Foreign Function Typing: Semantic Type Soundness for FFIs

DANIEL PATTERSON, Northeastern University, USA
AMAL AHMED, Northeastern University, USA

In practice, FFIs are implemented by translation to a target language and by conversions at the level of the target. We wish to establish type soundness in such a setting, where there are two languages making foreign calls to one another. In particular, we want a notion of *convertibility*, that a type τ_A from language A is convertible to a type τ_B from language B, which we will write $\tau_A \sim \tau_B$, such that conversions between these types maintain type soundness (dynamically or statically) of the overall system.

In this paper, we leverage the idea that sound FFIs are a generalization of sound gradual typing, except unlike gradual typing, FFIs mediate between languages *without* a common underlying term language. We start with two languages and define type soundness in a manner inspired by the practical implementation strategy: that the languages will be translated to a common target. We do this using a realizability model, that is, by setting up a logical relation indexed by source types but inhabited by target terms that behave as dictated by the source types. The conversions $\tau_A \sim \tau_B$ that *should* be allowed, are the ones implemented by target-level translations that convert terms that semantically behave like τ_A to terms that semantically behave like τ_B (and vice versa). This means that the converted term must either produce a runtime cast failure or produce some term or value that behaves as τ_B. With the notion of $\tau_A \sim \tau_B$ in hand, we can type check a foreign function call, and based on the semantics of $\tau_A \sim \tau_B$, we can prove that typing the foreign function call is sound.

ACM Reference Format:
Daniel Patterson and Amal Ahmed. 2020. Foreign Function Typing: Semantic Type Soundness for FFIs. In *Informal Proceedings of the first ACM SIGPLAN Workshop on Gradual Typing (WGT20).* ACM, New York, NY, USA, 12 pages.

1 INTRO

We study practical and principled ways for safe interoperation between languages. While in principle systems may involve many languages, a single interaction will be between two languages, so we focus on interoperability between two languages. Our practical inspiration comes from Foreign Function Interfaces (FFIs), which are the primary mechanism for reasonably high-performance cross-language function calls. Our principled inspiration comes from research on gradual typing—in particular, New *et al.*'s approach based on semantic foundations for gradual typing [New and Ahmed 2018; New et al. 2019].

Why FFIs. The reason why FFIs are considered challenging is that the abstractions that one language uses are not necessarily the abstractions of the other language. As a result, the reasoning that one can normally do (i.e., type-based reasoning – that a string can't be used as a number, or reasoning based on polymorphism, etc.) may not hold once values are passed across the FFI. For example, OCaml is a strongly typed, essentially type-sound language, but every value is represented uniformly with a single `value` representation across the FFI boundary with C. It's up to the programmer to be sure that strings are used as strings, numbers are used as numbers, etc. Even more insidiously, a value of type `'a` in OCaml crosses as the same `value` representation, and thus can be inspected to figure out what actual type it represents, even though in OCaml it isn't possible to tell (because of parametricity).

WGT20, January 25, 2020, New Orleans
2020. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM.

Our approach. In practice, FFIs are implemented by translation to a target and by conversions that happen at the level of the target. We want to figure out how to show type soundness in such a setting, where there are two languages making foreign calls to one another. In particular, we want a notion of *convertibility*, that a type τ_A from language A is convertible to a type τ_B from language B, which we will write $\tau_A \sim \tau_B$. But the question is: what types τ_A and τ_B can we allow conversions between while still maintaining type soundness (dynamically or statically) of the overall system?

In this paper, we start with two languages and define type soundness in a manner inspired by the practical implementation strategy: that the languages will be translated to a common target. We do this using a realizability model, that is, by setting up a logical relation that is indexed by source types but inhabited by the sets of target terms that behave as dictated by the source types (e.g., à la Benton and Hur [2009]; Benton and Tabareau [2009]). To understand what conversions $\tau_A \sim \tau_B$ *should* be allowed, we need to show that the target-level conversions that the implementation provides have the property that they translate terms that behave like τ_A to terms that behave like τ_B (and vice versa). More formally, if $e_A \in [\![\tau_A]\!]$ then converting e_A to language B using the conversion $C_{\tau_A \mapsto \tau_B}$, we have that $C_{\tau_A \mapsto \tau_B}(e_A) \in [\![\tau_B]\!]$. According to our logical relation, this means that the converted term must either produce a runtime cast failure or produce some term or value that behaves as τ_B. Once we have the notion of $\tau_A \sim \tau_B$, we can type check a foreign function call and based on the soundness of $\tau_A \sim \tau_B$, can prove that the typing of that foreign function call is itself sound.

Connection to Gradual Typing. In many ways, the problem of two languages interacting over an FFI is a generalization of the problem of gradual typing. The key difference is that gradual typing is concerned with interaction between two languages that, after some type erasure, share the same underlying syntax and operational semantics, whereas in the FFI setting there is no common underlying term language. By recognizing that FFIs are actually implemented by first translating to a common (target) language and defining the source-type-indexed logical relations by which we reason over those common terms, we can recover a common term language and reason about the soundness of conversions between terms that behave as types from different languages, which is similar to reasoning about casts/coercions between types in gradual typing. Indeed, the convertibility relation $\tau_A \sim \tau_B$ is analogous to consistency relations in gradual typing, and there are some similarities in the properties we want it to satisfy. But there are also differences: since there is no particular relationship between the two type systems, there is no (pre-ordained) notion of type precision and indeed there may be many different equally valid alternative options for $\sim$. In future work, we plan to investigate more closely how we can derive properties akin to the gradual guarantee from a particular choice of precision, or what may prevent that from happening. Finally, we note that if one of our two source languages was dynamically typed while the other is statically typed, most of our treatment should look very similar to gradual typing.

A Case Study. In this paper, we illustrate our methodology for designing safe FFIs with a case study involving two languages: a simply-typed linear language and a language with a simple indexed type system, both compiled to a simply-typed target that has fewer type features than either source. The target, indeed, could be dynamically typed—as will often be the case with practical FFI implementations—but for simplicity, we retain some types in the target and our compilers translate source terms by simply erasing extra type features in the source languages. We then design and prove safe a foreign function interface between the two languages.

2 A PAIR OF LANGUAGES

To begin our study, we consider two source languages, which each extend the simply-typed lambda calculus with interesting type features.

$$
\begin{array}{lrl}
\textsc{BabyDill} & \tau & ::= \quad unit \mid int \mid \tau \multimap \tau \mid !\tau \mid \tau \& \tau \mid \tau \otimes \tau \\
& e & ::= \quad () \mid x \mid a \mid \lambda a : \tau . e \mid e\ e \mid e\ e \mid !e \mid let\ !x = e\ in\ e' \mid \\
& & \quad \langle e, e' \rangle \mid e.1 \mid e.2 \mid (e, e) \mid let\ (a, a') = e\ in\ e' \\
& v & ::= \quad () \mid \lambda a : \tau . e \mid !e \mid \langle e, e' \rangle \mid (v, v')
\end{array}
$$

$$
\frac{}{\Gamma; a : \tau \vdash a : \tau} \qquad \frac{x : \tau \in \Gamma}{\Gamma; \cdot \vdash x : \tau} \qquad \frac{}{\Gamma; \cdot \vdash n : int}
$$

$$
\frac{\Gamma; \Delta, a : \tau_1 \vdash e : \tau_2}{\Gamma; \Delta \vdash \lambda a : \tau_1 . e : \tau_1 \multimap \tau_2} \qquad \frac{\Gamma; \Delta_1 \vdash e_1 : \tau_1 \multimap \tau_2 \qquad \Gamma; \Delta_2 \vdash e_2 : \tau_1 \qquad \Delta_3 \cong \Delta_1, \Delta_2}{\Gamma; \Delta_3 \vdash e_1\ e_2 : \tau_2}
$$

$$
\frac{\Gamma; \cdot \vdash e : \tau}{\Gamma; \cdot \vdash e : !\tau} \qquad \frac{\Gamma; \Delta_1 \vdash e : !\tau \qquad \Gamma, x : \tau; \Delta_2 \vdash e' : \tau' \qquad \Delta_3 \cong \Delta_1, \Delta_2}{\Gamma; \Delta_3 \vdash let\ !x = e\ in\ e'}
$$

$$
\frac{\Gamma; \Delta \vdash e_1 : \tau_1 \qquad \Gamma; \Delta \vdash e_2 : \tau_2}{\Gamma; \Delta \vdash \langle e_1, e_2 \rangle : \tau_1 \& \tau_2} \qquad \frac{\Gamma; \Delta \vdash e : \tau_1 \& \tau_2}{\Gamma; \Delta \vdash e.1 : \tau_1} \qquad \frac{\Gamma; \Delta \vdash e : \tau_1 \& \tau_2}{\Gamma; \Delta \vdash e.2 : \tau_2}
$$

$$
\frac{\Gamma; \Delta_1 \vdash e_1 : \tau_1 \qquad \Gamma; \Delta_1 \vdash e_2 : \tau_2 \qquad \Delta_3 \cong \Delta_1, \Delta_2}{\Gamma; \Delta_3 \vdash (e_1, e_2) : \tau_1 \otimes \tau_2}
$$

$$
\frac{\Gamma; \Delta_1 \vdash e : \tau_1 \otimes \tau_2 \qquad \Gamma; \Delta_2, a : \tau_1, a' : \tau_1 \vdash e' : \tau' \qquad \Delta_3 \cong \Delta_1, \Delta_2}{\Gamma; \Delta_3 \vdash let\ (a, a') = e\ in\ e' : \tau'}
$$

Fig. 1. Syntax and (very) selected static semantics for BabyDill.

Our first source language is BabyDill, a simply typed linear lambda calculus based on removing polymorphism from PDILL[Zhao et al. 2010] and adding base types. Briefly, BabyDill has both linear variables, written a and tracked in the linear environment Δ, and unrestricted variables, written x and tracked in the unrestricted environment Γ. In addition to the usual linear arrows (given type $\multimap$), we have both additive ($\tau_1 \& \tau_2$) and multiplicative ($\tau_1 \otimes \tau_2$) linear pairs. The syntax and a few typing rules are shown in Figure 1. The typing judgment has the form $\Gamma; \Delta \vdash e : \tau$. (The reader may consult [Zhao et al. 2010] for more details on the language BabyDill derives from.)

Our second source language is Index$^{\pm}$, the simply-typed lambda calculus with integers enriched with an positive/negative index algebra that can be used to track the sign of numbers. The syntax and a selection of the typing rules are in Figure 2. Number types η capture our index system, where literal numbers are either $+0$ (non-negative) or $-$ (negative), and computations that return numbers can degrade to the unknown-sign type num. Number type polymorphism allows functions like $e = \Lambda\alpha.\lambda x{:}\alpha.x * 2$, which has type $\forall\alpha.\alpha \to \alpha * +0$. Then $e[+0]2$ would have type $+0$, since type application reduces $+0 * +0$ to $+0$. The typing judgment has the form $H; \Gamma \vdash e : \tau$ where the environment H keeps track of number-type variables α. The operational semantics are standard; evaluation does not depend on indexes.

Note that each language has syntax for a foreign call to the other language. We defer the static semantics for these foreign calls until later, as it is indeed the entire subject of this paper. The foreign call mimics how FFIs typically work: a foreign function is applied to native arguments, with the result being used natively. Without the static rules for foreign calls, programs involving them would simply be ill-typed.

$$
\begin{array}{llll}
\textsc{Index}^{\pm} & \tau & ::= & \eta \mid \forall\alpha.\tau \mid \tau \to \tau \mid \tau \times \tau \\
& \eta & ::= & \alpha \mid +0 \mid - \mid \mathit{num} \mid \eta{+}\eta \mid \eta{*}\eta \\
& e & ::= & n \mid x \mid e + e \mid e * e \mid \lambda x : \tau.\, e \mid e\, e \mid e\, e \\
& & & \Lambda\alpha.e \mid e[\eta] \mid (e, e) \mid e.1 \mid e.2 \\
& v & ::= & n \mid \lambda x : \tau.\, e \mid \Lambda\alpha.e \mid (v, v)
\end{array}
$$

$$\frac{n \geq 0}{\mathrm{H};\Gamma \vdash n : +0} \qquad \frac{n < 0}{\mathrm{H};\Gamma \vdash n : -} \qquad \frac{x : \tau \in \Gamma}{\mathrm{H};\Gamma \vdash x : \tau} \qquad \frac{\mathrm{H};\Gamma \vdash e_1 : \eta_1 \qquad \mathrm{H};\Gamma \vdash e_2 : \eta_2}{\mathrm{H};\Gamma \vdash e_1 + e_2 : \Downarrow(\eta_1{+}\eta_2)}$$

$$\frac{\mathrm{H};\Gamma \vdash e_1 : \eta_1 \qquad \mathrm{H};\Gamma \vdash e_2 : \eta_2}{\mathrm{H};\Gamma \vdash e_1 * e_2 : \Downarrow(\eta_1{*}\eta_2)} \qquad \frac{\mathrm{H};\Gamma, x\ \tau_1 \vdash e : \tau_2}{\mathrm{H};\Gamma \vdash \lambda x.e : \tau_1 \to \tau_2} \qquad \frac{\mathrm{H};\Gamma \vdash e : \tau_1 \to \tau_2 \qquad \mathrm{H};\Gamma \vdash e' : \tau_1}{\mathrm{H};\Gamma \vdash e\ e' : \tau_2}$$

$$\frac{\mathrm{H}, \alpha;\Gamma \vdash e : \tau}{\mathrm{H};\Gamma \vdash \Lambda\alpha.e : \forall\alpha.\tau} \qquad \frac{\mathrm{H};\Gamma \vdash e : \forall\alpha.\tau}{\mathrm{H};\Gamma \vdash e[\eta] : \Downarrow(\tau[\eta/\alpha])} \qquad \frac{\mathrm{H};\Gamma \vdash e_1 : \tau_1 \qquad \mathrm{H};\Gamma \vdash e_2 : \tau_2}{\mathrm{H};\Gamma \vdash (e_1, e_2) : \tau_1 \times \tau_2}$$

$$\frac{\mathrm{H};\Gamma \vdash e : \tau_1 \times \tau_2}{\mathrm{H};\Gamma \vdash e.1 : \tau_1} \qquad \frac{\mathrm{H};\Gamma \vdash e : \tau_1 \times \tau_2}{\mathrm{H};\Gamma \vdash e.2 : \tau_2}$$

$$
\begin{array}{lcl}
\Downarrow(\eta_1 + \eta_2) & = & +\Downarrow(\Downarrow(n_1), \Downarrow(n_2)) \\
\Downarrow(\eta_1 * \eta_2) & = & *\Downarrow(\Downarrow(n_1), \Downarrow(n_2)) \\
\Downarrow(\eta) & = & \eta
\end{array}
$$

$$
\begin{array}{lcl}
+\Downarrow(+0, +0) & = & +0 \\
+\Downarrow(-, +0) & = & \mathit{num} \\
+\Downarrow(+0, -) & = & \mathit{num} \\
+\Downarrow(-, -) & = & - \\
+\Downarrow(\mathit{num}, \_) & = & \mathit{num} \\
+\Downarrow(\_, \mathit{num}) & = & \mathit{num} \\
+\Downarrow(\eta_1, \eta_2) & = & \eta_1{+}\eta_2
\end{array}
\qquad
\begin{array}{lcl}
*\Downarrow(+0, +0) & = & +0 \\
*\Downarrow(-, +0) & = & - \\
*\Downarrow(+0, -) & = & - \\
*\Downarrow(-, -) & = & +0 \\
*\Downarrow(\mathit{num}, \_) & = & \mathit{num} \\
*\Downarrow(\_, \mathit{numty}) & = & \mathit{num} \\
\Downarrow(\eta_1, \eta_2) & = & \eta_1{}\eta_2
\end{array}
$$

Fig. 2. Syntax and (very) selected static semantics for $\textsc{Index}^{\pm}$.

3 A PAIR OF COMPILERS

Our work models FFIs as they work, and that means modeling interaction in the context of the target language that the interacting source languages compile to. To show how this works, here we compile both BabyDill and $\textsc{Index}^{\pm}$ to a simply-typed lambda calculus that has the possibility of runtime failure, which we will use for cast failures. The target language syntax and selected static semantics are presented below in Figure 3. We emphasize that, like many target languages, this target has less static reasoning principles than either of our source languages, and indeed: our approach is viable even in the presence of untyped, unsound targets.

The compilers for our two languages are presented in Figure 4, following the type translations that guide the compilers, which essentially erase any extra type features not present in STLC.

We present the operational semantics for our target language in Figure 5. While the result of evaluation is completely standard call-by-value and the language contains no particularly exotic features, we do instrument the operational semantics in a certain way so that it allows us to reason in our logical relation about the use of variables. What this means is that we run with a machine configuration $\langle \sigma, e \rangle$ that pairs a variable store σ with a term e. In the variable store, each variable has a count of how many times it has been used. Variables are never removed from the store, and thus the counts do not affect evaluation, but they can be reasoned about logically: a store binding $x \overset{n}{\mapsto} v$ says that x has been accessed n times.

We present the evaluation using evaluation contexts; the evaluation contexts, which are standard, appeared in the grammar above, while the reduction rules are given below. As noted above, we

$$
\begin{array}{llll}
\text{STLC} & \tau & ::= & \text{int} \mid \tau + \tau \mid \tau \times \tau \mid \tau \to \tau \\
& e & ::= & n \mid x \mid e + e \mid e * e \mid \text{inl}\, e \mid \text{inr}\, e \mid (e, e) \mid \lambda x :\ \tau.\, e \mid e\, e \\
& & & \mid \text{match}\, e\, \text{with}\, x\, \{e\}\, y\, \{e\} \mid \text{let}\, (x, y) = e\, \text{in}\, e \mid \text{fail} \\
& & & \mid \text{let}\, x = e\, \text{in}\, e \mid \text{ltn}\, e\, e \\
& E & ::= & [\bullet] \mid E + e \mid v + E \mid E * e \mid v * E \mid \text{inl}\, E \mid \text{inr}\, E \mid (E, e) \mid (v, E) \mid E\, e \mid v\, E \\
& & & \mid \text{match}\, E\, \text{with}\, x\, \{e_1\}\, y\, \{e_2\} \mid \text{let}\, (x, y) = E\, \text{in}\, e \mid \text{let}\, x = E\, \text{in}\, e \\
& & & \mid \text{ltn}\, E\, e \mid \text{ltn}\, v\, E \\
& v & ::= & n \mid \text{inl}\, v \mid \text{inr}\, v \mid \lambda x :\ \tau.\, e
\end{array}
$$

$$
\frac{}{\Gamma \vdash n : \text{int}} \qquad \frac{\Gamma \vdash e_1 : \text{int} \qquad \Gamma \vdash e_2 : \text{int}}{\Gamma \vdash e_1 + e_2 : \text{int}} \qquad \frac{\Gamma \vdash e_1 : \text{int} \qquad \Gamma \vdash e_2 : \text{int}}{\Gamma \vdash e_1 * e_2 : \text{int}} \qquad \frac{\Gamma \vdash e : \tau}{\Gamma \vdash \text{inl}\, e : \tau + \tau'}
$$

$$
\frac{\Gamma \vdash e : \tau}{\Gamma \vdash \text{inr}\, e : \tau' + \tau} \qquad \frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash (e_1, e_2) : \tau_1 \times \tau_2} \qquad \frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau} \qquad \frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x :\ \tau_1.\, e : \tau_1 \to \tau_2}
$$

$$
\frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1\, e_2 : \tau_2} \qquad \frac{\Gamma \vdash e : \tau_1 + \tau_2 \qquad \Gamma, x : \tau_1 \vdash e_1 : \tau \qquad \Gamma, y : \tau_2 \vdash e_2 : \tau}{\Gamma \vdash \text{match}\, e\, \text{with}\, x\, \{e_1\}\, y\, \{e_2\}}
$$

$$
\frac{\Gamma \vdash e : \tau_1 \times \tau_2 \qquad \Gamma, x : \tau_1, y : \tau_2 \vdash e' : \tau}{\Gamma \vdash \text{let}\, (x, y) = e\, \text{in}\, e' : \tau} \qquad \frac{}{\Gamma \vdash \text{fail} : \tau} \qquad \frac{\Gamma \vdash e : \tau \qquad \Gamma, x : \tau \vdash e' : \tau'}{\Gamma \vdash \text{let}\, x = e\, \text{in}\, e' : \tau'}
$$

$$
\frac{\Gamma \vdash e_1 : \text{int} \qquad \Gamma \vdash e_2 : \text{int}}{\Gamma \vdash \text{ltn}\, e_1\, e_2 : \text{int} + \text{int}}
$$

Fig. 3. Syntax and (very) selected static semantics for STLC.

maintain use counts for variables, so the rule for variable lookup increments that count, and all the rules that add new bindings to the store set the initial count to 0.

4 A SOURCE-TYPE INDEXED LOGICAL RELATION

The first critical element of Foreign Function Typing is a logical relation inhabited by target terms but indexed by source types—of both languages. In order to support linearity, our target operational semantics is instrumented to maintain variables in a store which counts the number of times the variables are accessed. Indeed, any realizability model for linear types, never mind whether it's in the FFI setting, would require some way to ensure that computations maintain the invariants about variable use if our soundness theorem was to say anything about those linear invariants. And importantly, while our "instrumented" semantics is necessary for the logical relation, it admits a very straightforward erasure to an operational semantics without such tracking (as no operations depend on the use counts), and thus programs will run equivalently under either semantics. While the target language (intentionally) does not know about linearity and uses the store for *all* variables, the invariant of our logical relation is that the store fragments that inhabit the relation only contain the linear bindings. We present the full logical relation in Figure 6.

5 SOUNDNESS

We started this paper asking the question: how can we maintain soundness in the presence of FFI calls, cross-language interaction, etc. We have now set up a logical relation, but on its own, a logical relation doesn't tell us anything about the programs we write, only about the programs

$$
\begin{array}{rcl}
\textsc{BabyDill}\ (\tau)^+ & = & \text{STLC}\ \tau \\
(unit)^+ & = & int \\
(int)^+ & = & int \\
(\tau_1 \multimap \tau_2)^+ & = & {\tau_1}^+ \to {\tau_2}^+ \\
(!\tau)^+ & = & \tau^+ \\
(\tau_1 \& \tau_2)^+ & = & {\tau_1}^+ \times {\tau_2}^+ \\
(\tau_1 \otimes \tau_2)^+ & = & {\tau_1}^+ \times {\tau_2}^+ \\
\\
\textsc{BabyDill}\ (e)^+ & = & \text{STLC}\ e \\
(x : \tau)^+ & = & x : \tau^+ \\
(a : \tau)^+ & = & a : \tau^+ \\
(n : int)^+ & = & n : \text{int} \\
(\lambda a : \tau_1 . e : \tau_1 \multimap \tau_2)^+ & = & \lambda a : {\tau_1}^+ . {e^+} : {\tau_1}^+ \to {\tau_2}^+ \\
(e\ e' : \tau)^+ & = & {e^+}\ {e'}^+ : \tau^+ \\
(!e : !\tau)^+ & = & {e^+} : \tau^+ \\
(let\ !x = e\ in\ e' : \tau)^+ & = & \text{let}\ x = {e^+}\ \text{in}\ {e'}^+ : \tau^+ \\
\langle e_1, e_2 \rangle : \tau_1 \& \tau_2)^+ & = & ({e_1}^+, {e_2}^+) : {\tau_1}^+ \times {\tau_2}^+ \\
(e.1 : \tau)^+ & = & \text{let}\ (x, \_) = {e^+}\ \text{in}\ x : \tau^+ \\
(e.2 : \tau)^+ & = & \text{let}\ (\_, x) = {e^+}\ \text{in}\ x : \tau^+ \\
((e_1, e_2) : \tau_1 \otimes \tau_2)^+ & = & ({e_1}^+, {e_2}^+) : \tau_1^+ \times \tau_2^+ \\
(let\ (a, a') = e\ \text{in}\ e' : \tau)^+ & = & \text{let}\ (a, a') = {e^+}\ \text{in}\ {e'}^+ : \tau^+ \\
\\
\textsc{Index}^{\pm}\ \tau^+ & = & \text{STLC}\ \tau \\
(\eta)^+ & = & int \\
(\forall \alpha . \tau)^+ & = & int \to \tau^+ \\
(\tau_1 \to \tau_2)^+ & = & {\tau_1}^+ \to {\tau_2}^+ \\
(\tau_1 \times \tau_2)^+ & = & {\tau_1}^+ \times {\tau_2}^+ \\
\\
\textsc{Index}^{\pm}\ (e)^+ & = & \text{STLC}\ e \\
(n : \eta)^+ & = & n : \text{int} \\
(x : \tau)^+ & = & x : \tau^+ \\
(e_1 + e_2 : \eta)^+ & = & e_1^+ + e_2^+ : \text{int} \\
(e_1 * e_2 : \eta)^+ & = & e_1^+ * e_2^+ : \text{int} \\
(\lambda\, x.\, e : \tau_1 \to \tau_2)^+ & = & \lambda\, x.\, e^+ : \tau_1^+ \to \tau_2^+ \\
(e_1\ e_2 : \tau)^+ & = & e_1^+\ e_2^+ : \tau^+ \\
(\Lambda\, \alpha.\, e : \forall \alpha . \tau)^+ & = & \lambda\_ . e^+ : \tau^+ \\
(e_1[\eta] : \tau)^+ & = & e_1^+\ 0 : \tau^+ \\
((e_1, e_2) : \tau_1 \times \tau_2)^+ & = & (e_1^+, e_2^+) : \tau_1^+ \times \tau_2^+ \\
(e.1 : \tau)^+ & = & \text{let}\ (x, \_) = e^+\ \text{in}\ x : \tau^+ \\
(e.2 : \tau)^+ & = & \text{let}\ (\_, x) = e^+\ \text{in}\ x : \tau^+
\end{array}
$$

Fig. 4. Compilers from $\textsc{Index}^{\pm}$ and $\textsc{BabyDill}$ to STLC.

that are in the relation! What we have to prove is a soundness theorem, which says that if a term is syntactically well typed, then it is semantically well typed—that is, terms in the relation either terminate at values of the right (semantic) type, or fail with well-defined target-level errors. We state this for open terms as follows:

$$
\begin{array}{c}
\Gamma; \Delta \vdash e : \tau \Rightarrow \forall \gamma \in \mathcal{G}[\![\Gamma]\!].\ \forall \sigma \in \mathcal{D}[\![\Delta]\!].\ (\sigma, \gamma(e^+)) \in \mathcal{E}[\![\tau]\!]. \\
H; \Gamma \vdash e : \tau \Rightarrow \forall \gamma \in \mathcal{G}[\![\Gamma]\!].\ \forall \rho \in \mathcal{R}[\![H]\!].\ (\{\}, \gamma(e^+)) \in \mathcal{E}[\![\tau]\!]_\rho
\end{array}
$$

$$
\begin{array}{lcl}
\langle \sigma, e\rangle & \mapsto & \langle \sigma', e'\rangle \\
\hline
\langle \sigma_1 \uplus x \overset{n}{\mapsto} v, x\rangle & \mapsto & \langle \sigma_1 \uplus x \overset{n+1}{\mapsto} v, v\rangle \\
\langle \sigma, n_1 + n_2\rangle & \mapsto & \langle \sigma, n_3\rangle \mid n_3 = n_1 + n_2 \\
\langle \sigma, n_1 * n_2\rangle & \mapsto & \langle \sigma, n_3\rangle \mid n_3 = n_1 * n_2 \\
\langle \sigma, (\lambda\, x : \tau.e)\ v\rangle & \mapsto & \langle \sigma \uplus \{x \overset{0}{\mapsto} v\}, e\rangle \\
\langle \sigma, \text{match}\ (\text{inl}\ v)\ \text{with}\ x\ \{e_1\}\ y\ \{e_2\}\rangle & \mapsto & \langle \sigma \uplus \{x \overset{0}{\mapsto} v\}, e_1\rangle \\
\langle \sigma, \text{match}\ (\text{inr}\ v)\ \text{with}\ x\ \{e_1\}\ y\ \{e_2\}\rangle & \mapsto & \langle \sigma \uplus \{y \overset{0}{\mapsto} v\}, e_2\rangle \\
\langle \sigma, \text{let}\ (x, y)\ = (v_1, v_2)\ \text{in}\ e\rangle & \mapsto & \langle \sigma \uplus \{x \overset{0}{\mapsto} v_1, y \overset{0}{\mapsto} v_2\}, e\rangle \\
\langle \sigma, \text{let}\ x\ = v\ \text{in}\ e\rangle & \mapsto & \langle \sigma \uplus \{x \overset{0}{\mapsto} v\}, e\rangle \\
\langle \sigma, \text{ltn}\ n_1\ n_2\rangle \mid n_1 < n_2 & \mapsto & \langle \sigma, \text{inr}\ 0\rangle \\
\langle \sigma, \text{ltn}\ n_1\ n_2\rangle \mid n_1 >= n_2 & \mapsto & \langle \sigma, \text{inl}\ 0\rangle
\end{array}
$$

$$
\frac{\langle \sigma, e\rangle \mapsto \langle \sigma, e'\rangle}{\langle \sigma, E[e]\rangle \mapsto \langle \sigma, E[e']\rangle}
\qquad
\frac{}{\langle \sigma, E[\text{fail}]\rangle \mapsto \langle \sigma, \text{fail}\rangle}
$$

Fig. 5. STLC operational semantics.

$$
\begin{array}{lcl}
\mathcal{V}[\![\mathit{unit}]\!]_\rho & = & \{(\{\}, 0)\} \cup \{(\{x{\mapsto}0\}, x) \mid x \text{ is any variable}\} \\
\mathcal{V}[\![\mathit{int}]\!]_\rho & = & \{(\{\}, n)\} \cup \{(\{x{\mapsto}n\}, x) \mid n \text{ is any integer, } x \text{ is any variable}\} \\
\mathcal{V}[\![\tau_1 \multimap \tau_2]\!]_\rho & = & \{(\sigma_f, \lambda\, x.e) \mid \forall\, (\sigma_a, v) \in \mathcal{V}[\![\tau_1]\!]_\rho, (\sigma_f \uplus \sigma_a, (\lambda\, x.e)\ v) \in \mathcal{E}[\![\tau_2]\!]_\rho\} \\
\mathcal{V}[\![!\tau]\!]_\rho & = & \{(\{\}, v) \mid (\{\}, v) \in \mathcal{V}[\![\tau]\!]_\rho\} \\
\mathcal{V}[\![\tau_1 \& \tau_1]\!]_\rho & = & \{(\sigma, (v_1, v_2)) \mid (\sigma, v_1) \in \mathcal{V}[\![\tau_1]\!]_\rho \wedge (\sigma, v_2) \in \mathcal{V}[\![\tau_2]\!]_\rho\} \\
\mathcal{V}[\![\tau \otimes \tau]\!]_\rho & = & \{(\sigma_1 \uplus \sigma_2, (v_1, v_2)) \mid (\sigma_1, v_1) \in \mathcal{V}[\![\tau_1]\!]_\rho \wedge (\sigma_2, v_2) \in \mathcal{V}[\![\tau_2]\!]_\rho\} \\
\mathcal{V}[\![+0]\!]_\rho & = & \{(\{\}, n) \mid n \geq 0\} \\
\mathcal{V}[\![-]\!]_\rho & = & \{(\{\}, n) \mid n < 0\}\} \\
\mathcal{V}[\![\mathit{num}]\!]_\rho & = & \{(\{\}, n) \mid n \text{ is any integer}\} \\
\mathcal{V}[\![\alpha]\!]_\rho & = & \{(\{\}, v) \mid v \in \rho[\alpha]\} \\
\mathcal{V}[\![\eta_1 + \eta_2]\!]_\rho & = & \{(\{\}, v_1 + v_2) \mid (\{\}, v_1) \in \mathcal{V}[\![\eta_1]\!]_\rho \wedge (\{\}, v_2) \in \mathcal{V}[\![\eta_2]\!]_\rho\}\} \\
\mathcal{V}[\![\eta_1 * \eta_2]\!]_\rho & = & \{(\{\}, v_1 * v_2) \mid (\{\}, v_1) \in \mathcal{V}[\![\eta_1]\!]_\rho \wedge (\{\}, v_2) \in \mathcal{V}[\![\eta_2]\!]_\rho\}\} \\
\mathcal{V}[\![\forall \alpha.\tau]\!]_\rho & = & \{(\{\}, \lambda\, \_.e) \mid \forall\, P \subseteq \mathcal{V}[\![\mathit{num}]\!]_\rho, (\{\}, e) \in \mathcal{E}[\![\tau]\!]_{\rho, \alpha \mapsto P}\} \\
\mathcal{V}[\![\tau_1 \to \tau_2]\!]_\rho & = & \{(\{\}, \lambda\, x.e) \mid \forall\, (\{\}, v) \in \mathcal{V}[\![\tau_1]\!]_\rho, (\{\}, [v/x]e) \in \mathcal{E}[\![\tau_2]\!]_\rho\} \\
\mathcal{V}[\![\tau_1 \times \tau_2]\!]_\rho & = & \{(\{\}, (v_1, v_2)) \mid (\{\}, v_1) \in \mathcal{V}[\![\tau_1]\!]_\rho \wedge (\{\}, v_2) \in \mathcal{V}[\![\tau_2]\!]_\rho\} \\
\mathcal{E}[\![\tau]\!]_\rho & = & \{(\sigma_s, e) \mid \forall \sigma_r.\sigma_s \uplus \sigma_r; e \Downarrow \sigma'; \text{fail} \vee \\
 & & \qquad \sigma_s \uplus \sigma_r; e \Downarrow \sigma'; v \wedge \sigma' = \sigma_f \uplus \sigma_r \wedge \text{wf}(\sigma_f - \text{garbage}(\sigma_f, \sigma_s)) \wedge \\
 & & \qquad\qquad (\sigma_f - \text{garbage}(\sigma_f, \sigma_s) - \text{used}(\sigma_f), v) \in \mathcal{V}[\![\tau]\!]_\rho\}
\end{array}
$$

$\text{wf}(\sigma) = \text{all}(\sigma, \text{ref} <= 1)$
$\text{garbage}(\sigma_1, \sigma_2) = \text{dom}(\sigma_1) - \text{dom}(\sigma_2)$
$\text{used}(\sigma) = \text{filter}(\sigma, \text{ref} = 1)$
$\sigma; e \Downarrow \sigma'; v$ means no fail encountered.

Fig. 6. Source-type indexed logical relation.

Here we define $\mathcal{D}[\![\Delta]\!]$, $\mathcal{G}[\![\Gamma]\!]$, and $\mathcal{G}[\![\Gamma]\!]$ in the obvious way (mapping variables to values in the value relation at the right type. For Δ, the mapping carries 0 as the count of uses, as the assumption of a linear environment is that none of the variables have been used yet. $\mathcal{R}[\![\text{H}]\!]$ maps type variables to number predicates, reflecting that our only polymorphism is on number types.

With all those (standard) preliminaries, we have that well-typed source terms, after compilation, are in our relation, which means that they behave according to our $\mathcal{E}[\![\tau]\!]_\rho$ relation which captures

our notion of soundness: they either reduce to fail, or they run to a value that is in the relation at the right type. Note that this is all defined over terms *after compilation*.

This is, indeed, the right property to have in realistic settings where the only operational semantics, and indeed the only programs we run, are those we get after compilation.

We can prove that type soundness holds in the standard way, by induction over the typing derivation. Here, we show for demonstration, a few compatibility lemmas.

Lemma 5.1 (Compatibility n Non-negative). $\mathrm{H}; \Gamma \vdash n : +0 \Rightarrow \forall \gamma \in \mathcal{G}[\![\Gamma]\!]. \forall \rho \in \mathcal{R}[\![\mathrm{H}]\!]. (\{\}, \gamma(n^+)) \in \mathcal{E}[\![+0]\!]_\rho$

Proof. Given $\mathrm{H}; \Gamma \vdash n : +0$ and $\gamma \in \mathcal{G}[\![\Gamma]\!], \rho \in \mathcal{R}[\![\mathrm{H}]\!]$, we need to show that: $(\{\}, \gamma(n^+)) \in \mathcal{E}[\![\tau]\!]_\rho$. Since the compiler is the identity on numbers, and substitution will do nothing, this reduces to showing $(\{\}, n) \in \mathcal{E}[\![\tau]\!]_\rho$. Since n is already a value, we need only show that $(\{\}, n) \in \mathcal{V}[\![\tau]\!]_\rho$. But this holds based on the assumption from $\mathrm{H}; \Gamma \vdash n : +0$: that $n \geq 0$. □

Lemma 5.2 (Compatibility Pair). $\mathrm{H}; \Gamma \vdash (e_1, e_2) : \tau_1 \times \tau_2 \Rightarrow \forall \gamma \in \mathcal{G}[\![\Gamma]\!]. \forall \rho \in \mathcal{R}[\![\mathrm{H}]\!]. (\{\}, \gamma((e_1, e_2)^+)) \in \mathcal{E}[\![\tau_1 \times \tau_2]\!]_\rho$

Proof. Given $\mathrm{H}; \Gamma \vdash (e_1, e_2) : \tau_1 \times \tau_2$ and $\gamma \in \mathcal{G}[\![\Gamma]\!], \rho \in \mathcal{R}[\![\mathrm{H}]\!]$, we need to show that: $(\{\}, \gamma((e_1, e_2)^+)) \in \mathcal{E}[\![\tau_1 \times \tau_2]\!]_\rho$. The compiler and substitution pushes through the pair, meaning we really need to show: $(\{\}, (\gamma({e_1}^+), \gamma({e_2}^+)) \in \mathcal{E}[\![\tau_1 \times \tau_2]\!]_\rho$. From the induction hypotheses, we know that $(\{\}, \gamma({e_1}^+)) \in \mathcal{E}[\![\tau_1]\!]_\rho$, which means it will evaluate to a value in $\mathcal{V}[\![\tau_1]\!]_\rho$ (or else fail, and we're done). The other side evaluates as well, and we end up with a pair (v_1, v_2), where each element is in the respective value relation at the right type. This, in turn, is how $\mathcal{V}[\![\tau_1 \times \tau_2]\!]_\rho$ is defined, so we are done. □

Lemma 5.3 (Compatibility Linear Variable a). $\Gamma; \Delta \vdash a : \tau \Rightarrow \forall \gamma \in \mathcal{G}[\![\Gamma]\!]. \forall \sigma \in \mathcal{D}[\![\Delta]\!]. (\sigma, \gamma(a^+)) \in \mathcal{E}[\![\tau]\!]$.

Proof. Given $\Gamma; \Delta \vdash a : \tau$ and $\gamma \in \mathcal{G}[\![\Gamma]\!], \sigma \in \mathcal{D}[\![\Delta]\!]$, we need to show that: $(\sigma, \gamma(a^+)) \in \mathcal{E}[\![\tau]\!]$.. We know that a is not in Γ, so the substitution does nothing, and the compilation is the identity. We also know that Δ is a singleton with $a : \tau$, which means that σ is $\{a \overset{0}{\mapsto} v\}$ where $(\{\}, v) \in \mathcal{V}[\![\tau]\!]$.. So we can simplify what we have to show to: $(\{a \overset{0}{\mapsto} v\}, a) \in \mathcal{E}[\![\tau]\!]$. We can see that this will evaluate according to the operational semantics to $\langle \{a \overset{1}{\mapsto} v\}, v \rangle$, which means we will need to show that $(\{\}, v)$ is in $\mathcal{V}[\![\tau]\!]$. (since the store entry would be removed by the "used" case). But this is exactly what we have from our assumption, so we are done. This case demonstrates the way that we use our instrumented operational semantics to reason about linear environments, even though the target semantics doesn't have any features specifically about linearity (the same underlying features could be used for various substructural features, or ignored entirely, as with the semantics for Index$^\pm$). □

Proving the rest of the typing rules via compatibility lemmas will show that well-typed programs in our languages are sound with respect to our relation. As yet, however, well-typed programs cannot involve foreign calls, as we have given no way to type them. The goal of the rest of this paper is incorporating those calls into this setting and maintaining the same soundness result in the presence of those calls.

6 CONVERTIBILITY

We incorporate type checking of foreign calls by means of a convertibility relation that defines when we can safely convert between (terms of) a type in one language and (terms of) a type in the other language, which is defined up to the possibility of (well-defined) operational failure. This is a symmetric relation, written $\tau \sim \tau$. We can define rules inductively for this relation, for example:

$$\frac{}{int \sim +0}$$

However, on its own, such a relation does not tell us how we would actually realize (implement) the conversion. For this, we need a pair of target-language compilers/translations, $C_{\tau \mapsto \tau}$ and $C_{\tau \mapsto \tau}$, which may insert runtime coercions, transform code, etc. Hence, the full declarative judgment looks like:

$$\frac{}{(C_{int \mapsto +0}, C_{+0 \mapsto int}) : int \sim +0} \qquad \frac{(C_{\tau_1 \mapsto \tau_1}, C_{\tau_1 \mapsto \tau_1}) : \tau_1 \sim \tau_1 \qquad (C_{\tau_2 \mapsto \tau_2}, C_{\tau_2 \mapsto \tau_2}) : \tau_2 \sim \tau_2}{(C_{\tau_1 \otimes \tau_2 \mapsto \tau_1 \times \tau_2}, C_{\tau_1 \times \tau_2 \mapsto \tau_1 \otimes \tau_2}) : \tau_1 \otimes \tau_2 \sim \tau_1 \times \tau_2}$$

Here the intention is that for rules with premises, the compilers below the line can use the compilers from their premises in their implementations.

Having defined the judgment this way, we can define a typing rule for a foreign function call: we merely require that the argument and return types be convertible, and that the function itself be closed. The latter highlights that this captures semantics of foreign function calls, rather than multi-language semantics where bindings from both languages live in shared environments.

$$\frac{\cdot; \cdot \vdash e_1 : \tau_1 \to \tau_2 \qquad \Delta; \Gamma \vdash e_2 : \tau_1 \qquad \_ : \tau_1 \sim \tau_1 \qquad \_ : \tau_2 \sim \tau_2}{\Delta; \Gamma \vdash e_1\ e_2 : \tau_2}$$

This admits a straightforward translation into the target, by using the corresponding compilers at the right places: $e\ e' \rightsquigarrow C_{\tau_2 \mapsto \tau_2}(e^+\ C_{\tau_1 \mapsto \tau_1}(e'^+))$.

However, up until this point, we haven't made any requirements about what rules would be admissible, or what the compilers could do, or indeed, whether the result is sound. In order to prove soundness, we require that one lemma be proved, by induction on the convertibility derivation:

Lemma 6.1 (Convertibility Soundness).

If

$$(C_{\tau \mapsto \tau}, C_{\tau \mapsto \tau}) : \tau \sim \tau$$

then

$$\forall (\sigma, e) \in \mathcal{E}[\![\tau]\!]., (\sigma, C_{\tau \mapsto \tau}(e)) \in \mathcal{E}[\![\tau]\!].$$

and

$$\forall (\sigma, e) \in \mathcal{E}[\![\tau]\!]., (\sigma, C_{\tau \mapsto \tau}(e)) \in \mathcal{E}[\![\tau]\!].$$

We read this as requiring that the two compilers take terms that are semantically well-typed in one language to terms that are semantically well-typed in the other. The logical relation allows the possibility of runtime casts that can fail via the explicit fail instruction.

The requirement that we prove this lemma ensures that we can prove soundness in the presence of foreign calls, and indeed, may prevent us from defining convertibility between some types for which it is *impossible* to convert them soundly. This happens, for example, with linear types; linearity is a tricky thing and this is the reason why we chose it for our case study.

7 ADDING A FOREIGN FUNCTION INTERFACE IMPLEMENTATION

In order to make foreign calls, we have to, as described in the previous section, add $\tau \sim \tau$ rules, provide compilers to do the conversions on the boundaries, and prove the soundness lemma. For example, we might want to invoke a Index$^{\pm}$ function that expects a positive number with a BabyDill number, which doesn't have a sign. We therefore want to add the following judgment:

$$\frac{}{(C_{int \mapsto +0}, C_{+0 \mapsto int}) : int \sim +0}$$

The first conversion, $C_{int \mapsto +0}$, is relatively straightforward; it just wraps the expression with a runtime sign check that fails the computation if the sign check doesn't pass:

$$\text{match}\,(\text{ltn}\,e\,0)\,\text{with inl}\,\_\{\text{fail}\}\,\text{inr}\,x\{x\}$$

To prove that this is correct, we need to show that it is in $\mathcal{E}[\![+0]\!].$. Given the possible values from $\mathcal{V}[\![int]\!].$, (which we know we can reduce to), we either are running with a literal integer and an empty store, or a variable with a singleton store mapping to an integer. When we terminate, if we did not fail, we will have referenced the item in the store once (if we had any store), and will have a positive integer, and thus will be in $\mathcal{V}[\![+0]\!].$, as needed.

The other conversion function,$C_{+0 \mapsto int}(e)$, is actually easier! As we are given (after evaluation) a positive integer in an empty store, so we don't actually have to do anything—the conversion can be the identity!

This shows that, at least, at integer types, we can call between these two languages safely – which means, programs with these foreign calls and the realizing coercions inserted will still satisfy the relation, and provided that the relation captures the safety from each language that we desire, our FFI has not disturbed it!

Can we do this for more complex types? $\tau_1 \otimes \tau_2 \sim \tau_1 \times \tau_2$ is possible, provided that $\tau_1 \sim \tau_1$ and $\tau_2 \sim \tau_2$, with the following conversions:

$$C_{\tau_1 \otimes \tau_2 \mapsto \tau_1 \times \tau_2}(e) = \text{let}\,(x, y)\,\text{in}\,(C_{\tau_1 \mapsto \tau_1}(x), C_{\tau_2 \mapsto \tau_2} y)$$

We can see that from the evaluation relation, we know that e will reduce to a pair where the values, with given stores, are in the corresponding $\mathcal{V}$ relation. This then means they can be converted using the coercions on the respective types (e.g., if the conversion were $int \sim +0$, we could use the above conversion). The stores are the tricky part, but we know the entirety of the resulting store is split between v_1 and v_2 in the pair, and the coercion means that somehow, these stores were eliminated (if they even existed).

Going the other direction is essentially the same thing, just applying conversions in the reverse direction.

What about arrow types? i.e., prove the following sound:

$$\frac{(C_{\tau_1 \mapsto \tau_1}, C_{\tau_1 \mapsto \tau_1}) : \tau_1 \sim \tau_1 \qquad (C_{\tau_2 \mapsto \tau_2}, C_{\tau_2 \mapsto \tau_2}) : \tau_2 \sim \tau_2}{(C_{\tau_1 \multimap \tau_2 \mapsto \tau_1 \to \tau_2}, C_{\tau_1 \to \tau_2 \mapsto \tau_1 \multimap \tau_2}) : \tau_1 \multimap \tau_2 \sim \tau_1 \to \tau_2}$$

Consider first going from a $\tau_1 \to \tau_2$ to a $\tau_1 \multimap \tau_2$. We know, from the logical relation, that we will have an expression e that will evaluate to a lambda with no store fragment—i.e., it will be closed. To show that that is in $\mathcal{V}[\![\tau_1 \multimap \tau_2]\!].$ we need to show that given an arbitrary (σ_a, v) from $\mathcal{V}[\![\tau_1]\!].$, the lambda applied to v will be in $\mathcal{E}[\![\tau_2]\!].$ with σ_a as the store. From our hypothesis, we know we have a $C_{\tau_1 \mapsto \tau_1}(\cdot)$ that we can wrap around v, such that running it with σ_a results in a value in $\mathcal{V}[\![\tau_1]\!].$. That, in turn, means applying our function (which is in $\mathcal{V}[\![\tau_1 \to \tau_2]\!].$) will result in an expression in $\mathcal{E}[\![\tau_2]\!].$, and we can use our other conversion, $C_{\tau_2 \mapsto \tau_2}(\cdot)$, to wrap this to get a term in $\mathcal{E}[\![\tau_2]\!].$, and in particular one with an empty store (which is perfectly fine for $\mathcal{V}[\![\tau_1 \multimap \tau_2]\!].$).

The other direction, however, causes problems. By assumption, we have an expression in $\mathcal{E}[\![\tau_1 \multimap \tau_2]\!].$, which means it reduces to a function value with some store σ_f. That store is the essence of the problem. Because only a function without a store can be in $\mathcal{V}[\![\tau_1 \to \tau_2]\!].$. We could inline all of the bindings (which, since our languages are pure, wouldn't change the meaning),

which would produce such a lambda, but we would no longer know anything about the lambda—in particular, this means we couldn't use our conversions, as in the other direction, to prove that our function applied to an argument is in the relation. What this highlights is that the problem with linear functions is actually when they are open: and indeed, the fact that our foreign function calls work at all relies upon the functions that are called being *closed*.

This probably seems backwards: ones intuition is that linear functions should be easy to treat as unrestricted functions, and that it is the unrestricted functions that would be difficult to account for. But note that while we were able to show a conversion for unrestricted functions, our hypothesis was quite strong: for such a function to be able to be soundly used in a linear context, we need to be able to convert all linear arguments to unrestricted arguments. Intuitively, this amounts to consuming them at the call-site by copying them to an unrestricted version. Assuming that can be done, which it certainly cannot for all types, there are no longer any linear variables we need concern ourselves with, and this is why our unrestricted function can be used "safely".

8 BENEFITS OF THE APPROACH

We have now shown how it is possible to take a pair of languages with compilers to a common target and define an FFI between them and prove that the FFI maintains type soundness. The approach had several critical features:

(1) The logical relation, which allows us to prove soundness, is inhabited by target terms, which allows us to express the behavior of both of our source languages and coercions between them. This is how real languages work as well, as FFIs are built based on shared conventions on whatever target platform is used.
(2) We begin with a sound basis: no interactions at all. We then only allow interactions that, by definition, maintain soundness. This means, in particular, that certain types are impossible to translate, but this reflects the reality that certain types are unsafe to allow across boundaries (and this is implemented in certain FFIs, e.g., Haskell's).
(3) While our setting is idealized, it is critical that our target language does not have static features that nonetheless are critical invariants of our source language – e.g., the sign of integers in Index$^{\pm}$. While we may have to instrument the target in some ways, as we did here to add variable-use counts, the general idea is that the target can be low-level, untyped, etc. This reflects the reality of most compiler IRs, and the necessity that target language have enough positive expressivity (using the authors terminology developed in [Patterson and Ahmed 2017]) to account for any *behavior* in the source language. While negative expressivity (same citation) may be useful for separate compilation and linking, in the context of this work, most of it is instead embedded in the logical relation, and thus is an artifact of the proof, rather than needing to exist in the target language itself.

In future work, we plan to investigate a realizability model based on a binary logical relation rather than a unary one as we have done here. With a binary relation we can better capture relational properties of source types. We also plan to investigate notions of type precision between the types of the two source languages and how this affects migration from one language to another and associated properties akin to the gradual guarantee in gradually typed languages. This involves switching from a symmetric type convertibility relation to a directed refinement. Finally, there is much work to be done in applying this methodology to numerous pairs of practical source languages, using this technique to design type-sound FFIs between languages.

REFERENCES

Nick Benton and Chung-Kil Hur. 2009. Biorthogonality, Step-indexing and Compiler Correctness. In *Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming (ICFP '09)*. ACM, New York, NY, USA, 97–108.

https://doi.org/10.1145/1596550.1596567

Nick Benton and Nicolas Tabareau. 2009. Compiling functional types to relational specifications for low level imperative code. In *Proceedings of TLDI'09: 2009 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation, Savannah, GA, USA, January 24, 2009*. 3–14.

Max S. New and Amal Ahmed. 2018. Graduality from Embedding-Projection Pairs. *Proceedings of the ACM on Programming Languages* 2, ICFP, 73:1–73:30.

Max S New, Daniel R Licata, and Amal Ahmed. 2019. Gradual type theory. *Proceedings of the ACM on Programming Languages* 3, POPL (2019), 15:1–15:31.

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

Jianzhou Zhao, Qi Zhang, and Steve Zdancewic. 2010. Relational parametricity for a polymorphic linear lambda calculus. In *Asian Symposium on Programming Languages and Systems*. Springer, 344–359.