we will consider a source component $\mathsf{e} : \sigma$ and a target component $\mathbf{e} : \sigma^{\div}$ to be "equivalent" when $\mathcal{TS}^{\sigma}\,\mathsf{e}$ and $\mathbf{e}$ are contextually equivalent in λ^{ST}. We define cross-language equivalence on closed terms as follows.

Definition 3.4 (Cross-Lang. Equiv. ${}^{\mathcal{S}}\simeq^{\mathcal{T}}$: Closed Terms)

$$
\cdot \vdash \mathsf{e}\ {}^{\mathcal{S}}\simeq^{\mathcal{T}}\ \mathbf{e} : \sigma \stackrel{\mathrm{def}}{=} \cdot;\cdot \vdash \mathcal{TS}^{\sigma}\,\mathsf{e} \approx^{ctx}_{\mathrm{ST}} \mathbf{e} : \sigma^{\div}
$$

Recall that we wish to prove *compositional* semantics preservation for closure conversion—that is, that compilation of *components*, not just whole programs, is semantics preserving. Since our notion of component is an open term, next we define cross-language equivalence for open terms, which in the special case of closed terms yields the equivalence above.

For cross-language equivalence for open terms, we have an open source term $\Gamma \vdash \mathsf{e} : \sigma$ that we want to relate to an open target term $\cdot;\Gamma^{+} \vdash \mathbf{e} : \sigma^{\div}$—in particular, for compiler correctness we will be interested in the case where $\mathbf{e}$ is the compilation of e. The multi-language semantics again gives us a natural definition in terms of boundaries. With closed terms we used boundaries to translate the *output* of the programs. With open terms we will use boundaries to additionally translate the *inputs*, that is, the free variables.

To do this we introduce new syntactic sugar that acts like the boundary on free variables.

Definition 3.5 (Boundaries on Free Variables)

For a term $\Gamma \vdash \mathsf{e} : \sigma$, *where* $\Gamma = \mathsf{x}_1 : \sigma_1, \ldots, \mathsf{x}_n : \sigma_n$, *we define*

$$
\begin{array}{l}
\mathsf{e}^{\ulcorner}\mathcal{ST} \stackrel{\mathrm{def}}{=} \mathsf{let}\, \mathsf{x}_1 = {}^{\sigma_1}\mathcal{ST}\, \mathbf{return}\ \mathbf{x_1}\ \mathsf{in} \\
\qquad\qquad \vdots \\
\qquad\qquad \mathsf{let}\, \mathsf{x}_n = {}^{\sigma_n}\mathcal{ST}\, \mathbf{return}\ \mathbf{x_n}\ \mathsf{in}\ \mathsf{e}
\end{array}
$$

Note that $\cdot;\Gamma^{+} \vdash \mathsf{e}^{\ulcorner}\mathcal{ST} : \sigma$. *We define* $\mathbf{e}\,\mathcal{TS}^{\ulcorner^{+}}$ *analogously.*

We now define the source term e to be equivalent to the target term $\mathbf{e}$ when the latter is contextually equivalent to a term that translates inputs from the target to the source language, runs e and translates the output from the source back to the target.

Definition 3.6 (Cross-Lang. Equiv. ${}^{\mathcal{S}}\simeq^{\mathcal{T}}$: Closed Terms)

$$
\Gamma \vdash \mathsf{e}\ {}^{\mathcal{S}}\simeq^{\mathcal{T}}\ \mathbf{e} : \sigma \stackrel{\mathrm{def}}{=} \cdot;\Gamma \vdash \mathcal{TS}^{\sigma}\,\mathsf{e}^{\ulcorner}\mathcal{ST} \approx^{ctx}_{\mathrm{ST}} \mathbf{e} : \sigma^{\div}
$$

4. Correctness and Equivalence Reflection

Direct proofs of contextual equivalence are intractable due to the universal quantification over contexts in the definition of $\approx^{ctx}_{\mathrm{ST}}$. As is standard, we define a logical relation that is sound and complete with respect to contextual equivalence. and use this logical relation instead of contextual equivalence. We conclude this section with

$e \longmapsto e'$

$$
\begin{array}{rcl}
K\,[{}^{\sigma_1 \times \sigma_2}\mathcal{ST}\ \mathbf{return}\ \mathsf{v}] & \longmapsto & K\,[\mathsf{let}\ \mathsf{x}_1 = {}^{\sigma_1}\mathcal{ST}\ \mathsf{v}.1\ \mathsf{in}\ \mathsf{let}\ \mathsf{x}_2 = {}^{\sigma_2}\mathcal{ST}\ \mathsf{v}.2\ \mathsf{in}\ \langle \mathsf{x}_1, \mathsf{x}_2 \rangle] \\
K\,[\mathcal{TS}^{\sigma_1 \times \sigma_2}\ \mathsf{v}] & \longmapsto & K\,[\mathbf{let}\ \mathbf{x_1} = \mathcal{TS}^{\sigma_1}\ \pi_1\ \mathsf{v}\ \mathbf{in\ let\ x_2} = \mathcal{TS}^{\sigma_2}\ \pi_2\ \mathsf{v}\ \mathbf{in\ return}\ \langle \mathbf{x_1}, \mathbf{x_2} \rangle] \\
K\,[\mathcal{TS}^{1}\ \mathsf{v}] & \longmapsto & K\,[\mathbf{return}\ \langle\rangle] \\
K\,[{}^{1}\mathcal{ST}\ \mathbf{return}\ \mathsf{v}] & \longmapsto & K\,[\langle\rangle] \\
K\,[{}^{\sigma_1 + \sigma_2}\mathcal{ST}\ \mathbf{return\ inj_i}\ \mathsf{v}] & \longmapsto & K\,[\mathsf{let}\ \mathsf{x} = {}^{\sigma_i}\mathcal{ST}\ \mathbf{return}\ \mathsf{v}\ \mathsf{in\ inj_i\,x}] \\
K\,[\mathcal{TS}^{\sigma_1 + \sigma_2}\ \mathsf{inj_i\ v}] & \longmapsto & K\,[\mathbf{let\ x} = \mathcal{TS}^{\sigma_i}\ \mathsf{v}\ \mathbf{in\ return\ inj_i\,x}] \\
K\,[{}^{\mu\alpha.\sigma}\mathcal{ST}\ \mathbf{return}\ \mathsf{v}] & \longmapsto & K\,[\mathsf{let\ x} = {}^{\sigma[\mu\alpha.\sigma/\alpha]}\mathcal{ST}\ \mathbf{return\ unfold}\ \mathsf{v}\ \mathsf{in\ fold}_{\mu\alpha.\sigma}\,\mathsf{x}] \\
K\,[\mathcal{TS}^{\mu\alpha.\sigma}\ \mathsf{v}] & \longmapsto & K\,[\mathbf{let\ v} = \mathcal{TS}^{\sigma[\mu\alpha.\sigma/\alpha]}\ \mathsf{unfold\ v}\ \mathbf{in\ return\ fold}_{(\mu\alpha.\sigma)^+}\ \mathbf{v}] \\
K\,[{}^{\sigma_1 \to \sigma_2}\mathcal{ST}\ \mathbf{return}\ \mathsf{v}] & \longmapsto & K\,[\lambda(\mathsf{x}:\sigma_1).\,{}^{\sigma_2}\mathcal{ST}\ \left(\begin{array}{l}\mathbf{unpack}(\boldsymbol{\alpha}, \mathbf{z}) = \mathsf{v}\ \mathbf{in\ let\ x_f} = \mathbf{return\ z}.1\ \mathbf{in} \\ \qquad\qquad \mathbf{let\ x_{env}} = \mathbf{return\ z}.2\ \mathbf{in} \\ \qquad\qquad \mathbf{let\ x} = \mathcal{TS}^{\sigma_1}\,\mathsf{x}\ \mathbf{in\ x_f}\ [\boldsymbol{\alpha}]\ \langle \mathbf{x_{env}}, \mathbf{x} \rangle\end{array}\right)] \\
K\,[\mathcal{TS}^{\sigma_1 \to \sigma_2}\ \mathsf{v}] & \longmapsto & K\,[\mathbf{return\ pack}(\mathbf{1}, \langle \lambda(\mathbf{z}:\langle \mathbf{1}, \sigma_1{}^{+} \rangle).\,\mathcal{TS}^{\sigma_2}\ (\mathsf{let\ x} = {}^{\sigma_1}\mathcal{ST}\ \mathbf{return\ z}.2\ \mathsf{in\ v\ x}), \langle\rangle \rangle)]
\end{array}
$$

Figure 6. λ^{ST}: Dynamic Semantics

proofs of semantics preservation (Theorem 1.1) and equivalence reflection (Theorem 1.2).

In Figure 7 we define the logical relation for λ^{ST}. We use a *step-indexed*, *biorthogonal* logical relation [12]. Biorthogonality is a standard technique [21, 30] to ensure that a logical relation is complete with respect to contextual equivalence. Step-indexing is a standard technique to provide an induction metric in the presence of recursive types [4]—the logical relation is defined by induction on the step-index and nested induction on types φ. As step indices are not critical to understanding the definitions, we largely ignore them in our explanation.

The structure of the λ^{ST} logical relation is as follows: $\mathcal{O}$ relates programs that yield the same observations; $\mathcal{V}[\![\tau]\!]$ relates values at type τ; $\mathcal{R}[\![\tau]\!]$ related *results* at type τ; $\mathcal{K}[\![\theta]\!]$ relates evaluation contexts that, when given related results of type θ, yield related observations; $\mathcal{E}[\![\theta]\!]$ relates expressions of type θ that, when plugged into related evaluation contexts, yield related observations. Each relation is restricted to contain only well-typed members (elided for brevity). Each of these relations is indexed by a type φ and a relational interpretation ρ that provides mappings for the free type variables in φ.

The observation relation $\mathcal{O}$ says that two terms yield the same observations when given k steps if they are both still running after k steps or if they both terminate.

The value relation $\mathcal{V}[\![\sigma]\!]$ is the standard inductive definition. At type 1 the unit value $\langle\rangle$ is related to itself. Functions are related at type $\sigma_1 \to \sigma_2$ when, given arguments related in $\mathcal{V}[\![\sigma_1]\!]$, they produce related results in $\mathcal{E}[\![\sigma_2]\!]$. The relation for polymorphic functions is standard: for arbitrary types and admissible relations on those types, the bodies of the functions must be related under an extended relational interpretation. We write $\rho[\boldsymbol{\alpha} \mapsto (\boldsymbol{\tau_1}, \boldsymbol{\tau_2}, R)]$ to be the relational interpretation ρ extended with types $\boldsymbol{\tau_1}$ and $\boldsymbol{\tau_2}$ and relation R for the type variable $\boldsymbol{\alpha}$. At type $\boldsymbol{\alpha}$ two terms are related if they are in the relation for $\boldsymbol{\alpha}$ in ρ, written $\rho_R(\boldsymbol{\alpha})$.

The relation $\mathcal{K}[\![\theta]\!]$ is slightly non-standard to account for our modal type system. Specifically, two evaluation contexts are related in $\mathcal{K}[\![\theta]\!]$ if, when plugged with related *results*–rather than the standard related *values*–yield related observations. Results, related by $\mathcal{R}[\![\tau]\!]$ are just values in the source language, but are either **return** or **raise** of a value in the target language.

In Figure 8, we lift our logical relation to open terms. We first define when substitutions of type and term variables are related via $\mathcal{D}[\![\Delta]\!]$ and $\mathcal{G}[\![\Gamma]\!]\rho$. We write $\gamma_1(e)$ to close the term variables with the first component of the relational substitution γ, and similarly write $\gamma_2(e)$ to use the second component. We write $\rho_1(e)$ to substitute all type variables in e with the first component type in ρ and similarly $\rho_2(e)$ to use the second component type.

We prove the fundamental property of this logical relation, Theorem 4.1: every well-typed term is related to itself. The proof is by induction on the typing derivation. The case for boundary terms requires Lemma 4.2. This Bridge Lemma is of particular interest in §5.

Theorem 4.1 (Fundamental Property)
If $\Delta;\Gamma \vdash e : \theta$, *then* $\Delta;\Gamma \vdash e \approx^{log}_{\mathcal{E}} e : \theta$.

Lemma 4.2 (Bridge Lemma)

1. *If* $\Delta;\Gamma \vdash \mathbf{e_1} \approx^{log}_{\mathcal{E}} \mathbf{e_2} : \sigma\div$ *then* $\Delta;\Gamma \vdash {}^{\sigma}\mathcal{ST}\ \mathbf{e_1} \approx^{log}_{\mathcal{E}} {}^{\sigma}\mathcal{ST}\ \mathbf{e_2} : \sigma$
2. *If* $\Delta;\Gamma \vdash \mathsf{e_1} \approx^{log}_{\mathcal{E}} \mathsf{e_2} : \sigma$ *then* $\Delta;\Gamma \vdash \mathcal{TS}^{\sigma}\ \mathsf{e_1} \approx^{log}_{\mathcal{E}} \mathcal{TS}^{\sigma}\ \mathsf{e_2} : \sigma\div$.

Finally, we prove that the logical relation and contextual equivalence for the multi-language coincide. The proof is completely standard [4].

Theorem 4.3 ($ctx \equiv log$)
$\Delta;\Gamma \vdash e_1 \approx^{ctx}_{\mathrm{ST}} e_2 : \theta$ *if and only if* $\Delta;\Gamma \vdash e_1 \approx^{log}_{\mathcal{E}} e_2 : \theta$

Semantics preservation To model the cross-language contextual equivalence we defined in §3.1, we define a "cross-language" logical relation using our multi-language logical relation. The cross-language relation, defined in Figure 9, relates source terms of type σ to target terms of type $\sigma\div$, and similarly source values of type σ to target values of type σ^+. The cross-language logical relation is indexed by the source type.

We then prove the semantics preservation theorem, restated below, by using the logical relation instead of contextual equivalence.

Theorem 1.1 (Semantics Preservation)
If $\Gamma \vdash \mathsf{e} : \sigma \rightsquigarrow_e \mathbf{e}$, *then* $\Gamma \vdash \mathsf{e}\ {}^{\mathcal{S}}\simeq^{\mathcal{T}}\ \mathbf{e} : \sigma$.

Proof Sketch It suffices to show $\Gamma \vdash \mathsf{e} \approx_{\div} \mathbf{e} : \sigma$ by Theorem 4.3. The proof is by induction on the translation judgment. Most cases follow trivially by the induction hypothesis. The case for application requires some expected parametric reasoning about the closure environment. □

Equivalence Reflection Equivalence reflection is a simple corollary of semantics preservation, after proving Lemma 4.5, which gives us a means of closure-converting contexts:

Lemma 4.5 (Context Translation)
If $\vdash \mathsf{C} : (\Gamma \vdash \sigma) \Rightarrow (\Gamma' \vdash \sigma')$, $\Gamma \vdash \mathsf{e} : \sigma$ *and* $\Gamma \vdash \mathsf{e} : \sigma \rightsquigarrow_e \mathbf{e}$,
then there exists $\mathbf{C}$ *such that* $\Gamma' \vdash \mathsf{C}[\mathsf{e}] : \sigma \rightsquigarrow_e \mathbf{C}[\mathbf{e}]$.
Furthermore, if $\Gamma \vdash \mathsf{e}' : \sigma$ *and* $\Gamma \vdash \mathsf{e}' : \sigma \rightsquigarrow_e \mathbf{e}'$,
then $\Gamma' \vdash \mathsf{C}[\mathsf{e}'] : \sigma \rightsquigarrow_e \mathbf{C}[\mathbf{e}']$.

Theorem 1.2 (Equivalence Reflection)
If $\Gamma \vdash \mathsf{e_1} : \sigma \rightsquigarrow_e \mathbf{e_1}$, $\Gamma \vdash \mathsf{e_2} : \sigma \rightsquigarrow_e \mathbf{e_2}$ *and* $\cdot;\Gamma^+ \vdash \mathbf{e_1} \approx^{ctx}_{\mathbf{T}} \mathbf{e_2} : \sigma\div$,
then $\Gamma \vdash \mathsf{e_1} \approx^{ctx}_{\mathsf{S}} \mathsf{e_2} : \sigma$.

Proof Let $\mathsf{C} \in \lambda^{\mathrm{S}}$ be an appropriately typed context. We need to show that $\mathsf{C}[\mathsf{e_1}] \Updownarrow \mathsf{C}[\mathsf{e_2}]$. First by Lemma 4.5, we compile

Note: all relations restricted to well-typed members.

$$
\begin{array}{rcl}
\mathrm{Rel}[\tau_1,\tau_2] & \stackrel{\text{def}}{=} & \{\, R \mid \forall (k,v_1,v_2) \in R.\ \cdot;\cdot \vdash v_1 : \tau_1 \ \wedge\ \cdot;\cdot \vdash v_2 : \tau_2 \ \wedge\ \forall j < k.\ (j,v_1,v_2) \in R \,\} \\
\mathcal{O} & \stackrel{\text{def}}{=} & \{\, (k,e_1,e_2) \mid\ (e_1 \Downarrow\ \wedge\ e_2 \Downarrow) \ \vee\ (\mathrm{running}(k,e_1)\ \wedge\ \mathrm{running}(k,e_2)) \,\} \\
\mathcal{V}[\![1]\!]\,\rho & \stackrel{\text{def}}{=} & \{\, (k,\langle\rangle,\langle\rangle) \,\} \\
\mathcal{V}[\![\sigma \to \sigma']\!]\,\rho & \stackrel{\text{def}}{=} & \{\, (k,\lambda(x:\sigma).\,e_1,\lambda(x:\sigma).\,e_2) \mid \forall j \le k.\ \forall v_1,v_2.\ (j,v_1,v_2) \in \mathcal{V}[\![\sigma]\!]\,\rho \implies (j,e_1[v_1/x],e_2[v_2/x]) \in \mathcal{E}[\![\sigma']\!]\,\rho \,\} \\
\mathcal{V}[\![\mu\alpha.\sigma]\!]\,\rho & \stackrel{\text{def}}{=} & \{(0,v_1,v_2)\} \cup \{(k+1,\mathsf{fold}_{\mu\alpha.\sigma}\,v_1,\mathsf{fold}_{\mu\alpha.\sigma}\,v_2) \mid \forall j < k+1.\ (j,v_1,v_2) \in \mathcal{V}[\![\sigma[\mu\alpha.\sigma/\alpha]]\!]\,\rho \,\} \\
\mathcal{V}[\![\sigma_1 + \sigma_2]\!]\,\rho & \stackrel{\text{def}}{=} & \{\, (k,\mathsf{inj}_i\,v_1,\mathsf{inj}_i\,v_2) \mid i \in \{1,2\}\ \wedge\ (k,v_1,v_2) \in \mathcal{V}[\![\sigma_i]\!]\,\rho \,\} \\
\mathcal{V}[\![\sigma \times \sigma']\!]\,\rho & \stackrel{\text{def}}{=} & \{\, (k,\langle v_1,v_1'\rangle,\langle v_2,v_2'\rangle) \mid (k,v_1,v_2) \in \mathcal{V}[\![\sigma]\!]\,\rho\ \wedge\ (k,v_1',v_2') \in \mathcal{V}[\![\sigma']\!]\,\rho \,\} \\
\mathcal{V}[\![0]\!]\,\rho & \stackrel{\text{def}}{=} & \emptyset \\
\mathcal{V}[\![\tau_1 + \tau_2]\!]\,\rho & \stackrel{\text{def}}{=} & \{\, (k,\mathsf{inj}_i\,v_1,\mathsf{inj}_i\,v_2) \mid i \in \{1,2\}\ \wedge\ (k,v_1,v_2) \in \mathcal{V}[\![\tau_i]\!]\,\rho \} \\
\mathcal{V}[\![\langle\tau_1,\ldots,\tau_n\rangle]\!]\,\rho & \stackrel{\text{def}}{=} & \{\, (k,\langle v_1,\ldots,v_n\rangle,\langle v_1',\ldots,v_n'\rangle) \mid \forall i \in \{1\ldots n\}.\ (k,v_i,v_i') \in \mathcal{V}[\![\tau_i]\!]\,\rho \,\} \\
\mathcal{V}[\![\mu\alpha.\tau]\!]\,\rho & \stackrel{\text{def}}{=} & \{\, (k,\mathsf{fold}_{\rho_1(\mu\alpha.\tau)}\,v_1,\mathsf{fold}_{\rho_2(\mu\alpha.\tau)}\,v_2) \mid \forall j < k.\ (j,v_1,v_2) \in \mathcal{V}[\![\tau[\mu\alpha.\tau/\alpha]]\!]\,\rho \,\} \\
\mathcal{V}[\![\alpha]\!]\,\rho & \stackrel{\text{def}}{=} & \rho_R(\alpha) \\
\mathcal{V}[\![\forall[\alpha].\tau \to \mathbf{E}\,\tau_{\mathsf{exn}}\,\tau']\!]\,\rho & \stackrel{\text{def}}{=} & \{\, (k,\lambda[\alpha](x:\rho_1(\tau)).\,e_1,\lambda[\alpha](x:\rho_2(\tau)).\,e_2) \mid \\
 & & \quad \forall \tau_1,\tau_2,R \in \mathrm{Rel}[\tau_1,\tau_2].\ \forall j \le k.\ \forall (j,v_1,v_2) \in \mathcal{V}[\![\tau]\!]\,\rho[\alpha \mapsto (\tau_1,\tau_2,R)]. \\
 & & \quad\quad (j,e_1[\tau_1/\alpha][v_1/x],e_2[\tau_2/\alpha][v_2/x]) \in \mathcal{E}[\![\mathbf{E}\,\tau_{\mathsf{exn}}\,\tau']\!]\,\rho[\alpha \mapsto (\tau_1,\tau_2,R)] \,\} \\
\mathcal{V}[\![\exists\alpha.\tau]\!]\,\rho & \stackrel{\text{def}}{=} & \{\, (k,\mathsf{pack}(\tau_1,v_1),\mathsf{pack}(\tau_2,v_2)) \mid \exists R \in \mathrm{Rel}[\tau_1,\tau_2].\ (k,v_1,v_2) \in \mathcal{V}[\![\tau]\!]\,\rho[\alpha \mapsto (\tau_1,\tau_2,R)] \,\} \\
\mathcal{R}[\![\sigma]\!]\rho & \stackrel{\text{def}}{=} & \mathcal{V}[\![\sigma]\!]\,\rho \\
\mathcal{R}[\![\mathbf{E}\,\tau_{\mathsf{exn}}\,\tau]\!]\rho & \stackrel{\text{def}}{=} & \{\, (k,\mathsf{return}\,v_1,\mathsf{return}\,v_2) \mid (k,v_1,v_2) \in \mathcal{V}[\![\tau]\!]\,\rho \,\} \\
 & \cup & \{\, (k,\mathsf{raise}\,v_1,\mathsf{raise}\,v_2) \mid (k,v_1,v_2) \in \mathcal{V}[\![\tau_{\mathsf{exn}}]\!]\,\rho \,\} \\
\mathcal{E}[\![\theta]\!]\,\rho & \stackrel{\text{def}}{=} & \{\, (k,e_1,e_2) \mid \forall K_1,K_2.\ (k,K_1,K_2) \in \mathcal{K}[\![\theta]\!]\,\rho \implies (k,K_1[e_1],K_2[e_2]) \in \mathcal{O} \} \\
\mathcal{K}[\![\theta]\!]\,\rho & \stackrel{\text{def}}{=} & \{\, (k,K_1,K_2) \mid\ \forall j \le k, r_1, r_2.\ (j,r_1,r_2) \in \mathcal{R}[\![\theta]\!]\rho \implies (j,K_1[r_1],K_2[r_2]) \in \mathcal{O} \} \\
\mathcal{D}[\![\cdot]\!] & \stackrel{\text{def}}{=} & \{\, \emptyset \,\} \\
\mathcal{D}[\![\Delta,\alpha]\!] & \stackrel{\text{def}}{=} & \{\, \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 & \stackrel{\text{def}}{=} & \{\, (k,\emptyset) \mid\ k \in \mathbb{N} \,\} \\
\mathcal{G}[\![\Gamma,x:\tau]\!]\,\rho & \stackrel{\text{def}}{=} & \{\, (k,\gamma[x \mapsto (v_1,v_2)]) \mid\ (k,\gamma) \in \mathcal{G}[\![\Gamma]\!]\,\rho\ \wedge\ (k,v_1,v_2) \in \mathcal{V}[\![\tau]\!]\,\rho \,\}
\end{array}
$$

Figure 7. λ^{ST}: Logical Relation for Closed Terms

$$
\begin{array}{rcl}
\Delta;\Gamma \vdash e_1 \approx^{log}_{\mathcal{E}} e_2 : \theta & \stackrel{\text{def}}{=} & \Delta;\Gamma \vdash e_1 : \theta\ \wedge\ \Delta;\Gamma \vdash e_2 : \theta\ \wedge \\
 & & \forall k \ge 0.\ \forall \rho,\gamma.\ \rho \in \mathcal{D}[\![\Delta]\!]\ \wedge \\
 & & (k,\gamma) \in \mathcal{G}[\![\Gamma]\!]\,\rho \implies \\
 & & (k,\rho_1(\gamma_1(e_1)),\rho_2(\gamma_2(e_2))) \in \mathcal{E}[\![\theta]\!]\,\rho \\
\Delta;\Gamma \vdash v_1 \approx^{log}_{\mathcal{V}} v_2 : \tau & \stackrel{\text{def}}{=} & \cdots \implies \\
 & & (k,\rho_1(\gamma_1(e_1)),\rho_2(\gamma_2(e_2))) \in \mathcal{V}[\![\tau]\!]\,\rho
\end{array}
$$

Figure 8. λ^{ST}: Logical Relation for Open Terms

$$
\begin{array}{rcl}
\mathcal{V}^{+}[\![\sigma]\!] & \stackrel{\text{def}}{=} & \{(k,v_1,v_2) \mid \\
 & & \exists v_2.\ {}^{\sigma}\mathcal{ST}\,v_2 \longmapsto^{*} v_2 \wedge (k,v_1,v_2) \in \mathcal{V}[\![\sigma]\!]\,\emptyset\} \\
\mathcal{E}^{\div}[\![\sigma]\!] & \stackrel{\text{def}}{=} & \{(k,e,e) \mid (k,e,{}^{\sigma}\mathcal{ST}\,e) \in \mathcal{E}[\![\sigma]\!]\,\emptyset\} \\
\mathcal{G}^{+}[\![\cdot]\!] & \stackrel{\text{def}}{=} & \{(k,\emptyset,\emptyset) \mid k \in \mathbb{N}\} \\
\mathcal{G}^{+}[\![\Gamma,x:\sigma]\!] & \stackrel{\text{def}}{=} & \{(k,\gamma[x \mapsto v],\gamma[x \mapsto v]) \mid \\
 & & (k,\gamma,\gamma) \in \mathcal{G}^{+}[\![\Gamma]\!] \wedge (k,v,v) \in \mathcal{V}^{+}[\![\sigma]\!]\} \\
\Gamma \vdash v \approx_{+} v : \sigma & \stackrel{\text{def}}{=} & v \in \lambda^{\mathrm{S}} \wedge v \in \lambda^{\mathrm{T}} \wedge \Gamma \vdash v : \sigma \wedge \cdot;\Gamma^{+} \vdash v : \sigma^{+} \wedge \\
 & & \forall k.\ (k,\gamma,\gamma) \in \mathcal{G}^{+}[\![\Gamma]\!] \implies \\
 & & (k,\gamma(v),\gamma(v)) \in \mathcal{V}^{+}[\![\sigma]\!] \\
\Gamma \vdash e \approx_{\div} e : \sigma & \stackrel{\text{def}}{=} & e \in \lambda^{\mathrm{S}} \wedge e \in \lambda^{\mathrm{T}} \wedge \Gamma \vdash e : \sigma \wedge \cdot;\Gamma^{+} \vdash e : \sigma^{\div} \wedge \\
 & & \forall k.\ (k,\gamma,\gamma) \in \mathcal{G}^{+}[\![\Gamma]\!] \implies \\
 & & (k,\gamma(e),\gamma(e)) \in \mathcal{E}^{\div}[\![\sigma]\!]
\end{array}
$$

Figure 9. Cross-Language Logical Relation for λ^{ST}

$\cdot \vdash C[e_1] : \sigma' \leadsto_e \mathbf{C}[e_1]$ and $\cdot \vdash C[e_2] : \sigma' \leadsto_e \mathbf{C}[e_2]$. Then by Theorem 1.1, $C[e_1] \Updownarrow \mathbf{C}[e_1]$ and $C[e_2] \Updownarrow \mathbf{C}[e_2]$. Finally since $\cdot;\Gamma^{+} \vdash e_1 \approx^{ctx}_{\mathbf{T}} e_2 : \sigma^{\div}$, $\mathbf{C}[e_1] \Updownarrow \mathbf{C}[e_2]$ giving:

$$C[e_1] \Updownarrow \mathbf{C}[e_1] \Updownarrow \mathbf{C}[e_2] \Updownarrow C[e_2] \qquad \square$$

5. Back-Translation and Equivalence Preservation

To prove equivalence preservation, it is sufficient to be able to "back-translate" a target context to an equivalent source context. To see why, suppose that $\Gamma \vdash e_1 : \sigma \leadsto_e \mathbf{e_1}$, $\Gamma \vdash e_2 : \sigma \leadsto_e \mathbf{e_2}$ and $\Gamma \vdash e_1 \approx^{ctx}_{\mathsf{S}} e_2 : \sigma$ and that we want to show $\cdot;\Gamma^{+} \vdash \mathbf{e_1} \approx^{ctx}_{\mathbf{T}} \mathbf{e_2} : \sigma^{\div}$. Given a target context $\mathbf{C}$ such that $\mathbf{C}[\mathbf{e_1}] \Downarrow$, we want to come up with a source context C such that $\mathbf{C}[\mathbf{e_1}] \Updownarrow C[e_1]$ and $\mathbf{C}[\mathbf{e_2}] \Updownarrow C[e_2]$. Since e_1 and e_2 are equivalent we know that $C[e_1] \Updownarrow C[e_2]$, so we can conclude that $\mathbf{C}[\mathbf{e_2}] \Downarrow$.

However, there are several differences between the source and target languages that make this difficult. First, the target has exceptions while the source does not. Second, the target has polymorphic functions, which cannot be precisely typed in the source language.

While the target language has features that are untypeable in the source language, the dynamics of the target language can clearly be simulated in the source language. We define the type of untyped lambda terms via a universal type [34] that encodes all values of the source and target languages: unit, sum, product, function and recursive type:

$$
\begin{array}{rcl}
\mathsf{U} & \stackrel{\text{def}}{=} & \mu\alpha.1 + (\alpha + \alpha) + (\alpha \times \alpha) + (\alpha \to \mathsf{R}(\alpha)) + \alpha \\
\mathsf{R}(\alpha) & \stackrel{\text{def}}{=} & \alpha + \alpha
\end{array}
$$

The type $\mathsf{R}(\alpha)$ is the parameterized type of *results* in the target language, i.e., either a successful or exceptional value. We write R to mean the closed version of this type, $\mathsf{R}(\alpha)[\mathsf{U}/\alpha]$.

This encoding allows us to back-translate target terms into the source language, but it is not enough to ensure full abstraction. Note

that back-translating a term of type $\sigma \div$ produces a term of type R, not a term of type σ.

However, R is truly a universal type in the algebraic sense (see Longley [22] for a more thorough introduction), that is every type σ can be embedded in R via a function we denote $\text{EMBED}(\sigma)$ and there is a partial function that selects the elements of σ from R which we denote $\text{PROJECT}(\sigma)$. These two form a *retract* which means that $\text{PROJECT}(\sigma)(\text{EMBED}(\sigma)x) \approx^{ctx}_{S} x$.

This property is perfectly analogous with one case of Lemma 3.2, that is ${}^{\sigma}\mathcal{ST}\,\mathcal{TS}^{\sigma}\, x \approx^{ctx}_{S} x$, and indeed our full abstraction proof relies on a formal correspondence between the embedding-projection pairs and the multi-language boundaries.

More precisely, when we want to prove that equivalence is preserved, we start with a target context C interacting with the translation of a source term e, which is equivalent under the multi-language semantics to a target context interacting with the source term, mediated by the language boundaries ($\mathcal{ST}, \mathcal{TS}$).

We want a source context C whose interaction with e is the same as that of C. To this end we backtranslate the target context to a source context at universal type and replace the language boundaries ($\mathcal{TS}$,$\mathcal{ST}$) with the universal type boundaries ($\text{EMBED}(\sigma), \text{PROJECT}(\sigma)$).

The crucial detail is then a proof of correspondence between these two types of boundaries: Formally stated in Lemma 5.4 and Lemma 5.5, we essentially prove that if a context C with hole of type $\sigma \div$ and return type $\sigma' \div$ is back translated to C with hole of type R and return type $\sigma' \div$, then

$$\text{PROJECT}(\sigma')\ (\mathsf{C}\,[\text{EMBED}(\sigma)\ [\cdot]]) \approx {}^{\sigma'}\mathcal{ST}\ (\mathsf{C}\,[\mathcal{TS}^{\sigma}\ [\cdot]])$$

5.1 Back-Translation, Formally

We present a portion of the back-translation in Figure 10. To enhance readability, we define metafunctions that construct terms of universal type via the appropriate application of $\mathsf{fold}_{\mathsf{U}}$ and inj_i. For instance, $\text{UNIT} = \mathsf{fold}_{\mathsf{U}}\ (\mathsf{inj}_1\ \langle\rangle)$. The elimination metafunctions and $\text{PROJECT}()$ also handle failure since the sub-language is untyped. The metafunction $\text{TOLHS}()$ is one such metafunction which returns the expected value if back-translation succeeds, and fails otherwise. For example, in the metafunction $\text{TOFUN}(\mathsf{e}')$, if the term e' represents a function, it is applied, otherwise it results in failure:

$$\mathsf{let}\ x = \text{TOFUN}(\text{LAMBDA}(\lambda(y : \mathsf{U}).\, \mathsf{e}))\ \mathsf{in}\ x\ v \longmapsto^{*} \mathsf{e}[v/y]$$
$$\mathsf{let}\ x = \text{TOFUN}(\text{UNIT})\ \mathsf{in}\ x\ v \longmapsto^{*} \mathcal{U}$$

where $\mathcal{U}$ represents undefined behavior. Since our back-translation only produces well-behaved terms, our development is independent of what $\mathcal{U}$ means. [1]

We back-translate source terms to themselves, target terms to constructors for the universal type, and boundaries to $\text{EMBED}()$ and $\text{PROJECT}()$. While some types, like sums, can be directly simulated in the source, others must be encoded. For instance, tuples are back-translated to cons-lists. Crucially, all polymorphism is erased, using dynamic typing to encode the behavior of the polymorphic terms of the target language. Finally, we preserve the **fold** constructor for formal convenience.

In Figure 11, we define $\text{EMBED}()$ and $\text{PROJECT}()$ for each type. The code is fairly ugly since we are programming in a bare-bones calculus in monadic normal form, but its meaning is straightforward and follows the multi-language boundaries.

At the top level we have $\text{EMBED}(\sigma)$ and $\text{PROJECT}(\sigma)$ which translate between R and σ. These deal with the exception mechanisms of the target language. As in the multi-language boundary, no exception should be thrown from target to source or vice-versa. Thus, $\text{EMBED}(\sigma)$ uses the $\text{RETURN}()$ function to return a successful value

[1] In our proofs, we model failure via divergence since it can be defined uniformly for any type.

If $\Delta;\Gamma \vdash \mathsf{e} : \sigma$, then $\Gamma^{\rightarrow} \vdash \mathsf{e}^{\rightarrow} : \sigma$	If $\Delta;\Gamma \vdash \mathbf{v} : \boldsymbol{\tau}$, then $\Gamma^{\rightarrow} \vdash \mathbf{v}^{\rightarrow} : \mathsf{U}$
If $\Delta;\Gamma \vdash \mathbf{e} : \boldsymbol{\theta}$, then $\Gamma^{\rightarrow} \vdash \mathbf{e}^{\rightarrow} : \mathsf{R}$	

$$\begin{aligned}
\cdot^{\rightarrow} &\stackrel{\text{def}}{=} \cdot \\
(\Gamma, x : \sigma)^{\rightarrow} &\stackrel{\text{def}}{=} \Gamma^{\rightarrow}, x : \sigma \\
(\Gamma, \mathbf{y} : \boldsymbol{\tau})^{\rightarrow} &\stackrel{\text{def}}{=} \Gamma^{\rightarrow}, y : \mathsf{U} \\
x^{\rightarrow} &\stackrel{\text{def}}{=} x \\
(\mathsf{let}\ x = \mathsf{e}\ \mathsf{in}\ \mathsf{e}')^{\rightarrow} &\stackrel{\text{def}}{=} \mathsf{let}\ x = \mathsf{e}^{\rightarrow}\ \mathsf{in}\ \mathsf{e}'^{\rightarrow} \\
&\vdots \\
({}^{\sigma}\mathcal{ST}\,\mathsf{e})^{\rightarrow} &\stackrel{\text{def}}{=} \mathsf{let}\ x = \mathsf{e}^{\rightarrow}\ \mathsf{in}\ \text{PROJECT}(\sigma)\ x \\
\mathbf{y}^{\rightarrow} &\stackrel{\text{def}}{=} y \\
\langle\rangle^{\rightarrow} &\stackrel{\text{def}}{=} \text{UNIT} \\
\langle \mathbf{v}_1, \ldots, \mathbf{v}_n \rangle^{\rightarrow} &\stackrel{\text{def}}{=} \text{CONS}(\mathbf{v}_1{}^{\rightarrow}, \langle \ldots, \mathbf{v}_n \rangle^{\rightarrow}) \\
(\boldsymbol{\lambda}\,[\boldsymbol{\alpha}]\,(\mathbf{y} : \boldsymbol{\tau}).\,\mathbf{e})^{\rightarrow} &\stackrel{\text{def}}{=} \text{LAMBDA}(\lambda(y : \mathsf{U}).\,\mathbf{e}^{\rightarrow}) \\
(\mathbf{pack}(\boldsymbol{\tau}', \mathbf{v}))^{\rightarrow} &\stackrel{\text{def}}{=} \mathbf{v}^{\rightarrow} \\
(\mathbf{fold}_{\mu\alpha.\tau}\ \mathbf{v})^{\rightarrow} &\stackrel{\text{def}}{=} \text{FOLD}(\mathbf{v}^{\rightarrow}) \\
(\mathbf{return}\ \mathbf{v})^{\rightarrow} &\stackrel{\text{def}}{=} \text{RETURN}(\mathbf{v}^{\rightarrow}) \\
(\mathbf{raise}\ \mathbf{v})^{\rightarrow} &\stackrel{\text{def}}{=} \text{RAISE}(\mathbf{v}^{\rightarrow}) \\
&\vdots \\
(\mathbf{v}.\mathbf{i})^{\rightarrow} &\stackrel{\text{def}}{=} \mathsf{let}\ x = \text{PRJ}(i, \mathbf{v}^{\rightarrow})\ \mathsf{in}\ \text{RETURN}(x) \\
(\mathbf{v_1}\ \mathbf{v_2})^{\rightarrow} &\stackrel{\text{def}}{=} \mathsf{let}\ x = \text{TOFUN}(\mathbf{v_1}^{\rightarrow})\ \mathsf{in}\ x\ \mathbf{v_2}^{\rightarrow} \\
(\mathbf{unpack}(\boldsymbol{\alpha}, \mathbf{y}) = \mathbf{v}\ \mathbf{in}\ \mathbf{e})^{\rightarrow} &\stackrel{\text{def}}{=} \mathsf{let}\ y = \mathbf{v}^{\rightarrow}\ \mathsf{in}\ \mathbf{e} \\
(\mathbf{unfold}\ \mathbf{v})^{\rightarrow} &\stackrel{\text{def}}{=} \mathsf{let}\ x = \text{TOFOLD}(\mathbf{v}^{\rightarrow})\ \mathsf{in}\ \text{RETURN}(x) \\
(\mathbf{handle}\ \mathbf{e}\ \mathbf{with} & \stackrel{\text{def}}{=} \mathsf{let}\ x_r = \mathbf{e}^{\rightarrow}\ \mathsf{in}\ \mathsf{case}\ x_r \\
\quad (\mathbf{x_1}.\mathbf{e_1}) & \qquad\qquad x_1.\mathbf{e_1}^{\rightarrow} \\
\quad (\mathbf{x_2}.\mathbf{e_2}))^{\rightarrow} & \qquad\qquad x_2.\mathbf{e_2}^{\rightarrow} \\
&\vdots \\
(\mathcal{TS}^{\sigma}\,\mathsf{e})^{\rightarrow} &\stackrel{\text{def}}{=} \mathsf{let}\ x = \mathsf{e}^{\rightarrow}\ \mathsf{in}\ \text{EMBED}(\sigma)\ x
\end{aligned}$$

Figure 10. Back-Translation (excerpts)

and $\text{PROJECT}(\sigma)$ uses the $\text{TOLHS}()$ function to handle any possible exceptions. If it is a successful value of type R then this is further projected but if it is an exception then $\text{TOLHS}()$ diverges.

These functions are defined in terms of $\text{EMBED}(\delta, \sigma)$ and $\text{PROJECT}(\delta, \sigma)$, which mediate between σ and U, the type of values in the target language.

Since we have higher-order recursive types, we must define $\text{EMBED}(\delta, \sigma)$ and $\text{PROJECT}(\delta, \sigma)$ mutually recursively. This is the purpose of the δ parameter. It maps free type variables to the recursive type that they represent and the recursive binding to the thunk that yields the embedding-projection pair for that type when forced. The recursive process is started in Figure 12 which uses a standard call-by-value fixed point combinator (definition elided) to define two mutually recursive functions.

For function types, $\text{EMBED}(\delta, \sigma_1 \to \sigma_2)$ packs a function with an empty environment and $\text{PROJECT}(\delta, \sigma_1 \to \sigma_2)$ unpacks a closure and applies the function to its environment.

Pairs are straightforward, handling the fact that a two-tuple in the target is a cons list of length two. Sums and the unit type are even more straightforward so we elide them.

5.2 Correctness of Back-Translation

The correctness theorem for the back-translation mirrors Theorem 1.1 (semantics preservation)—that is, back-translation of e is equivalent to placing appropriate boundaries around e. However the back-translation thus far uses the universal type, since we back-translate all target language terms and not just those that are of translation type. So we get a more refined theorem, one that says

$\emptyset \vdash \text{EMBED}(\sigma) : \sigma \to \mathsf{R}$ | $\emptyset \vdash \text{PROJECT}(\sigma) : \mathsf{R} \to \sigma$

$\delta_\Gamma \vdash \text{EMBED}(\delta, \sigma) : \delta_\sigma(\sigma) \to \mathsf{U}$ | $\delta_\Gamma \vdash \text{PROJECT}(\delta, \sigma) : \mathsf{U} \to \delta_\sigma(\sigma)$

$$\text{EMBED}(\sigma) \stackrel{\text{def}}{=} \lambda(\mathsf{x} : \sigma).\ \mathsf{let}\ \mathsf{x}_u = \text{EMBED}(\emptyset, \sigma)\ \mathsf{x}\ \mathsf{in}\ \text{RETURN}(\mathsf{x}_u)$$

$$\text{PROJECT}(\sigma) \stackrel{\text{def}}{=} \lambda(\mathsf{x}_r : \mathsf{R}).\ \mathsf{let}\ \mathsf{x}_u = \text{TOLHS}(\mathsf{x}_r)\ \mathsf{in}\ \text{PROJECT}(\emptyset, \sigma)\ \mathsf{x}_u$$

$$\text{EMBED}(\delta, \sigma_1 \times \sigma_2) \stackrel{\text{def}}{=} \lambda(\mathsf{x} : \delta_\sigma(\sigma_1 \times \sigma_2)).\ \mathsf{let}\ \mathsf{x}_1 = \pi_1\ \mathsf{x}\ \mathsf{in}\ \mathsf{let}\ \mathsf{x}_2 = \pi_2\ \mathsf{x}\ \mathsf{in}\ \mathsf{let}\ \mathsf{x}_1' = \text{EMBED}(\delta, \sigma_1)\ \mathsf{x}_1\ \mathsf{in}\ \mathsf{let}\ \mathsf{x}_2' = \text{EMBED}(\delta, \sigma_2)\ \mathsf{x}_2\ \mathsf{in}\ \text{CONS}(\mathsf{x}_1', \text{CONS}(\mathsf{x}_2', \text{UNIT}))$$

$$\text{EMBED}(\delta, \sigma_1 \to \sigma_2) \stackrel{\text{def}}{=} \lambda(\mathsf{x}_f : \delta_\sigma(\sigma_1 \to \sigma_2)).\ \mathsf{let}\ \mathsf{x}_f{}' = \lambda(\mathsf{x}_u : \mathsf{U}).\ \mathsf{in}\ \mathsf{let}\ \mathsf{x}_u{}' = \text{PRJ}(2, \mathsf{x}_u)\ \mathsf{in}\ \mathsf{let}\ \mathsf{x} = \text{PROJECT}(\delta, \sigma_1)\ \mathsf{x}_u{}'\ \mathsf{in}\ \mathsf{let}\ \mathsf{y} = \mathsf{x}_f\ \mathsf{x}\ \mathsf{in}\ \mathsf{let}\ \mathsf{x}_u{}'' = \text{EMBED}(\delta, \sigma_2)\ \mathsf{y}\ \mathsf{in}\ \text{RETURN}(\mathsf{x}_u{}'')\ \text{CONS}(\mathsf{x}_f{}', \text{CONS}(\text{UNIT}, \text{UNIT}))$$

$$\text{EMBED}(\delta, \alpha) \stackrel{\text{def}}{=} \lambda(\mathsf{x} : \delta_\sigma(\alpha)).\ \mathsf{let}\ \mathsf{x}_{ep} = \delta_\mathsf{x}(\alpha)\ \langle\rangle\ \mathsf{in}\ \mathsf{let}\ \mathsf{x}_{embed} = \pi_1\ \mathsf{x}_{ep}\ \mathsf{in}\ \mathsf{x}_{embed}\ \mathsf{x}$$

$$\text{EMBED}(\delta, \mu\alpha.\sigma) \stackrel{\text{def}}{=} \lambda(\mathsf{x} : \delta_\sigma(\mu\alpha.\sigma)).\ \mathsf{let}\ \mathsf{x}_{ep} = \text{EP}(\delta, \mu\alpha.\sigma)\ \langle\rangle\ \mathsf{in}\ \mathsf{let}\ \mathsf{x}_{embed} = \pi_1\ \mathsf{x}_{ep}\ \mathsf{in}\ \mathsf{x}_{embed}\ \mathsf{x}$$

$$\text{PROJECT}(\delta, \sigma_1 \times \sigma_2) \stackrel{\text{def}}{=} \lambda(\mathsf{x}_u : \mathsf{U}).\ \mathsf{let}\ \mathsf{x} = \text{TOPAIR}(\mathsf{x}_u)\ \mathsf{in}\ \mathsf{let}\ \mathsf{x}_1 = \pi_1\ \mathsf{x}\ \mathsf{in}\ \mathsf{let}\ \mathsf{x}_1' = \text{PROJECT}(\delta, \sigma_1)\ \mathsf{x}_1\ \mathsf{in}\ \mathsf{let}\ \mathsf{y} = \pi_2\ \mathsf{x}\ \mathsf{in}\ \mathsf{let}\ \mathsf{y}' = \text{TOPAIR}(\mathsf{y})\ \mathsf{in}\ \mathsf{let}\ \mathsf{x}_2 = \pi_1\ \mathsf{y}'\ \mathsf{in}\ \mathsf{let}\ \mathsf{x}_2' = \text{PROJECT}(\delta, \sigma_2)\ \mathsf{x}_2\ \mathsf{in}\ \langle \mathsf{x}_1', \mathsf{x}_2' \rangle$$

$$\text{PROJECT}(\delta, \sigma_1 \to \sigma_2) \stackrel{\text{def}}{=} \lambda(\mathsf{x}_u : \mathsf{U}).\ \mathsf{let}\ \mathsf{x}_u{}' = \text{TOPAIR}(\mathsf{x}_u)\ \mathsf{in}\ \mathsf{let}\ \mathsf{x}_f = \text{PRJ}(1, \mathsf{x}_u{}')\ \mathsf{in}\ \mathsf{let}\ \mathsf{x}_{env} = \text{PRJ}(2, \mathsf{x}_u{}')\ \mathsf{in}\ \lambda(\mathsf{y} : \delta_\sigma(\sigma_1)).\ \mathsf{let}\ \mathsf{y}_u = \text{EMBED}(\delta, \sigma_1)\ \mathsf{y}\ \mathsf{in}\ \mathsf{let}\ \mathsf{x} = \text{CONS}(\mathsf{x}_{env}, \text{CONS}(\mathsf{y}_u, \text{UNIT}))\ \mathsf{in}\ \mathsf{let}\ \mathsf{x}_r = \mathsf{x}_f\ \mathsf{x}\ \mathsf{in}\ \mathsf{let}\ \mathsf{x}_u{}'' = \text{TOLHS}(\mathsf{x}_r)\ \mathsf{in}\ \text{PROJECT}(\delta, \sigma_2)\ \mathsf{x}_u{}''$$

$$\text{PROJECT}(\delta, \alpha) \stackrel{\text{def}}{=} \lambda(\mathsf{x}_u : \mathsf{U}).\ \mathsf{let}\ \mathsf{x} = \delta_\mathsf{x}(\alpha)\ \langle\rangle\ \mathsf{in}\ \mathsf{let}\ \mathsf{x}_f = \pi_2\ \mathsf{x}\ \mathsf{in}\ \mathsf{x}'\ \mathsf{x}$$

$$\text{PROJECT}(\delta, \mu\alpha.\sigma) \stackrel{\text{def}}{=} \lambda(\mathsf{x}_u : \mathsf{U}).\ \mathsf{let}\ \mathsf{x} = \text{EP}(\delta, \mu\alpha.\sigma)\ \langle\rangle\ \mathsf{in}\ \mathsf{let}\ \mathsf{x}_f = \pi_2\ \mathsf{x}\ \mathsf{in}\ \mathsf{x}_f\ \mathsf{x}_u$$

Figure 11. Embedding and Projection Functions

that a target term and its back-translation are indistinguishable in their *interactions with source programs*.

Programs of universal type interact with programs of source type mediated by EMBED() and PROJECT() just as multi-language interactions are mediated by boundaries. For closed programs we prove that they satisfy the following equivalence:

Definition 5.1 (Universal Type Equivalence for Closed Programs)
If $\cdot;\cdot \vdash \mathbf{e} : \sigma^{\div}$ *and* $\cdot \vdash \mathsf{e}_u : \mathsf{R}$, *then*

$$\cdot \vdash \mathsf{e}_u \ {}^{u}\!\simeq^{\tau}\ \mathbf{e} : \sigma \stackrel{\text{def}}{=} \cdot;\cdot \vdash \text{PROJECT}(\sigma) \circ \mathsf{e}_u \approx^{ctx}_{\text{ST}}{}^{\sigma}\ \mathcal{ST}\,\mathbf{e} : \sigma$$

where

$$\text{PROJECT}(\sigma) \circ \mathsf{e}_u \stackrel{\text{def}}{=} \mathsf{let}\ \mathsf{x}_u = \mathsf{e}_u\ \mathsf{in}\ \text{PROJECT}(\sigma)\ \mathsf{x}_u$$

$$\delta_\Gamma \vdash \text{EP}(\delta, \mu\alpha.\sigma) : 1 \to ((\delta_\sigma(\mu\alpha.\sigma) \to \mathsf{U}) \times (\mathsf{U} \to \delta_\sigma(\mu\alpha.\sigma)))$$

$$\text{EP}(\delta, \mu\alpha.\sigma) \stackrel{\text{def}}{=}$$

$$\begin{array}{l}
\text{FIX}_{1 \to ((\delta_\sigma(\mu\alpha.\sigma) \to \mathsf{U}) \times (\mathsf{U} \to \delta_\sigma(\mu\alpha.\sigma)))} \\
\ \lambda(\mathsf{x}_{\mu\alpha.\sigma} : 1 \to ((\delta_\sigma(\mu\alpha.\sigma) \to \mathsf{U}) \times (\mathsf{U} \to \delta_\sigma(\mu\alpha.\sigma)))). \\
\quad \lambda(\mathsf{x}_{unit} : 1). \\
\quad\quad \mathsf{let}\ \mathsf{x}_{embed} = \\
\quad\quad\ \lambda(\mathsf{x} : \delta_\sigma(\mu\alpha.\sigma)).\ \mathsf{in} \\
\quad\quad\quad \mathsf{let}\ \mathsf{y} = \mathsf{unfold}\ \mathsf{x}\ \mathsf{in} \\
\quad\quad\quad \mathsf{let}\ \mathsf{y}_u = \text{EMBED}(\delta[\alpha \mapsto \mu\alpha.\sigma, \mathsf{x}_{\mu\alpha.\sigma}], \sigma)\ \mathsf{y}\ \mathsf{in} \\
\quad\quad\quad \text{FOLD}(\mathsf{y}_u) \\
\quad\quad \mathsf{let}\ \mathsf{x}_{project} = \\
\quad\quad\ \lambda(\mathsf{x}_u : \mathsf{U}).\ \mathsf{in} \\
\quad\quad\quad \mathsf{let}\ \mathsf{y}_u = \text{TOFOLD}(\mathsf{x}_u)\ \mathsf{in} \\
\quad\quad\quad \mathsf{let}\ \mathsf{y} = \text{PROJECT}(\delta[\alpha \mapsto \mu\alpha.\sigma, \mathsf{x}_{\mu\alpha.\sigma}], \sigma)\ \mathsf{y}_u\ \mathsf{in} \\
\quad\quad\quad \mathsf{fold}_{\mu\alpha.\sigma}\ \mathsf{y} \\
\quad\quad \langle \mathsf{x}_{embed}, \mathsf{x}_{project} \rangle
\end{array}$$

Figure 12. Embedding-Projection Pair at Recursive Type

To define equivalence for open programs, we need to define how to *embed* all the free variables from source to universal type just as we had to extend the boundary to act on the free variables. Note that here we use the version of EMBED() whose return type is U. The asymmetry here is because we are in a call-by-value language. We then arrive at the following definition for open programs

Definition 5.2 (Universal Type Equivalence)
If $\cdot;\Gamma^{+} \vdash \mathbf{e} : \sigma^{\div}$ *and* $\Gamma^{+\to} \vdash \mathsf{e}_u : \mathsf{R}$, *and* $\Gamma = \mathsf{x}_1 : \sigma_1, \ldots, \mathsf{x}_n : \sigma_n$, *we define*

$$\Gamma \vdash \mathsf{e}_u \ {}^{u}\!\simeq^{\tau}\ \mathbf{e} : \sigma \stackrel{\text{def}}{=} \cdot;\Gamma \vdash \text{PROJECT}(\sigma) \circ \mathsf{e}_u \circ \text{EMBED}(\Gamma) \approx^{ctx}_{\text{ST}}{}^{\sigma}\ \mathcal{ST}\,\mathbf{e}\,\mathcal{TS}^{\Gamma} : \sigma$$

where

$$\mathsf{e}_u \circ \text{EMBED}(\Gamma) \stackrel{\text{def}}{=} \begin{array}{l} \mathsf{let}\ \mathsf{x}_{u1} = \text{EMBED}(\emptyset, \sigma_1)\ \mathsf{x}_1\ \mathsf{in} \\ \quad\vdots \\ \mathsf{let}\ \mathsf{x}_n = \text{EMBED}(\emptyset, \sigma_n)\ \mathsf{x}_n\ \mathsf{in}\ \mathsf{e}_u \end{array}$$

In order to have an inductive invariant strong enough to prove back-translation is correct, we define a cross-language logical relation to formalize when a term of universal type represents an *arbitrary* target term, rather than just a target term of translation type. In Figure 13 we define a step-indexed biorthogonal cross-language logical relation that relates the typed target language to the untyped embedding in the source language.

As usual, we lift the relation on closed terms to a relation on open terms via closing substitution. However, recall that we define the back-translation for the entire multi-language and not just the target language. While the logical relation on closed terms only relates source terms to target terms, we must extend the relation on open terms to relate source terms to source terms. The definition for source terms merely appeals to the multi-language logical relation defined in §4.

The fundamental property of this logical relation, Theorem 5.3, gives us that every term is related to its back-translation. As in §4, we require a form of bridge lemma, Lemma 5.4, which also connects our two notions of logical equivalence, namely the embedding interpretation and the multi-language interoperability semantics.

Theorem 5.3 (Fundamental Property)

1. *If* $\Delta;\Gamma \vdash \mathsf{e} : \sigma$ *then* $\Delta;\Gamma \vdash \mathsf{e}^{\to} \approx^{log}_{\mathcal{E}^{\mathsf{U}}} \mathsf{e} : \sigma$.
2. *If* $\Delta;\Gamma \vdash \mathbf{e} : \theta$ *then* $\Delta;\Gamma \vdash \mathbf{e}^{\to} \approx^{log}_{\mathcal{E}^{\mathsf{U}}} \mathbf{e} : \theta$.

Lemma 5.4 (Interpret = Interoperate)

Note: all relations restricted to well-typed member.

$$
\begin{array}{rcl}
\mathrm{Rel}^{\mathsf{U}}[\tau] & \stackrel{\text{def}}{=} & \{R \mid \forall j \le k, \mathsf{v}, \mathbf{v} \,.\, \cdot;\cdot \vdash \mathsf{v} : \mathsf{U} \wedge \cdot;\cdot \vdash \mathbf{v} : \tau \wedge (k,\mathsf{v},\mathbf{v}) \in R \implies (j,\mathsf{v},\mathbf{v}) \in R\} \\
\mathcal{V}^{\mathsf{U}}[\![\mathbf{0}]\!]\rho^{\mathsf{U}} & \stackrel{\text{def}}{=} & \emptyset \\
\mathcal{V}^{\mathsf{U}}[\![\langle\rangle]\!]\rho^{\mathsf{U}} & \stackrel{\text{def}}{=} & \{(k, \underline{\text{UNIT}}, \langle\rangle)\} \\
\mathcal{V}^{\mathsf{U}}[\![\langle\tau_1,\tau_2,\ldots,\tau_n\rangle]\!]\rho^{\mathsf{U}} & \stackrel{\text{def}}{=} & \{(k, \underline{\text{CONS}}(\mathsf{v}_u,\mathsf{v}_u'), \langle\mathbf{v}_1,\mathbf{v}_2,\ldots,\mathbf{v}_n\rangle) \mid (k,\mathsf{v}_u,\mathbf{v}_1) \in \mathcal{V}^{\mathsf{U}}[\![\tau_1]\!]\rho^{\mathsf{U}} \wedge (k,\mathsf{v}_u',\langle\mathbf{v}_2,\ldots,\mathbf{v}_n\rangle) \in \mathcal{V}^{\mathsf{U}}[\![\langle\tau_2,\ldots,\tau_n\rangle]\!]\rho^{\mathsf{U}}\} \\
\mathcal{V}^{\mathsf{U}}[\![\tau_1+\tau_2]\!]\rho^{\mathsf{U}} & \stackrel{\text{def}}{=} & \{(k, \underline{\text{IN}}(\mathsf{i},\mathsf{v}_u), \mathbf{inj_i}\,\mathbf{v}) \mid (k,\mathsf{v}_u,\mathbf{v}) \in \mathcal{V}^{\mathsf{U}}[\![\tau_\mathbf{i}]\!]\rho^{\mathsf{U}}\} \\
\mathcal{V}^{\mathsf{U}}[\![\alpha]\!]\rho^{\mathsf{U}} & \stackrel{\text{def}}{=} & \rho_R^{\mathsf{U}}(\alpha) \\
\mathcal{V}^{\mathsf{U}}[\![\forall[\alpha].\tau\to\theta]\!]\rho^{\mathsf{U}} & \stackrel{\text{def}}{=} & \left\{ \begin{array}{l} (k, \underline{\text{LAMBDA}}(\lambda(\mathsf{x}_u:\mathsf{U}).\mathsf{e}_u), \lambda[\alpha](\mathbf{x}:\tau).\mathbf{e}) \mid \forall \tau', R \in \mathrm{Rel}^{\mathsf{U}}[\rho^{\mathsf{U}}(\tau')], j \le k, (j,\mathsf{v}_u,\mathbf{v}) \in \mathcal{V}^{\mathsf{U}}[\![\tau]\!]\rho^{\mathsf{U}\prime}. \\ \quad (j,\mathsf{e}_u[\mathsf{v}_u/\mathsf{x}_u], \mathbf{e}[\tau'/\alpha][\mathbf{v}/\mathbf{x}]) \in \mathcal{E}^{\mathsf{U}}[\![\theta]\!]\rho^{\mathsf{U}\prime} \text{ where } \rho^{\mathsf{U}\prime} = \rho^{\mathsf{U}}[\alpha \mapsto \tau', R] \end{array} \right\} \\
\mathcal{V}^{\mathsf{U}}[\![\exists\alpha.\tau]\!]\rho^{\mathsf{U}} & \stackrel{\text{def}}{=} & \{(k,\mathsf{v}_u,\mathbf{pack}(\tau',\mathbf{v})) \mid \exists R \in \mathrm{Rel}^{\mathsf{U}}[\rho^{\mathsf{U}}(\tau')].\ (k,\mathsf{v}_u,\mathbf{v}) \in \mathcal{V}^{\mathsf{U}}[\![\tau]\!]\rho^{\mathsf{U}}[\alpha \mapsto \tau', R]\} \\
\mathcal{V}^{\mathsf{U}}[\![\mu\alpha.\tau]\!]\rho^{\mathsf{U}} & \stackrel{\text{def}}{=} & \{(0,\mathsf{v}_u,\mathbf{v})\} \cup \{(k+1, \underline{\text{FOLD}}(\mathsf{v}_u), \mathbf{fold}_{\rho^{\mathsf{U}}(\mu\alpha.\tau)}\,\mathbf{v}) \mid (k,\mathsf{v}_u,\mathbf{v}) \in \mathcal{V}^{\mathsf{U}}[\![\tau[\mu\alpha.\tau/\alpha]]\!]\rho^{\mathsf{U}}\} \\
\mathcal{R}^{\mathsf{U}}[\![\mathbf{E}\,\tau_{\mathbf{exn}}\,\tau]\!]\rho^{\mathsf{U}} & \stackrel{\text{def}}{=} & \{(k, \underline{\text{RETURN}}(\mathsf{v}_u), \mathbf{return}\,\mathbf{v}) \mid (k,\mathsf{v}_u,\mathbf{v}) \in \mathcal{V}^{\mathsf{U}}[\![\tau]\!]\rho^{\mathsf{U}}\} \cup \{(k, \underline{\text{RAISE}}(\mathsf{v}_u), \mathbf{raise}\,\mathbf{v}) \mid (k,\mathsf{v}_u,\mathbf{v}) \in \mathcal{V}^{\mathsf{U}}[\![\tau_{\mathbf{exn}}]\!]\rho^{\mathsf{U}}\} \\
\mathcal{E}^{\mathsf{U}}[\![\theta]\!]\rho^{\mathsf{U}} & \stackrel{\text{def}}{=} & \{(k,\mathsf{e}_u,\mathbf{e}) \mid \forall j \le k, K_1, K_2.\ (j,K_1,K_2) \in \mathcal{K}[\![\theta]\!]\rho \implies (j, K_1[\mathsf{e}_u], K_2[\mathbf{e}]) \in \mathcal{O}\} \\
\mathcal{K}^{\mathsf{U}}[\![\theta]\!]\rho^{\mathsf{U}} & \stackrel{\text{def}}{=} & \{(k,K_1,K_2) \mid \forall j \le k, \mathsf{v}_u, \mathbf{r}\,.\,(j,\mathsf{v}_u,\mathbf{r}) \in \mathcal{R}[\![\theta]\!]\rho \implies (j, K_1[\mathsf{v}_u], K_2[\mathbf{r}]) \in \mathcal{O}\} \\
\mathcal{D}^{\mathsf{U}}[\![\cdot]\!] & \stackrel{\text{def}}{=} & \{\emptyset\} \\
\mathcal{D}^{\mathsf{U}}[\![\Delta,\alpha]\!] & \stackrel{\text{def}}{=} & \{\rho^{\mathsf{U}}[\alpha \mapsto \tau, R] \mid \rho^{\mathsf{U}} \in \mathcal{D}[\![\Delta]\!] \wedge R \in \mathrm{Rel}^{\mathsf{U}}[\tau]\} \\
\mathcal{G}^{\mathsf{U}}[\![\cdot]\!]\rho^{\mathsf{U}} & \stackrel{\text{def}}{=} & \{(k,\emptyset) \mid k \in \mathbb{N}\} \\
\mathcal{G}^{\mathsf{U}}[\![\Gamma,\mathsf{x}:\sigma]\!]\rho^{\mathsf{U}} & \stackrel{\text{def}}{=} & \{(k,\gamma^{\mathsf{U}}[\mathsf{x} \mapsto \mathsf{v}_1,\mathsf{v}_2]) \mid (k,\gamma^{\mathsf{U}}) \in \mathcal{G}^{\mathsf{U}}[\![\Gamma]\!]\rho^{\mathsf{U}} \wedge (k,\mathsf{v}_1,\mathsf{v}_2) \in \mathcal{V}[\![\sigma]\!]\emptyset\} \\
\mathcal{G}^{\mathsf{U}}[\![\Gamma,\mathbf{x}:\tau]\!]\rho^{\mathsf{U}} & \stackrel{\text{def}}{=} & \{(k,\gamma^{\mathsf{U}}[\mathbf{x} \mapsto \mathbf{v}][\mathsf{x}_u \mapsto \mathsf{v}_u]) \mid (k,\gamma^{\mathsf{U}}) \in \mathcal{G}^{\mathsf{U}}[\![\Gamma]\!]\rho^{\mathsf{U}} \wedge (k,\mathsf{v}_u,\mathbf{v}) \in \mathcal{V}^{\mathsf{U}}[\![\tau]\!]\rho^{\mathsf{U}}\} \\
\Delta;\Gamma \vdash \mathbf{e}' \approx^{log}_{\mathcal{E}^{\mathsf{U}}} \mathbf{e} : \sigma & \stackrel{\text{def}}{=} & \mathbf{e}' \in \lambda^{\mathrm{S}} \wedge \forall \rho^{\mathsf{U}} \in \mathcal{D}^{\mathsf{U}}[\![\Delta]\!], (k,\gamma^{\mathsf{U}}) \in \mathcal{G}^{\mathsf{U}}[\![\Gamma]\!]\rho^{\mathsf{U}}.\ (k,\gamma^{\mathsf{U}}(\mathbf{e}'),\rho^{\mathsf{U}}(\gamma^{\mathsf{U}}(\mathbf{e}))) \in \mathcal{E}[\![\sigma]\!]\emptyset \\
\Delta;\Gamma \vdash \mathsf{e}_u \approx^{log}_{\mathcal{E}^{\mathsf{U}}} \mathbf{e} : \theta & \stackrel{\text{def}}{=} & \mathsf{e}_u \in \lambda^{\mathrm{S}} \wedge \forall \rho^{\mathsf{U}} \in \mathcal{D}^{\mathsf{U}}[\![\Delta]\!], (k,\gamma^{\mathsf{U}}) \in \mathcal{G}^{\mathsf{U}}[\![\Gamma]\!]\rho^{\mathsf{U}}.\ (k,\gamma^{\mathsf{U}}(\mathsf{e}_u),\rho^{\mathsf{U}}(\gamma^{\mathsf{U}}(\mathbf{e}))) \in \mathcal{E}^{\mathsf{U}}[\![\theta]\!]\rho^{\mathsf{U}}
\end{array}
$$

Figure 13. Universal Type Logical Relation

1. *If* $\Delta;\Gamma \vdash \mathsf{e}_u \approx^{log}_{\mathcal{E}^{\mathsf{U}}} \mathbf{e} : \sigma\,\dot{\div}$, *then*
   $\Delta;\Gamma \vdash \underline{\text{PROJECT}}(\sigma) \circ \mathsf{e}_u \approx^{log}_{\mathcal{E}^{\mathsf{U}}} {}^{\sigma}\mathcal{ST}\,\mathbf{e} : \sigma$
2. *If* $\Delta;\Gamma \vdash \mathbf{e} \approx^{log}_{\mathcal{E}^{\mathsf{U}}} \mathbf{e}' : \sigma$, *then*
   $\Delta;\Gamma \vdash \underline{\text{EMBED}}(\sigma) \circ \mathbf{e} \approx^{log}_{\mathcal{E}^{\mathsf{U}}} \mathcal{TS}^{\,\sigma}\,\mathbf{e}' : \sigma\,\dot{\div}$

As a corollary, we have the correctness theorem for our back-translation.

Corollary 5.5 (Back Translation Correctness)
If $\cdot;\Gamma^{+} \vdash \mathbf{e} : \sigma\,\dot{\div}$, *then*

$$\Gamma \vdash \mathbf{e}^{\twoheadrightarrow}\ {}^{\mathcal{U}}\!\simeq^{\mathcal{T}} \mathbf{e} : \sigma$$

Proof Let $\gamma \in \mathcal{G}[\![\Gamma]\!]\emptyset$ and $\mathsf{v}_i = \gamma(\mathsf{x}_i)$.
Since boundaries always terminate and $\underline{\text{EMBED}}()$ always terminates we can define for each $\mathsf{x}_i : \sigma_i \in \Gamma$, define $\mathbf{v_i}, \mathsf{v}_{ui}$ to be the unique values satisfying:

$$\mathcal{TS}^{\,\sigma_i}\,\mathsf{v}_i \longmapsto^* \mathbf{return}\,\mathbf{v_i}$$
$$\underline{\text{EMBED}}(\emptyset,\sigma_i)\,\mathsf{v}_i \longmapsto^* \mathsf{v}_{ui}$$

Then we define γ^{U} such that $\gamma^{\mathsf{U}}(\mathsf{x}_{ui}) = \mathsf{v}_{ui}$ and $\gamma^{\mathsf{U}}(\mathbf{x_i}) = \mathbf{v_i}$. By definition $\gamma^{\mathsf{U}} \in \mathcal{G}^{\mathsf{U}}[\![\Gamma^{+}]\!]$.
Then we have

$$\gamma({}^{\sigma}\mathcal{ST}\,\mathbf{e}\,\mathcal{TS}^{\Gamma}) \longmapsto^* {}^{\sigma}\mathcal{ST}\gamma^{\mathsf{U}}(\mathbf{e})$$
$$\gamma(\underline{\text{PROJECT}}(\sigma) \circ \mathbf{e}^{\twoheadrightarrow} \circ \underline{\text{EMBED}}(\Gamma)) \longmapsto^* \underline{\text{PROJECT}}(\sigma) \circ \gamma^{\mathsf{U}}(\mathbf{e}^{\twoheadrightarrow})$$

and by Theorem 5.3 and Lemma 5.4, for any k

$$(k, {}^{\sigma}\mathcal{ST}\gamma^{\mathsf{U}}(\mathbf{e}), \underline{\text{PROJECT}}(\sigma) \circ \gamma^{\mathsf{U}}(\mathbf{e}^{\twoheadrightarrow})) \in \mathcal{E}[\![\sigma]\!]\emptyset$$

So the two original terms are logically related since relatedness is preserved by anti-reduction. Finally by Lemma 4.3 we conclude:

$$\cdot;\Gamma \vdash \underline{\text{PROJECT}}(\sigma) \circ \mathsf{e}_u \circ \underline{\text{EMBED}}(\Gamma) \approx^{ctx}_{\mathrm{ST}} {}^{\sigma}\mathcal{ST}\,\mathbf{e}\,\mathcal{TS}^{\Gamma} : \sigma \qquad \square$$

5.3 Equivalence Preservation

We prove equivalence preservation as a consequence of the correctness of the back-translation.

In fact, we present two proofs, one *syntactic* based on translation of contexts and the other *algebraic* based on rewriting. The latter proof especially highlights the benefits of multi-language semantics, crucially using "mixed" equations to derive single-language equivalences..

For our first proof, we extend the back-translation to *contexts*. We exploit the fact that our back-translation is compositional, *i.e.*, we have that $(C[e])^{\twoheadrightarrow} = C^{\twoheadrightarrow}[e^{\twoheadrightarrow}]$, which simplifies our first equivalence preservation proof.

Theorem 1.3 (Equivalence Preservation)
If $\Gamma \vdash \mathbf{e}_1 : \sigma \rightsquigarrow_e \mathbf{e_1}$, $\Gamma \vdash \mathbf{e}_2 : \sigma \rightsquigarrow_e \mathbf{e_2}$ *and* $\Gamma \vdash \mathbf{e}_1 \approx^{ctx}_{\mathbf{S}} \mathbf{e}_2 : \sigma$, *then* $\cdot;\Gamma^{+} \vdash \mathbf{e_1} \approx^{ctx}_{\mathbf{T}} \mathbf{e_2} : \sigma\,\dot{\div}$.

Proof Suppose $\mathbf{C} \in \lambda^{\mathrm{T}}$. We proceed in two steps. First we use the compiler semantics preservation theorem to construct a multi-language context for the source terms:

$$\cdot;\Gamma^{+} \vdash \mathbf{e_1} \approx^{ctx}_{\mathrm{ST}} \mathcal{TS}^{\,\sigma}\,\mathbf{e}_1\,{}^{\Gamma}\mathcal{ST} : \sigma\,\dot{\div}.$$

Let $C = \mathbf{C}[\mathcal{TS}^{\,\sigma}[\cdot]^{\,\Gamma}\mathcal{ST}]$. Then we have $\mathbf{C}[\mathbf{e}_1] \Updownarrow C[\mathbf{e}_1]$ and similarly $C[\mathbf{e}_2] \Updownarrow \mathbf{C}[\mathbf{e_2}]$.
We now have a context for the source terms, however the source terms are equivalent in the *source language* but C is a *multi-language* context. To arrive at a fully source context, we back-translate:

and
$$(C[\mathbf{e}_1])^{\twoheadrightarrow} = C^{\twoheadrightarrow}[\mathbf{e}_1^{\twoheadrightarrow}] = C^{\twoheadrightarrow}[\mathbf{e}_1]$$
$$(C[\mathbf{e}_2])^{\twoheadrightarrow} = C^{\twoheadrightarrow}[\mathbf{e}_2]$$

Then by Theorem 5.3 we get $C[\mathbf{e}_1] \Updownarrow C^{\twoheadrightarrow}[\mathbf{e}_1]$ and $C[\mathbf{e}_2] \Updownarrow C^{\twoheadrightarrow}[\mathbf{e}_2]$
.
Finally since $\Gamma \vdash \mathbf{e}_1 \approx^{ctx}_{\mathbf{S}} \mathbf{e}_2 : \sigma$ and $C^{\twoheadrightarrow} \in \lambda^{\mathrm{S}}$, $C^{\twoheadrightarrow}[\mathbf{e}_1] \Updownarrow C^{\twoheadrightarrow}[\mathbf{e}_2]$.
This completes the chain of iffs giving us $\mathbf{C}[\mathbf{e_1}] \Updownarrow \mathbf{C}[\mathbf{e_2}]$:
$\mathbf{C}[\mathbf{e_1}] \Updownarrow C[\mathbf{e}_1] \Updownarrow C^{\twoheadrightarrow}[\mathbf{e}_1] \Updownarrow C^{\twoheadrightarrow}[\mathbf{e}_2] \Updownarrow C[\mathbf{e}_2] \Updownarrow \mathbf{C}[\mathbf{e_2}]$. $\square$

Our second proof is more abstract, but also succinct. First we define some simplifying notation. For $\Gamma \vdash \mathbf{e} : \sigma$ we define

$$[\![\mathbf{e}]\!] \stackrel{\text{def}}{=} \mathbf{e} \text{ where } \Gamma \vdash \mathbf{e} : \sigma \rightsquigarrow_e \mathbf{e}$$

For $\cdot;\Gamma^{+} \vdash \mathbf{e} : \sigma\,\dot{\div}$,

$$(\!|\mathbf{e}|\!) \stackrel{\text{def}}{=} \underline{\text{PROJECT}}(\sigma) \circ \mathbf{e}^{\twoheadrightarrow} \circ \underline{\text{EMBED}}(\Gamma)$$

These theorems give us that, up to equivalence, the back translation is a post-inverse to the compiler.

Lemma 5.7 (There and Back is Identity)
For all $\Gamma \vdash \mathsf{e} : \sigma$,

$$\Gamma \vdash \mathsf{e} \approx^{ctx}_{\mathsf{S}} (\!|[\![\mathsf{e}]\!]|\!) : \sigma$$

Proof By Theorem 1.1, Lemma 5.5, and Lemma 3.2 (eliding obvious types):

$$\begin{aligned}(\!|[\![\mathsf{e}]\!]|\!) &\approx^{ctx}_{\mathrm{ST}} \mathcal{ST}\,[\![\mathsf{e}]\!]\,\mathcal{TS} \\ &\approx^{ctx}_{\mathrm{ST}} \mathcal{ST}\,\mathcal{TS}\,\mathsf{e}\,\mathcal{ST}\,\mathcal{TS} \\ &\approx^{ctx}_{\mathrm{ST}} \mathsf{e}\end{aligned}$$

Now we have $(\!|[\![\mathsf{e}]\!]|\!) \approx^{ctx}_{\mathrm{ST}} \mathsf{e}$, which is sufficient to prove that $(\!|[\![\mathsf{e}]\!]|\!) \approx^{ctx}_{\mathsf{S}} \mathsf{e}$, since $\approx^{ctx}_{\mathrm{ST}} \subset \approx^{ctx}_{\mathsf{S}}$. □

Note the simplicity of this proof using multi-language semantics. By providing a framework in which source and target programs can be freely mixed, we enable direct equational reasoning between programs and their compilation. Since multi-language equivalence is trivially sound with respect to source or target equivalence, a chain of equations using intermediate multi-language terms can be used to prove purely source or purely target programs are equivalent.

As an obvious corollary, the composition of compiler and back-translation is fully abstract. However we only use that it preserves equivalence, since we can already easily prove that it reflects equivalence.

Lemma 5.8 (There and Back is Fully Abstract)

$$\Gamma \vdash \mathsf{e} \approx^{ctx}_{\mathsf{S}} \mathsf{e}' : \sigma \text{ iff } \Gamma \vdash (\!|[\![\mathsf{e}]\!]|\!) \approx^{ctx}_{\mathsf{S}} (\!|[\![\mathsf{e}']\!]|\!) : \sigma$$

Finally, we have that the back-translation is equivalence reflecting.

Lemma 5.9 (Back Translation is Equivalence Reflecting)
If $\Gamma \vdash (\!|\mathbf{e}|\!) \approx^{ctx}_{\mathsf{S}} (\!|\mathbf{e}'|\!) : \sigma$ *then* $\cdot; \Gamma^{+} \vdash \mathbf{e} \approx^{ctx}_{\mathbf{T}} \mathbf{e}' : \sigma^{\div}$.

Proof Analogous to proof of Theorem 1.2, using Lemma 5.5. □

We conclude with the proof of equivalence preservation as an obvious consequence. This part of the proof is highly general and completely elementary. Its content is that if the composition of two translations ($[\![\cdot]\!]$ and $(\!|\cdot|\!)$) is equivalence preserving and the second ($(\!|\cdot|\!)$) is equivalence reflecting, then the first ($[\![\cdot]\!]$) is equivalence preserving.

Proof Assume $\Gamma \vdash \mathsf{e} \approx^{ctx}_{\mathsf{S}} \mathsf{e}' : \sigma$.
By Lemma 5.8, $\Gamma \vdash (\!|[\![\mathsf{e}]\!]|\!) \approx^{ctx}_{\mathsf{S}} (\!|[\![\mathsf{e}']\!]|\!) : \sigma$.
By Lemma 5.9, $\cdot; \Gamma^{+} \vdash [\![\mathsf{e}]\!] \approx^{ctx}_{\mathbf{T}} [\![\mathsf{e}']\!] : \sigma^{\div}$. □

6. Related Work

Interest in fully abstract translation has a long history [24, 32, 16, 38, 35, 33, 5, 6, 2, 17, 3, 15, 27, 28], with several authors pointing out that failures of full abstraction can be turned into security exploits [1, 20, 13].

Until recently, most results on fully abstract translations have strategically used source and target languages that are in close correspondence, either (1) picking syntactically identical source and target languages [5, 15] or (2) adding problematic target-language features to the source language [32, 33, 15]. This can be attributed to the lack of back-translation techniques for settings where the target is more expressive than the source. Our *Universal Embedding* technique offers a solution to this problem, discussed further in §7.

Protection via types Ahmed and Blume [5] (AB08) were first to prove that typed closure conversion of System F with recursive types is fully abstract. Since their source and target languages coincide, they define type-directed wrappers in the same language to coerce terms from type τ to τ^{+}, and vice versa, instead of a back-translation. Our work, with different source and target languages (even without exceptions), provides one significant advantage over theirs: our target language's type system guarantees that function bodies are closed, as they should be after closure conversion.

Later, Ahmed and Blume [6] (AB11) developed a typed CPS translation from simply typed λ-calculus to System F and proved it fully abstract using a multi-language semantics and *Back-Translation by Partial Evaluation*. To back-translate a term of translation type, the technique uses partial evaluation to eliminate all subterms of non-translation type.

Bowman and Ahmed [10] define a noninterference-preserving translation from DCC to F_{ω}, both terminating languages. Like contextual equivalence, noninterference is a relational property. Their proof, like AB11, uses back-translation by partial evaluation, but also requires a complex well-foundedness argument as their back-translation is not inductively defined.

Glew [16] presents closure conversion for an object calculus and proves it fully abstract. He notes, however, that object closure conversion is simpler than functional closure conversion. In particular, the latter can be encoded as the composition of (1) encoding functions as objects, (2) object closure conversion, and (3) an object encoding. Glew only shows that step (2) is fully abstract. Hence, for full abstraction of functional closure conversion, one would also need to prove encodings (1) and (3) fully abstract.

Protection via dynamic checks Fournet et al. [15] prove full abstraction of a translation from monomorphic ML-like language to js^{*}. While their actual compiler implementation produces JavaScript, their proof of full abstraction is for the translation to js^{*}, which is not actual JavaScript, but rather an encoding of JavaScript in their source language. Their proof technique, like AB08's, relies on the source and target languages being identical, to which end they added exceptions and fatal errors to their source. Our work is the first full abstraction result where the target language contains exceptions but the source does not. Like AB08, Fournet et al. use type-based invariants to prove full abstraction. However, unlike AB08, they use type-directed wrappers to *dynamically* protect translated components from target contexts.

Agten, Patrignani, *et al.* [3, 27, 28] give a fully abstract translation from an object-based language—Java Jr. [18] extended with exceptions—to a *protected module architecture*—an untyped assembly language extended with a hardware-supported isolation mechanism. Their translation relies on hardware-provided fine-grained memory access protection to ensure full abstraction. They use fully abstract trace semantics for Java Jr. [18] so they can prove their theorems in terms of trace equivalence.

Retracts The concept of types as retracts of a universal domain goes back to Scott [34], but the concept of type-indexed embedding-projection pairs for mediating between a typed source language and an embedded untyped language draws directly from Benton [7] and Ramsey [31], where they use this as an *implementation technique* for embedding a scripting language communicating with a host language. Here we use their implementation technique as a formal tool to prove secure compilation.

Approximate back-translation In recent work, Devriese et al. [11] showed how to prove full abstraction of a translation from simply typed λ-calculus with recursive functions (but not recursive types) to the untyped λ-calculus. Since their target is untyped, they back-translate to a universal type in the source language. However, the difficulty in that work is in the fact that the appropriate universal type cannot be constructed in the source (due to a lack of recursive types). Thus they show that the full universal type is not actually necessary, as long as arbitrarily large approximations of it exist. We discuss a potential use of this in §7.

In comparison, our work—done independently of theirs—tackles a nontrivial program transformation, and demonstrates universal embedding for a *more richly typed* target language with exceptions. Also, since the universal type can be constructed in our source language, we are able to separate the core of the technique—universal embedding—from the use of approximate back-translation, making our presentation more elementary. We view their independent discovery and use of (approximate) back-translation to universal type in a fairly different setting to be evidence for the generality of the technique.

Compositional Compiler Correctness While the central focus of this paper has been on using *full abstraction* to ensure security of compiled components with respect to source-language reasoning we have also proved *compositional compiler correctness* (Theorem 1.1), a topic that has been the focus of several recent papers. Perconti and Ahmed [29] showed how to compile components correctly while allowing interoperability with target components that have no equivalent in the source. This allows linking with components that are inexpressible in the source, but requires extra-language reasoning to achieve security. Like us, Perconti and Ahmed [29] use multi-language semantics to specify interoperability between source and target components, and prove compositional compiler correctness of a multi-pass compiler from System F to a low-level language with mutable references. They allow linking with any well-typed target component, but unlike our work, their type translation does not restrict the target language to make only the same observations as the source language. Stewart et al. [37] use *interaction semantics*, which provide an abstract specification of interoperability between source and target components, to prove compositional correctness of the CompCert C compiler. This work allows linking with any target component that respects restrictions imposed by CompCert's memory model via a specified interactions semantics. Neis et al. [26] prove compositional correctness of a compiler for an ML-like language to an assembly-like language, and allow linking with only those components that are related to some source component via a *parametric inter-language simulation* that specifies equivalence between source-language and target-language code. Recent work by Kang et al. [19] demonstrates that if one makes the pragmatic choice to restrict linking to only those components produced by the same verified compiler—i.e., supporting separate compilation—that significantly eases the proof effort. In particular, they prove correctness of SepCompCert which permits linking only with other components produced by their modified version of CompCert.

7. Discussion

With much recent work, we can begin to classify different techniques for proving full abstraction. Some techniques are subsumed by others, but offer a trade-off of power vs. simplicity. Here we focus on comparing back-translation-based techniques, though there are other techniques for proving full abstraction, such as those that make use fully abstract trace semantics [3, 27, 28] as mentioned in §6.

The first technique may be called *Back Translation by Partial Evaluation* as used in Ahmed and Blume [6], Bowman and Ahmed [10], Shikuma and Igarashi [36]. Back-translation by partial evaluation relies on the fact that in some languages, if a term has translation type, then any uses of non-translation type in a subterm are *inessential* and can therefore be eliminated by partial evaluation. However, this is not a realistic assumption for non-terminating languages or languages with information hiding. For instance, a diverging program may do arbitrary computation using non-translation type that cannot be eliminated by partial evaluation. Furthermore, this technique alone will not work when both languages have state or existential types since programs of target type can use values of non-translation type in their closure or as the existential witness. For this reason, back translation by partial evaluation has only been applied to purely functional, normalizing languages. However, it has the benefit of being fairly systematic when it is applicable.

The second technique may be called *Back Translation by Deep Embedding* as used in Ahmed and Blume [5], Fournet et al. [15], Devriese et al. [11], and this paper. This technique translates all target code, not just those at translation type. The translation type code is not translated directly to the source type, but instead wrapped with a semantic boundary.

In Ahmed and Blume [5] and Fournet et al. [15], the target and source are the same so the back-translation is the identity and the boundaries are called "wrappers". These wrappers are witness to a type *isomorphism* and consequently the translation is fully abstract. We say that this back-translation is *precise* because the embedding of the target language is into isomorphic types.

In this paper, we back-translate to a universal type and the boundaries are *retractions*, that is $\mathrm{PROJECT}(\sigma)\ \mathrm{EMBED}(\sigma)\ \mathtt{e} \approx_{\mathsf{S}}^{ctx} \mathtt{e}$. We say our back-translation is *over-approximating* in that it embeds the target language at types which include many more behaviors (full untyped lambda calculus), but due to the boundaries the code is only run on "good" values, i.e., those that represent source values.

Finally, in Devriese et al. [11], they back-translate an untyped language to a simply typed language *without* recursive types, so they cannot construct the universal type as in this work. However, they can construct arbitrarily large approximations to the universal type, and a family of increasingly precise approximations to it. They show that due to the finitary nature of observation, i.e. that observing termination only takes a finite number of steps, they can find a *large enough* approximation to show that equivalence is preserved for any particular program and context. We say their back-translation is *under-approximating* since it embeds the target language at types which include only a subset of the behaviors of the target.

It seems likely to be fruitful to combine the last two deep-embedding approaches, to construct a sequence of *under-approximations* to an *over-approximating* type (like a universal type).

Candidates for such a technique would include embedding languages into intermediate languages with richer type systems where a precise universal type for the source cannot be constructed. For instance, proving that embedding a non-terminating System F-like language into an Fω-like or dependently-typed language would require simulating the more flexible typed terms of the target language.

What should a universal type for an F-like language be? In addition to being able to encode all of the base types it should also encode all terms of abstract types that are in scope, so we need not one U, but an indexed family:

$$\mathsf{U}_\Delta \cong (\mathsf{U}_\Delta \to \mathsf{U}_\Delta) + (\forall\alpha.\,\mathsf{U}_{\Delta,\alpha}) + \Sigma(\Delta)$$

where $\Sigma(\alpha_1,\ldots,\alpha_n) = \alpha_1 + \cdots + \alpha_n$. Since the universal type includes polymorphic types, it depends on the definition of the universal type with one type parameter. Clearly recursive types alone are not enough to construct such a type so it seems that a sequence of under-approximations is necessary.

This source-target combination may seem strange but this scenario would arise if one were to construct a reusable compiler backend for fully abstract compilation. Since the backend would have to have typing features of the most richly typed source language it supports, then using it as a fully abstract backend would require proving that embedding less richly typed languages into more richly typed languages is secure.

We now have a spectrum of Deep Embedding techniques, and as a practical matter one should use the simplest one necessary for the job. For instance, a precise encoding is fairly simple to prove fully abstract using wrappers. However, if the target has programs

untypeable *precisely* in the source then an over-approximation seems necessary. Finally, if the source is too weak to express the intended back-translation type, then a sequence of approximations may be used.

Effects Our technique should be effective in back-translating other well-bracketed/delimited effects. For instance, if compiling a purely functional language to a stateful language that ensures that the compiled terms only link with pure code, target terms can be back-translated to store-passing style, changing the result type to something like $\mathsf{R} = \mathsf{S} \to (\mathsf{S} \times \mathsf{U})$. Similarly, if the target contains delimited continuations (protecting source code with delimiters), it can be back-translated to continuation-passing style, with $\mathsf{R} = (\mathsf{U} \to \mathsf{U}) \to \mathsf{U}$.

Information Hiding The technique should easily handle information hiding, such as state and existential types. As an example if the source and target have existential types, then a term of type $\exists\alpha\,.\,\sigma^+$ must be back-translated to a source term of existential type. An existential package may instantiate α with a target type not in the image of the back-translation, meaning we must be able to simulate the behavior of a target term of non-translation type using a source term. With our technique this is simple, because the witness type can be back-translated to dynamic type. On the other hand, back-translation by partial evaluation, as explained above, could not be used.

References

[1] M. Abadi. Protection in programming-language translations. In *International Colloquium on Automata, Languages and Programming (ICALP)*, pages 868–883, 1998.

[2] M. Abadi and G. Plotkin. On protection by layout randomization. *ACM Transactions on Information and Systems Security*, 15(2), July 2012.

[3] P. Agten, R. Strackx, B. Jacobs, and F. Piessens. Secure compilation to modern processors. In *Computer Security Foundations Symposium (CSF)*, pages 171–185, 2012.

[4] A. Ahmed. Step-indexed syntactic logical relations for recursive and quantified types. In *European Symposium on Programming (ESOP)*, pages 69–83, Mar. 2006.

[5] A. Ahmed and M. Blume. Typed closure conversion preserves observational equivalence. In *International Conference on Functional Programming (ICFP), Victoria, British Columbia, Canada*, pages 157–168, Sept. 2008.

[6] A. Ahmed and M. Blume. An equivalence-preserving CPS translation via multi-language semantics. In *International Conference on Functional Programming (ICFP), Tokyo, Japan*, pages 431–444, Sept. 2011.

[7] N. Benton. Embedded interpreters. *Journal of Functional Programming*, 15(04):503–542, 2005.

[8] N. Benton and A. Kennedy. Exceptional syntax. *Journal of Functional Programming*, 11(04):395–410, 2001.

[9] N. Benton, A. Kennedy, and G. Russell. Compiling Standard ML to Java bytecodes. In *International Conference on Functional Programming (ICFP), Baltimore, Maryland, USA*, pages 129–140, 1998. URL `http://doi.acm.org/10.1145/289423.289435`.

[10] W. J. Bowman and A. Ahmed. Noninterference for free. In *International Conference on Functional Programming (ICFP), Vancouver, British Columbia, Canada*, 2015. URL `http://dx.doi.org/10.1145/2784731.2784733`.

[11] D. Devriese, M. Patrignani, and F. Piessens. Fully-abstract compilation by approximate back-translation. In *ACM Symposium on Principles of Programming Languages (POPL), St. Petersburg, Florida*, page To appear, 2016.

[12] D. Dreyer, G. Neis, and L. Birkedal. The impact of higher-order state and control effects on local relational reasoning. *Journal of Functional Programming*, 22(4&5):477–528, 2012.

[13] V. D'Silva, M. Payer, and D. Song. The correctness-security gap in compiler optmization. In *Language-theoretic Security IEEE Security and Privacy Workshop (LangSec)*, 2015.

[14] M. Felleisen and R. Hieb. A revised report on the syntactic theories of sequential control and state. *Theor. Comput. Sci.*, 103(2):235–271, 1992.

[15] C. Fournet, N. Swamy, J. Chen, P.-E. Dagand, P.-Y. Strub, and B. Livshits. Fully abstract compilation to JavaScript. In *ACM Symposium on Principles of Programming Languages (POPL), Rome, Italy*, pages 371–384, 2013.

[16] N. Glew. Object closure conversion. In *Higher-Order Operational Techniques in Semantics (HOOTS '99)*, Sept. 1999.

[17] R. Jagadeesan, C. Pitcher, J. Rathke, and J. Riely. Local memory via layout randomization. In *Computer Security Foundations Symposium (CSF)*, pages 161–174, 2011.

[18] A. Jeffrey. A fully abstract semantics for a concurrent functional language with monadic types. In *IEEE Symposium on Logic in Computer Science (LICS), San Diego, California*, 1995.

[19] J. Kang, Y. Kim, C.-K. Hur, D. Dreyer, and V. Vafeiadis. Lightweight verification of separate compilation. In *ACM Symposium on Principles of Programming Languages (POPL), St. Petersburg, Florida*, pages 178–190. ACM, 2016.

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

[21] J.-L. Krivine. Classical logic, storage operators and second-order lambda-calculus. *Annals of Pure and Applied Logic*, 68(1):53–78, 1994.

[22] J. R. Longley. Universal types and what they are good for. In *Domain theory, logic and computation*, pages 25–63. Springer, 2003.

[23] J. Matthews and R. B. Findler. Operational semantics for multi-language programs. In *ACM Symposium on Principles of Programming Languages (POPL), Nice, France*, pages 3–10, Jan. 2007.

[24] A. Meyer and J. G. Riecke. Continuations may be unreasonable. In *Conf. on LISP and functional programming, LFP '88*, pages 63–71, 1988.

[25] Y. Minamide, G. Morrisett, and R. Harper. Typed closure conversion. In *ACM Symposium on Principles of Programming Languages (POPL), St. Petersburg Beach, Florida*, pages 271–283, Jan. 1996.

[26] G. Neis, C.-K. Hur, J.-O. Kaiser, C. McLaughlin, D. Dreyer, and V. Vafeiadis. Pilsner: A compositionally verified compiler for a higher-order imperative language. In *International Conference on Functional Programming (ICFP), Vancouver, British Columbia, Canada*, Aug. 2015.

[27] M. Patrignani, D. Clarke, and F. Piessens. Secure compilation of object-oriented components to protected module architectures. In *Proceedings of the 11th Asian Symposium on Programming Languages and Systems (APLAS), Melbourne, Australia*, Dec. 2013.

[28] M. Patrignani, P. Agten, R. Strackx, B. Jacobs, D. Clarke, and F. Piessens. Secure compilation to protected module architectures. *ACM Transactions on Programming Languages and Systems*, 37(2): 6:1–6:50, Apr. 2015.

[29] J. T. Perconti and A. Ahmed. Verifying an open compiler using multi-language semantics. In *European Symposium on Programming (ESOP)*, Apr. 2014.

[30] A. M. Pitts and I. D. Stark. Operational reasoning for functions with local state. *Higher order operational techniques in semantics*, pages 227–273, 1998.

[31] N. Ramsey. Embedding an interpreted language using higher-order functions and types. *Journal of Functional Programming*, 21(06): 585–615, 2011.

[32] J. G. Riecke. Fully abstract translations between functional languages. In *ACM Symposium on Principles of Programming Languages (POPL), Orlando, Florida*, pages 245–254, 1991.

[33] S. B. Sanjabi and C.-H. L. Ong. Fully abstract semantics of additive aspects by translation. In *Proceedings of the 6th international confer-*

ence on Aspect-oriented software development (AOSD), pages 135–148, 2007.

[34] D. Scott. Data types as lattices. *Siam Journal on computing*, 5(3): 522–587, 1976.

[35] N. Shikuma and A. Igarashi. Proving noninterference by a fully complete translation to the simply typed λ-calculus. In *Proceedings of the 11th Asian computing science conference on Advances in computer science: secure software and related issues*, pages 301–315. Springer-Verlag, 2007.

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

[37] G. Stewart, L. Beringer, S. Cuellar, and A. W. Appel. Compositional compcert. In *ACM Symposium on Principles of Programming Languages (POPL), Mumbai, India*, 2015.

[38] S. Tse and S. Zdancewic. Translating dependency into parametricity. In *International Conference on Functional Programming (ICFP), Snowbird, Utah*, pages 115–125. ACM, 2004.

[39] P. Wang, S. Cuellar, and A. Chlipala. Compiler verification meets cross-language linking via data abstraction. In *ACM Symposium on Object Oriented Programming: Systems, Languages, and Applications (OOPSLA)*, Oct. 2014.