Semantic Soundness for Language Interoperability

ANONYMOUS AUTHOR(S)

Interoperability with other languages is a feature supported by most existing languages, typically in the form of foreign function calls and a foreign function interface (FFI). Even between type-safe languages, interoperability is fraught with danger as foreign calls may well violate type-soundness guarantees. In their seminal 2007 paper, Matthews and Findler proposed a technique for modeling multi-language systems by augmenting interoperating languages with a pair of *boundaries* that allow code from the first language to be used in contexts of the second, and vice versa. While this technique has been widely applied, their source-to-source interoperability model doesn't reflect how multi-language systems are implemented in the wild, where typically interoperability takes place after compilation to a common target and is mediated by insertion of "glue code" that handles conversions.

In this paper, we demonstrate a framework for proving type soundness of FFIs between type-safe languages that follows an interoperability-after-compilation strategy. We first show how an implementor can type check a foreign function call, which requires that they specify a *convertibility* relation $\tau_A \leq \tau_B$ which says that we may convert—by providing glue code, or conversions, at the target level—between data of type τ_A from language A and data of type τ_B from language B. Second, we show how by giving a semantic model of source-language types as sets of target-language terms, we can establish soundness of conversions: i.e., whenever $\tau_A \leq \tau_B$, the corresponding pair of conversions (glue code) convert target terms that behave like τ_A to target terms that behave like τ_B, and vice versa.

We also show how to reason about *code migration*, which is inspired by the gradual guarantee in gradual typing. We provide a method for showing that, if $\tau_A \leq \tau_B$, then replacing some code of type τ_B written in language B with code of type τ_A written in language A might introduce errors but will not otherwise change the behavior of the program.

Note: We typeset our source languages in `different` **colors**. *This paper will be more pleasant to read in color.*

1 INTRODUCTION

All large-scale practical language implementations come with some way of interoperating with code written in a different language, usually via a foreign function interface (FFI). This is allows development of software systems that include different components written in different languages, whether to support legacy libraries or different paradigms. For instance, you might have a system with a high-performance data store written in Rust interoperating with business logic implemented in OCaml. This interoperability is supported by tools and libraries that insert boilerplate or "glue code" to mediate between the two languages (such as the binding generator SWIG [Beazley 1996], C->HS [Chakravarty 1999], OCaml-ctypes [Yallop et al. 2018], NLFFI [Blume 2001], etc).

Matthews and Findler [2007] noted that while there were numerous FFIs that supported interoperation between languages, there had been no effort to study the semantics of interoperability. They proposed a simple and elegant model for abstractly studying interoperability between a language A and B by embedding the existing syntax and semantics into a multi-language AB and adding boundaries to mediate between the two. Specifically a boundary $\mathcal{AB}(\cdot)$ allows a term e_B to be embedded in an A context and a boundary $\mathcal{BA}(\cdot)$ allows a term e_A to be embedded in a B context. Boundaries are given operational semantics where the basic idea is that a term such as $\mathcal{AB}(e_B)$ is evaluated using the B language semantics to $\mathcal{AB}(v_B)$ and then a type-directed conversion converts the value v_B of type τ_B to a corresponding term of a specified type τ_A. This framework allows flexibility in defining what types can be converted at all and what type they should be converted to. Using this framework, by proving type safety for the multi-language, which includes the typing

2020. 2475-1421/2020/1-ART1 $15.00
https://doi.org/

rules of the embedded language and the typing rules of the boundaries, one is able to show that the entire multi-language system is sound. This framework has served as a foundation for a significant amount of work on language interoperability: between source and target languages of compilers ([Ahmed and Blume 2011; New et al. 2016; Perconti and Ahmed 2014]), between simple and dependently typed languages ([Osera et al. 2012]), between languages with unrestricted and substructural types ([Scherer et al. 2018; Tov and Pucella 2010]), and between a high-level functional language and assembly ([Patterson et al. 2017]).

Unfortunately, while Matthews-Findler-style boundaries give an elegant abstract model for interoperability between languages, they don't connect to actual implementations of interoperability. To date, there still does not exist a way to give semantics for interoperating languages as implemented "in the wild". Actual FFI implementations involve compiling code from the source languages to a common target, and insertion of glue code at the boundaries between components that originated in different languages. The job of the glue code is to implement foreign function calls and to convert data represented in one language to the appropriate representation for the other. While we could show that a Matthews-Findler-style multi-language system is compiled in a semantics-preserving way, which would require showing that the multi-language boundaries are correctly realized/implemented by this glue code, we aim in this paper to avoid doing compiler verification for multi-language systems and instead focus on a lighter-weight technique for type soundness of FFIs.

Directly reasoning about type soundness of FFIs is challenging. For an FFI to be sound, it has to be the case that the glue code correctly converts between the source types τ_A and τ_B, and raises errors when a value cannot be correctly converted. However, this conversion happens not at the level of the source languages, as is the case in the multi-language semantics, but rather at the level of the compiled target code that implements those types. For example, if typed languages are compiled to an untyped target, there is a non-trivial gap between the source and target that makes reasoning about these conversions hard.

It's worth noting that an FFI implementor already has certain notions of correct conversions in their head. There is normally no formal specification (available to the programmer, say in the form of a type checker), but usually there is documentation or tooling that indicates which types from language A can be converted to other types from language B.

In this paper, we present a framework for proving type soundness of FFIs between (statically or dynamically) type-safe languages. Our framework requires first that an FFI implementer specifies a convertibility judgment $\tau_A \leq \tau_B$ that indicates when a type τ_A can be safely converted to τ_B. The judgment indicates that the reverse conversion, while it exists, may raise an error at runtime. Next, we set up a *model* for source-language types as sets of target terms that "behave" as dictated by the source type. Then we use this model to prove not just semantic type soundness of the original languages, but that the glue code written by the FFI implementor satisfies the convertibility judgment: a conversion, when given terms that behave like source type τ_A, results in code that behaves like source type τ_B, and vice versa.

To illustrate the above framework, we consider two source languages with the ability to call functions in the other, and a target language they compile to. Our source languages are both statically typed functional languages: `MiniML`, an ML-like language with polymorphism and dynamically allocated mutable references, and **Affi**, and a language with affine types. These compile to an untyped target language, *Target*, allowing us to demonstrate how to consider semantic soundness for the most prevalent form of interoperability, which is via compilation to an untyped target. In general, we will see that we require features in the target in order to define a model that captures our source-type invariants, which means that one could use a target language that is statically typed.

This work also takes some inspiration from gradual typing. It has long been thought that multi-language systems are in some sense a generalization of gradual typing, which allows mixing of static and dynamically typed code, but requires that the underlying term language is the same. By contrast, in multi-language systems we are mixing not just more and less precise terms from the same language but from languages that do not share term syntax. Nonetheless, there is an important concept in gradual languages called the gradual guarantee, derived from *type and term precision* that says: if a term with type τ is given a more precise type τ' (with fewer instances of the "unknown" type ?), this results in a program that may have more errors but otherwise behaves the same.

In a multi-language software system, we might want to replace code written in a language A with code written in a language B, and it would be nice to have guarantees about code migration along the same lines as the gradual guarantee ([Siek et al. 2015]). While there is no canonical syntactic way to define type precision in a multi-language setting, we show in §7 that one can set up the convertibility relation such that $\tau_A \leq \tau_B$ means that τ_A is "semantically more precise" than τ_B. Drawing inspiration from [New and Ahmed 2018] on embedding-projection pairs (ep-pairs), we show that our conversions satisfy what they call the ep-pair property: that if τ_A is more precise than τ_B (i.e., $\tau_A \leq \tau_B$) then converting from τ_A to τ_B and back is the identity, whereas converting from τ_B to τ_A and back may produce errors.

Proofs elided from this paper are provided in our anonymous supplementary materials.

2 MAIN IDEAS

In order to study soundness in the context of foreign function calls, we must consider two languages. In this section, we will call them $\mathtt{M}$ and $\mathtt{A}$.

Typechecking Foreign Calls. Consider a foreign call $\mathtt{e_M}\ \mathtt{e_A}$. How would we typecheck such an application?

$$\frac{\vdash \mathtt{e_M} : \tau_\mathtt{M} \to \tau'_\mathtt{M} \qquad \Gamma \vdash \mathtt{e_A} : \tau_\mathtt{A}}{\Gamma \vdash \mathtt{e_M}\ \mathtt{e_A} : ?}$$

In order for this to make sense, we need to know that that the input of type $\tau_\mathtt{A}$ is convertible to type $\tau_\mathtt{M}$ and then that the result returned by language $\mathtt{M}$ of type $\tau_\mathtt{M}$ can be converted to some $\mathtt{A}$ language type $\tau_\mathtt{A}$. To do this, we can define "convertibility" judgment where we declaratively specify what types in language $\mathtt{M}$ can be converted to language $\mathtt{A}$. (Note that we only consider calling closed foreign functions, a decision we discuss further in §9.3.)

We note a few things about the convertibility relation we would like to define, informed by how such conversions are carried out in practice. First, sometimes these conversions will always succeed, in both directions, but sometimes one direction may fail. The former indicates two types are isomorphic: it may be that the two types are indeed just different names for the exact same values (perhaps, twos complement 64 bit integers), and no conversion need happen, or it could be that they are not identical, but express the same values (e.g., a boolean value and a tagged sum with exactly two variants). In the second case, where one direction of conversion may fail, one type has more possible values than the other, as when a precise type gets encoded into a more general value (e.g., a tagged variant gets converted to a bit array, where not all bit patterns are actual variants).

To capture this directionality, our convertibility judgment has the shape $\tau \leq \tau'$ if τ can always be converted to τ' but the reverse may fail (here τ and τ' are types drawn from different languages, but either could be from $\mathtt{M}$ or $\mathtt{A}$). Even though our convertibility relation is capturing this directionality,

when typechecking a foreign function call, conversion should be allowed in either direction, as foreign calls should allow the possibility of dynamic failure. We write $\tau \gtrless \tau'$ to indicate either $\tau \leq \tau'$ or $\tau' \leq \tau$ holds. We can now specify our foreign call typing rule as follows:

$$\frac{\vdash \mathsf{e_M} : \tau_\mathsf{M} \to \tau'_\mathsf{M} \qquad \Gamma \vdash \mathbf{e_A} : \tau_\mathbf{A} \qquad \tau_\mathsf{M} \gtrless \tau_\mathbf{A} \qquad \tau'_\mathsf{M} \gtrless \tau'_\mathbf{A}}{\Gamma \vdash \mathsf{e_M}\, \mathbf{e_A} : \tau'_\mathbf{A}}$$

Convertibility and Conversions. Where do these convertibility rules come from? They must be defined by the implementor of the FFI. Ideally the source language type system would have a convertibility judgment like this that could be extended over time with additional convertibility rules.

Of course, convertibility rules is not the only thing that an implementor needs to specify: actual target-level conversion ("glue") code is needed as well. To reflect this, we write the convertibility judgment in the following form:

$$C_{\tau_A \mapsto \tau_B}, C_{\tau_B \mapsto \tau_A} : \tau_A \leq \tau_B$$

This judgment is meant to indicate that there is a conversion $C_{\tau_A \mapsto \tau_B}(\cdot)$ that given e (which should behave as a τ_A) will produce code that behaves as a τ_B, and the other conversion does the reverse. As noted, the former should never cause a runtime failure, but the latter may. Of course, this is aspirational, and we will prove later that this is indeed the case.

These rules may be inductively defined, e.g., the conversions for a pair may use those for the types of its elements:

$$\frac{C_{\tau_{A1} \mapsto \tau_{B1}}, C_{\tau_{B1} \mapsto \tau_{A1}} : \tau_{A1} \leq \tau_{B1} \qquad C_{\tau_{A2} \mapsto \tau_{B2}}, C_{\tau_{B2} \mapsto \tau_{A2}} : \tau_{A2} \leq \tau_{B2}}{C_{\tau_{A1} \times \tau_{A2} \mapsto \tau_{B1} \times \tau_{B2}}, C_{\tau_{B1} \times \tau_{B2} \mapsto \tau_{A1} \times \tau_{A2}} : \tau_{A1} \times \tau_{A2} \leq \tau_{B1} \times \tau_{B2}}$$

This allows not only the type checker to use the rules to decide if a foreign call is okay, but also the compiler to then insert appropriate conversions when generating code. Hence, the foreign call above, $\mathsf{e_M}\, \mathbf{e_A}$ will compile to $C_{\tau'_\mathsf{M} \mapsto \tau'_\mathbf{A}}(\mathsf{e_M}^+\, C_{\tau_\mathbf{A} \mapsto \tau_\mathsf{M}}(\mathbf{e_A}^+))$, where we write e^+ to indicate compilation.

Proving Foreign Calls Sound. What is described above is a good implementation story, but if convertibility judgments can be freely written, there is no guarantee of soundness. In order to prove that the target-level glue code is sound, we need to be able to show that it correctly converts target code that behaves like type τ_A into target code that behaves like type τ_B. As we briefly discussed in §1, we do this by defining a model (a logical relation) of source-language types inhabited by target-level terms that behave appropriately. (Such models are often called realizability models because the logical relation is inhabited by target terms that realize source types [Benton and Tabareau 2009]). In essence we would like to use the well-established technique of semantic type soundness to prove type soundness for the source languages and the typing of foreign calls.

Typically to prove semantic type soundness we must show that if we can syntactically derive that $\vdash e : \tau$ then e belongs to the interpretation of τ. In our case, however, since the logical relation is inhabited by target terms, we must show that the compiled version of e, written e^+, is in the interpretation of τ. In particular, we need to prove that our target-level conversions satisfy the model, as we need to that to prove the soundness of the foreign call rule.

We choose this semantic approach to type soundness not because it makes proving soundness of the two source languages easier (it does not!), but because it allows us to bring the two languages together, and prove that the foreign calls and conversions that mediate them are sound. By giving meaning to types in terms of a common set of target terms, we can talk about a type in one language being equivalent to a type in the other language (if those sets are the same), or about a wrapped term from one language behaving as a type from another.

Reasoning About Code Migration. As we discussed above, the convertibility relation we defined has a direction. Recall that above we said that the convertibility judgment has a directionality to it. We write $\tau_A \leq \tau_B$ when we expect conversions from τ_A to τ_B to always succeed but τ_B to τ_A may fail. Specifically, what we want to be able to show is that our two conversions form an embedding-projection pair, that is that converting from τ_A to τ_B and back is the identity, whereas converting from τ_B to τ_A and back may produce errors. In order to prove that, we define a binary logical relation that captures error approximation: i.e., it says that e_1 is related to e_2 if e_1 errors or e_1 and e_2 behave the same way. Our logical relation also provides users with a proof method for reasoning about code migration. Specifically, one can use the logical relation to reason about the replacement of code from one language to the other and the resulting error behavior. We discuss this further in §7.

3 INTEROPERATING SOURCE LANGUAGES AND A TARGET LANGUAGE

As discussed in §1, we formally demonstrate our idea for proving semantic type soundness in the presence of foreign calls by considering two source languages, MiniML and Affi, which we alluded to in §2 as *F* and *A*. The former is a functional language with polymorphism and mutable references: it is intended to model a scaled down ML, minimal to allow us to focus on the techniques and the issue of interoperability, but with enough features so as not to be trivial. The latter is an affine lambda calculus, built by allowing weakening in a linear lambda calculus. We do this by allowing affine resources to go unused in the terminal rules, rather than defining a single weakening rule.

3.1 Source Languages: MiniML and Affi

We present the syntax and static semantics for MiniML in Figure 1. We typeset MiniML terms and environments in blue typewriter font. The language is quite standard, with base types unit and int, pairs with projection fst and snd, tagged sums with a pattern matching construct match, term-level and type-level abstraction and application, and ML-style mutable references ref τ.

The typing judgment is standard, with the form $\Delta; \Gamma \vdash \mathtt{e} : \tau$ where the type environment Δ tracks type variables α and the term environment Γ keeps track of term variables and their types: $\mathtt{x} : \tau$. Note that in Figure 1 we have shown syntax, but not yet typing, for a foreign function call: $\mathbf{e}\,\mathtt{e}$. Until we add a typing rule for such calls (see §5), any program with such a call would be ill-typed.

We present the syntax and static semantics for our affine language Affi in Figure 2. We typeset Affi terms and environments in **orange bold font**. As with MiniML, we have base types **unit** and **int**. The affine (use-at-most-once) resources in this language are variables **a**, which are introduced by functions $\lambda \mathbf{a} : \boldsymbol{\tau}.\mathbf{e}$ of type $\boldsymbol{\tau} \multimap \boldsymbol{\tau}'$. This can be seen as a vastly simplified model of a language like Rust, rather than a language that models ownership of the heap (as is more typical in separation logic and modeled, for instance, in L^3 [Ahmed et al. 2007]). Other types include a duplicable type $!\boldsymbol{\tau}$, which necessarily cannot include any affine resources; a lazy product $\boldsymbol{\tau} \& \boldsymbol{\tau}$ for which only one component of the product can be projected (used); and a strict product $\boldsymbol{\tau} \otimes \boldsymbol{\tau}$ for which both components can be used, but each may be used at most once since the language is affine.

Types τ := $\texttt{unit} \mid \texttt{int} \mid \tau \times \tau \mid \tau + \tau \mid \tau \to \tau \mid \forall\alpha.\tau \mid \alpha \mid \texttt{ref}\ \tau$

Expressions $\texttt{e}$:= $() \mid \mathbb{Z} \mid \texttt{x} \mid (\texttt{e},\texttt{e}) \mid \texttt{fst e} \mid \texttt{snd e} \mid \texttt{inl e} \mid \texttt{inr e} \mid \texttt{match e x\{e\} y\{e\}} \mid \lambda \texttt{x} : \tau.\texttt{e} \mid \texttt{e e} \mid \Lambda\alpha.\texttt{e} \mid \texttt{e}\langle\tau\rangle \mid \texttt{ref e} \mid \texttt{!e} \mid \texttt{e} := \texttt{e} \mid \texttt{e e}$

$$\frac{}{\Delta;\Gamma \vdash () : \texttt{unit}} \qquad \frac{}{\Delta;\Gamma \vdash \mathbb{Z} : \texttt{int}} \qquad \frac{\Delta \vdash \tau \qquad \texttt{x} : \tau \in \Gamma}{\Delta;\Gamma \vdash \texttt{x} : \tau} \qquad \frac{\Delta;\Gamma \vdash \texttt{e}_1 : \tau_1 \qquad \Delta;\Gamma \vdash \texttt{e}_2 : \tau_2}{\Delta;\Gamma \vdash (\texttt{e}_1,\texttt{e}_2) : \tau_1 \times \tau_2}$$

$$\frac{\Delta;\Gamma \vdash \texttt{e} : \tau_1 \times \tau_2}{\Delta;\Gamma \vdash \texttt{fst e} : \tau_1} \qquad \frac{\Delta;\Gamma \vdash \texttt{e} : \tau_1 \times \tau_2}{\Delta;\Gamma \vdash \texttt{snd e} : \tau_2} \qquad \frac{\Delta \vdash \tau_2 \qquad \Delta;\Gamma \vdash \texttt{e} : \tau_1}{\Delta;\Gamma \vdash \texttt{inl e} : \tau_1 + \tau_2} \qquad \frac{\Delta \vdash \tau_1 \qquad \Delta;\Gamma \vdash \texttt{e} : \tau_2}{\Delta;\Gamma \vdash \texttt{inr e} : \tau_1 + \tau_2}$$

$$\frac{\Delta;\Gamma \vdash \texttt{e} : \tau_1 + \tau_2 \qquad \Delta;\Gamma[\texttt{x} : \tau_1] \vdash \texttt{e}_1 : \tau \qquad \Delta;\Gamma[\texttt{y} : \tau_2] \vdash \texttt{e}_2 : \tau}{\Delta;\Gamma \vdash \texttt{match e x}\{\texttt{e}_1\}\ \texttt{y}\{\texttt{e}_2\} : \tau} \qquad \frac{\Delta;\Gamma[\texttt{x} : \tau_1] \vdash \texttt{e} : \tau_2}{\Delta;\Gamma \vdash \lambda \texttt{x} : \tau_1.\texttt{e} : \tau_1 \to \tau_2}$$

$$\frac{\Delta;\Gamma \vdash \texttt{e} : \tau' \to \tau \qquad \Delta;\Gamma \vdash \texttt{e}' : \tau}{\Delta;\Gamma \vdash \texttt{e e}' : \tau} \qquad \frac{\Delta,\alpha;\Gamma \vdash \texttt{e} : \tau}{\Delta;\Gamma \vdash \Lambda\alpha.\texttt{e} : \forall\alpha.\tau} \qquad \frac{\Delta \vdash \tau \qquad \Delta;\Gamma \vdash \texttt{e} : \forall\alpha.\tau}{\Delta;\Gamma \vdash \texttt{e}\langle\tau'\rangle : \tau[\tau'/\alpha]}$$

$$\frac{\Delta;\Gamma \vdash \texttt{e} : \tau}{\Delta;\Gamma \vdash \texttt{ref e} : \texttt{ref}\ \tau} \qquad \frac{\Delta;\Gamma \vdash \texttt{e} : \texttt{ref}\ \tau}{\Delta;\Gamma \vdash \texttt{!e} : \tau} \qquad \frac{\Delta;\Gamma \vdash \texttt{e}_1 : \texttt{ref}\ \tau \qquad \Delta;\Gamma \vdash \texttt{e}_2 : \tau}{\Delta;\Gamma \vdash \texttt{e}_1 := \texttt{e}_2 : \texttt{unit}}$$

Fig. 1. Syntax and static semantics for MiniML.

An important detail to highlight regarding the term language is that we use variables **x** for unrestricted (non-affine) bindings and variables **a** for affine bindings that introduce a use-at-most-once resource. Our affine language also has syntax for a foreign call, **e e**, which again does not yet have a typing rule.

The elimination form for duplicable types is $!\tau$ is **let !x = e in e′**; it introduces unrestricted (non-affine) bindings when type checking the body **e′**. The elimination form for strict products $\tau \otimes \tau$ is the destructuring **let (a, a′)**, while lazy products $\tau \& \tau$ are eliminated using **e.1** and **e.2**, which denote the first and second projection from the lazy product. Note that projecting out uses up the lazy product as there is no way to project both components of a lazy pair. A related point to note is that a lazy product value form contains expressions rather than values. Even though we do not provide an operational semantics for **Affi**, for clarity, we show the grammar for **Affi** values in Figure 2 as some of it is non-standard, and these value forms will later guide compilation.

The static semantics for **Affi** are relatively standard. Typing judgments have the form $\Gamma; \Omega \vdash \texttt{e} : \tau$ where the unrestricted environment Γ keeps track of unrestricted variables **x** and their types, while the affine environment Ω keeps track of affine variables **a** and their types. Since the language is affine (not linear), the terminal rules—i.e., the ones for type checking affine and unrestricted variables (**a** and **x**), and unit and integer values—do not require that Ω be empty. When type checking term abstraction $\lambda \texttt{a} : \tau.\texttt{e}$, we add affine bindings to Ω. Most rules split the affine environment when typechecking subterms, with a few notable exceptions: a term **!e** must have no affine variables, because assigning it the type $!\tau$ will allow it to be duplicated freely; and for the lazy product $\langle \texttt{e}_1, \texttt{e}_2 \rangle$, since elimination will only allow one of the subterms to be used, the same affine environment Ω is used when type checking both subterms.

Finally, note that we do not give operational semantics for MiniML or **Affi**, but instead give compilers from both languages (see §4) to a target language with a formally specified operational semantics. The reader should view this as: the semantics of our source languages is "defined" via compilation, as is common for most language implementations. Of course, there is no reason why

$$
\begin{array}{lrcl}
\text{Types} & \tau & ::= & \mathbf{unit} \mid \mathbf{int} \mid \tau \multimap \tau \mid !\tau \mid \tau \& \tau \mid \tau \otimes \tau \\
\text{Expressions} & \mathbf{e} & ::= & () \mid \mathbf{n} \mid \mathbf{x} \mid \mathbf{a} \mid \lambda \mathbf{a} : \tau.\mathbf{e} \mid \mathbf{e}\ \mathbf{e} \mid !\mathbf{e} \mid \mathbf{let}\ !\mathbf{x} = \mathbf{e}\ \mathbf{in}\ \mathbf{e}' \mid \\
 & & & \langle \mathbf{e}, \mathbf{e}' \rangle \mid \mathbf{e}.1 \mid \mathbf{e}.2 \mid (\mathbf{e}, \mathbf{e}) \mid \mathbf{let}\ (\mathbf{a}, \mathbf{a}') = \mathbf{e}\ \mathbf{in}\ \mathbf{e}' \mid \mathbf{e}\ \mathbf{e} \\
\text{Values} & \mathbf{v} & ::= & () \mid \lambda \mathbf{a} : \tau.\mathbf{e} \mid !\mathbf{e} \mid \langle \mathbf{e}, \mathbf{e}' \rangle \mid (\mathbf{v}, \mathbf{v}')
\end{array}
$$

$$
\frac{\mathbf{a} : \tau \in \Omega}{\Gamma; \Omega \vdash \mathbf{a} : \tau} \qquad \frac{\mathbf{x} : \tau \in \Gamma}{\Gamma; \Omega \vdash \mathbf{x} : \tau} \qquad \frac{}{\Gamma; \Omega \vdash () : \mathbf{unit}} \qquad \frac{}{\Gamma; \Omega \vdash \mathbf{n} : \mathbf{int}}
$$

$$
\frac{\Gamma; \Omega, \mathbf{a} : \tau_1 \vdash \mathbf{e} : \tau_2}{\Gamma; \Omega \vdash \lambda \mathbf{a} : \tau_1.\mathbf{e} : \tau_1 \multimap \tau_2} \qquad \frac{\Gamma; \Omega_1 \vdash \mathbf{e}_1 : \tau_1 \multimap \tau_2 \qquad \Gamma; \Omega_2 \vdash \mathbf{e}_2 : \tau_1}{\Gamma; \Omega_1 \uplus \Omega_2 \vdash \mathbf{e}_1\ \mathbf{e}_2 : \tau_2}
$$

$$
\frac{\Gamma; \cdot \vdash \mathbf{e} : \tau}{\Gamma; \cdot \vdash !\mathbf{e} : !\tau} \qquad \frac{\Gamma; \Omega_1 \vdash \mathbf{e} : !\tau \qquad \Gamma, \mathbf{x} : \tau; \Omega_2 \vdash \mathbf{e}' : \tau'}{\Gamma; \Omega_1 \uplus \Omega_2 \vdash \mathbf{let}\ !\mathbf{x} = \mathbf{e}\ \mathbf{in}\ \mathbf{e}'}
$$

$$
\frac{\Gamma; \Omega \vdash \mathbf{e}_1 : \tau_1 \qquad \Gamma; \Omega \vdash \mathbf{e}_2 : \tau_2}{\Gamma; \Omega \vdash \langle \mathbf{e}_1, \mathbf{e}_2 \rangle : \tau_1 \& \tau_2} \qquad \frac{\Gamma; \Omega \vdash \mathbf{e} : \tau_1 \& \tau_2}{\Gamma; \Omega \vdash \mathbf{e}.1 : \tau_1} \qquad \frac{\Gamma; \Omega \vdash \mathbf{e} : \tau_1 \& \tau_2}{\Gamma; \Omega \vdash \mathbf{e}.2 : \tau_2}
$$

$$
\frac{\Gamma; \Omega_1 \vdash \mathbf{e}_1 : \tau_1 \qquad \Gamma; \Omega_2 \vdash \mathbf{e}_2 : \tau_2}{\Gamma; \Omega_1 \uplus \Omega_2 \vdash (\mathbf{e}_1, \mathbf{e}_2) : \tau_1 \otimes \tau_2} \qquad \frac{\Gamma; \Omega_1 \vdash \mathbf{e} : \tau_1 \otimes \tau_2 \qquad \Gamma; \Omega_2, \mathbf{a} : \tau_1, \mathbf{a}' : \tau_1 \vdash \mathbf{e}' : \tau'}{\Gamma; \Omega_1 \uplus \Omega_2 \vdash \mathbf{let}\ (\mathbf{a}, \mathbf{a}') = \mathbf{e}\ \mathbf{in}\ \mathbf{e}' : \tau'}
$$

Fig. 2. Syntax and static semantics for Affi.

we cannot give formal operational semantics for both languages, but we note that we will not need to do so for the purpose of proving semantic soundness of both languages and their foreign function calls.

3.2 Target Language: *Target*

Our *Target* language and its syntax and small-step operational semantics is presented in Figure 3. We typeset the terms and environments of our target language in *black italic font*. This language is an untyped (or dynamically typed) lambda calculus with pairs, sums, mutable references, and *dynamic failure*. The last is important so that we can encode conversions with assertions that can signal (well-defined) failure if, in essence, a converted value does not conform to what is expected.

As mentioned in §1, we choose an untyped target language since that reflects what most compilers do. However, the reader may be surprised to see that our target language is a fairly high-level language rather than, say, assembly or LLVM IR. While we could have picked a lower-level language—and indeed we want future practical efforts to do precisely that—since our goal in this paper is to illustrate a framework for proving soundness of FFIs, we have tried to avoid the unnecessary complexity a lower-level target would add to our presentation. In practice, however, what we have in mind is that the chosen target language should be the language where interop *actually* happens, which means it may be low-level or fairly high level: e.g., WebAssembly, JVM bytecode, or Racket (if the interoperating source languages are Racket DSLs).

A further consideration is that the target language must be rich enough (in statics/types or in dynamics/runtime-tracking) to be able to capture the invariants or requirements of source language type systems. For our case study, since we are compiling to a dynamically typed language from Affi, a language with affine types, and from MiniML, which has only unrestricted types, we need some mechanism in our target to keep track of the affine resources. Hence, our *Target* language also, critically, has affine variables, which are indicated with underscores, so affine let is written

let $\underline{x}$, and an affine lambda is written $\lambda\underline{x}$. Technically speaking, this is necessary in order to build the logical relation later: we need to be able to capture the behavior of our source languages—in this case, we need to make sure that affine resources are not used twice. While there are other ways to capture the affine invariants without building them into the language, they would be clumsier and introduce more noise into the semantics. Excluding the introduction and uses of affine variables, our syntax is quite standard.

The *Target* language operational semantics is defined as small-step transitions on machine configurations $\langle \sigma, \gamma, e \rangle$, where σ is the heap (mapping locations ℓ to values), γ is the variable store (mapping variables x or $\underline{x}$ to values), and e is the expression. Note that when bindings are added to the variable store, they are marked as affine or not, so that there are two rules for variable lookup: the first one, where the value that comes out $\underline{v}$ is affine, and thus is removed from the variable store, and the second where it is not, and is left alone. Note that, as is standard, locations ℓ cannot be written in programs: they will only appear in reductions after having been created by *ref*.

4 COMPILERS FOR MiniML AND Affi

We compile both our source languages MiniML and **Affi** to out target *Target*. We show both compilers in Figure 4. Both compilers are type-directed, even though we (mostly) write them as if they are not for simplicity; we point out the one place where we need type information that isn't already available in the source syntax.

For MiniML, most of the compiler is essentially a type-erasing identity transformation since our target is an untyped version of our source. One deviation is for type-abstractions, which we turn into term abstractions (normal lambdas) since the body of the type abstraction must be suspended after compilation; and correspondingly for type application, which we turn into application with the unit value.

The other deviation is for the foreign function call. As described in §2, we compile both the function and the argument, and we insert glue code wrappers around first the argument (to ensure it is appropriate for the foreign function) and around the result of the call (to ensure the result is okay in the context). The type τ_2, of course, depends on the compiler knowing the context that the foreign call would take place in, but in a well-typed program (and we only compile well-typed programs), this would be true. That is, our compiler is actually type-directed so here we simply include the relevant type information (which would be available in the source typing derivation) as annotations on the subterms.

For **Affi**, again most of the compiler is an identity transformation, where affine bindings **a** turn into target affine bindings $\underline{a}$, and unrestricted **let !x** becomes a target *let* with an unrestricted target binding. There are, however, a few exceptions. Lazy products have to be compiled to a pair of thunks, as otherwise both sides of the pair would be evaluated before one could be projected, which would get stuck if the source program included an expression like $\langle \mathbf{a}, \mathbf{a} \rangle.\mathbf{1}$. This means that projections from lazy products first project and then force (by passing a unit () value to the function). Destructuring matching on strict pairs also has to be done somewhat carefully, as the pair expression **e** cannot be evaluated more than once. Hence, we bind the translation of **e** to an unrestricted variable x_{fresh} and then use that to project out the two components and allocate an affine binding for each.

5 CONVERTIBILITY AND TYPE-CHECKING FOREIGN CALLS

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 operational failure.

$$\text{Expressions} \quad e := () \mid \mathbb{Z} \mid \ell \mid x \mid \underline{x} \mid (e, e) \mid \mathit{fst}\ e \mid \mathit{snd}\ e \mid \mathit{inl}\ e \mid \mathit{inr}\ e \mid \mathit{match}\ e\ x\{e\}\ y\{e\} \mid \mathit{let}\ x = e\ \mathit{in}\ e \mid \mathit{let}\ \underline{x} = e\ \mathit{in}\ e \mid \lambda x\{e\} \mid \lambda \underline{x}\{e\} \mid e\ e \mid \mathit{ref}\ e \mid !e \mid e := e \mid \mathit{fail}$$

$$\frac{\gamma[x] = \underline{v}}{\langle \sigma, \gamma, x\rangle \to \langle \sigma, \gamma \setminus x, v\rangle} \qquad \frac{\gamma[x] = v}{\langle \sigma, \gamma, x\rangle \to \langle \sigma, \gamma, v\rangle} \qquad \frac{\langle \sigma, \gamma, e_1\rangle \to \langle \sigma', \gamma', e'\rangle}{\langle \sigma, \gamma, (e_1, e_2)\rangle \to \langle \sigma', \gamma, (e', e_2)\rangle}$$

$$\frac{\langle \sigma, \gamma, e_2\rangle \to \langle \sigma', \gamma', e'\rangle}{\langle \sigma, \gamma, (v, e_2)\rangle \to \langle \sigma', \gamma, (v, e')\rangle} \qquad \frac{}{\langle \sigma, \gamma, \mathit{fst}\ (v, v')\rangle \to \langle \sigma, \gamma, v\rangle} \qquad \frac{}{\langle \sigma, \gamma, \mathit{snd}\ (v', v)\rangle \to \langle \sigma, \gamma, v\rangle}$$

$$\frac{\langle \sigma, \gamma, e\rangle \to \langle \sigma', \gamma', e'\rangle}{\langle \sigma, \gamma, \mathit{inl}\ e\rangle \to \langle \sigma', \gamma, \mathit{inl}\ e'\rangle} \qquad \frac{\langle \sigma, \gamma, e\rangle \to \langle \sigma', \gamma', e'\rangle}{\langle \sigma, \gamma, \mathit{inr}\ e\rangle \to \langle \sigma', \gamma, \mathit{inr}\ e'\rangle}$$

$$\frac{\langle \sigma, \gamma, e\rangle \to \langle \sigma', \gamma', e'\rangle}{\langle \sigma, \gamma, \mathit{match}\ e\ x\{e_1\}\ y\{e_2\}\rangle \to \langle \sigma', \gamma, \mathit{match}\ e'\ x\{e_1\}\ y\{e_2\}\rangle}$$

$$\frac{}{\langle \sigma, \gamma, \mathit{match}\ \mathit{inl}\ v\ x\{e_1\}\ y\{e_2\}\rangle \to \langle \sigma, \gamma[x \mapsto v], e_1\rangle} \qquad \frac{}{\langle \sigma, \gamma, \mathit{match}\ \mathit{inr}\ v\ x\{e_1\}\ y\{e_2\}\rangle \to \langle \sigma, \gamma[y \mapsto v], e_2\rangle}$$

$$\frac{\langle \sigma, \gamma, e\rangle \to \langle \sigma', \gamma', e'\rangle}{\langle \sigma, \gamma, \mathit{let}\ x = e\ \mathit{in}\ e_2\rangle \to \langle \sigma', \gamma, \mathit{let}\ x = e'\ \mathit{in}\ e_2\rangle} \qquad \frac{}{\langle \sigma, \gamma, \mathit{let}\ x = v\ \mathit{in}\ e\rangle \to \langle \sigma, \gamma[x \mapsto v], e\rangle}$$

$$\frac{\langle \sigma, \gamma, e\rangle \to \langle \sigma', \gamma', e'\rangle}{\langle \sigma, \gamma, \mathit{let}\ \underline{x} = e\ \mathit{in}\ e_2\rangle \to \langle \sigma', \gamma, \mathit{let}\ \underline{x} = e'\ \mathit{in}\ e_2\rangle} \qquad \frac{}{\langle \sigma, \gamma, \mathit{let}\ \underline{x} = v\ \mathit{in}\ e\rangle \to \langle \sigma, \gamma[\underline{x} \mapsto v], e\rangle}$$

$$\frac{\langle \sigma, \gamma, e\rangle \to \langle \sigma', \gamma', e'\rangle}{\langle \sigma, \gamma, e\ e_2\rangle \to \langle \sigma', \gamma, e'\ e_2\rangle} \qquad \frac{\langle \sigma, \gamma, e\rangle \to \langle \sigma', \gamma', e'\rangle}{\langle \sigma, \gamma, \lambda x\{e_b\}\ e\rangle \to \langle \sigma, \gamma, \lambda x\{e_b\}\ e'\rangle}$$

$$\frac{}{\langle \sigma, \gamma, \lambda x\{e_b\}\ v\rangle \to \langle \sigma, \gamma[x \mapsto v], e_b\rangle} \qquad \frac{\langle \sigma, \gamma, e\rangle \to \langle \sigma', \gamma', e'\rangle}{\langle \sigma, \gamma, \lambda \underline{x}\{e_b\}\ e\rangle \to \langle \sigma, \gamma, \lambda \underline{x}\{e_b\}\ e'\rangle}$$

$$\frac{}{\langle \sigma, \gamma, \lambda \underline{x}\{e_b\}\ v\rangle \to \langle \sigma, \gamma[x \mapsto \underline{v}], e_b\rangle} \qquad \frac{\langle \sigma, \gamma, e\rangle \to \langle \sigma', \gamma', e'\rangle}{\langle \sigma, \gamma, \mathit{ref}\ e\rangle \to \langle \sigma, \gamma, \mathit{ref}\ e'\rangle}$$

$$\frac{\ell \notin \mathrm{dom}(\sigma)}{\langle \sigma, \gamma, \mathit{ref}\ v\rangle \to \langle \sigma[\ell \mapsto v], \gamma, \ell\rangle} \qquad \frac{\langle \sigma, \gamma, e\rangle \to \langle \sigma', \gamma', e'\rangle}{\langle \sigma, \gamma, !e\rangle \to \langle \sigma', \gamma, !e'\rangle} \qquad \frac{\sigma[\ell] = v}{\langle \sigma, \gamma, !\ell\rangle \to \langle \sigma, \gamma, v\rangle}$$

$$\frac{\langle \sigma, \gamma, e\rangle \to \langle \sigma', \gamma', e'\rangle}{\langle \sigma, \gamma, e := e_2\rangle \to \langle \sigma', \gamma, e' := e_2\rangle} \qquad \frac{\langle \sigma, \gamma, e\rangle \to \langle \sigma', \gamma', e'\rangle}{\langle \sigma, \gamma, \ell := e\rangle \to \langle \sigma', \gamma, \ell := e'\rangle} \qquad \frac{}{\langle \sigma, \gamma, \ell := v\rangle \to \langle \sigma[\ell] := v, \gamma, ()\rangle}$$

Note: rules for *fail* are elided above: it simply propagates outward out of any subterm until it is at the top-level (it does not step).

Fig. 3. Syntax and operational semantics for *Target*.

Note that if all that we wanted to do is show semantic type soundness of FFIs (i.e., if we did not want to show migration properties), then the following rule for typechecking foreign calls would suffice:

$()$	$\rightsquigarrow$	$()$
$\mathbb{Z}$	$\rightsquigarrow$	$\mathbb{Z}$
$\mathtt{x}$	$\rightsquigarrow$	x
$(\mathtt{e}_1, \mathtt{e}_2)$	$\rightsquigarrow$	$({\mathtt{e}_1}^+, {\mathtt{e}_2}^+)$
$\mathtt{fst\ e}$	$\rightsquigarrow$	$\mathit{fst}\ \mathtt{e}^+$
$\mathtt{snd\ e}$	$\rightsquigarrow$	$\mathit{snd}\ \mathtt{e}^+$
$\mathtt{inl\ e}$	$\rightsquigarrow$	$\mathit{inl}\ \mathtt{e}^+$
$\mathtt{inr\ e}$	$\rightsquigarrow$	$\mathit{inr}\ \mathtt{e}^+$
$\mathtt{match\ e\ x\{e_1\}\ y\{e_2\}}$	$\rightsquigarrow$	$\mathit{match}\ \mathtt{e}^+\ x\{{\mathtt{e}_1}^+\}\ y\{{\mathtt{e}_2}^+\}$
$\lambda \mathtt{x} : \tau.\mathtt{e}$	$\rightsquigarrow$	$\lambda x.\mathtt{e}^+$
$\mathtt{e}_1\ \mathtt{e}_2$	$\rightsquigarrow$	${\mathtt{e}_1}^+\ {\mathtt{e}_2}^+$
$\Lambda\alpha.\mathtt{e}$	$\rightsquigarrow$	$\lambda\_.\mathtt{e}^+$
$\mathtt{e}\!<\!\tau\!>$	$\rightsquigarrow$	$\mathtt{e}^+\ ()$
$\mathtt{ref\ e}$	$\rightsquigarrow$	$\mathit{ref}\ \mathtt{e}^+$
$!\mathtt{e}$	$\rightsquigarrow$	$!\mathtt{e}^+$
$\mathtt{e}_1 := \mathtt{e}_2$	$\rightsquigarrow$	${\mathtt{e}_1}^+ := {\mathtt{e}_2}^+$
$(\mathtt{e} : \tau_1 \multimap \tau_2)\ (\mathtt{e} : \tau_1)$	$\rightsquigarrow$	$C_{\tau_2 \mapsto \tau_2}(\mathtt{e}^+\ (C_{\tau_1 \mapsto \tau_1}(\mathtt{e}^+))$

$()$	$\rightsquigarrow$	$()$
$\mathbf{x}$	$\rightsquigarrow$	x
$\mathbf{a}$	$\rightsquigarrow$	$\underline{a}$
$\lambda \mathbf{a} : \tau.\mathbf{e}$	$\rightsquigarrow$	$\lambda \underline{a}.\mathbf{e}^+$
$\mathbf{e}_1\ \mathbf{e}_2$	$\rightsquigarrow$	${\mathbf{e}_1}^+\ {\mathbf{e}_2}^+$
$!\mathbf{e}$	$\rightsquigarrow$	$\mathbf{e}^+$
$\mathbf{let\ !x = e\ in\ e'}$	$\rightsquigarrow$	$\mathit{let}\ x = \mathbf{e}^+\ \mathit{in}\ \mathbf{e'}^+$
$\langle \mathbf{e}, \mathbf{e'} \rangle$	$\rightsquigarrow$	$(\lambda\_.\mathbf{e}^+, \lambda\_.\mathbf{e'}^+)$
$\mathbf{e.1}$	$\rightsquigarrow$	$(\mathit{fst}\ \mathbf{e}^+)\ ()$
$\mathbf{e.2}$	$\rightsquigarrow$	$(\mathit{snd}\ \mathbf{e}^+)\ ()$
$(\mathbf{e}, \mathbf{e'})$	$\rightsquigarrow$	$(\mathbf{e}^+, \mathbf{e'}^+)$
$\mathbf{let\ (a, a') = e\ in\ e'}$	$\rightsquigarrow$	$\mathit{let}\ x_{\text{fresh}} = \mathbf{e}^+\ \mathit{in\ let}\ \underline{a} = \mathit{fst}\ x_{\text{fresh}}\ \mathit{in\ let}\ \underline{a'} = \mathit{snd}\ x_{\text{fresh}}\ \mathit{in}\ \mathbf{e'}^+$
$(\mathbf{e} : \tau_1 \to \tau_2)\ (\mathbf{e} : \tau_1)$	$\rightsquigarrow$	$C_{\tau_2 \mapsto \tau_2}(\mathbf{e}^+\ C_{\tau_1 \mapsto \tau_1}(\mathbf{e}^+))$

Fig. 4. Compilers from MiniML and Affi to *Target*

Simplified Foreign Call

$$\frac{\cdot \vdash \mathbf{e}_1 : \tau_1 \multimap \tau_2 \qquad \Delta; \Gamma \vdash \mathbf{e}_2 : \tau_1 \qquad \_ : \tau_1 \sim \tau_1 \qquad \_ : \tau_2 \sim \tau_2}{\Delta; \Gamma \vdash \mathbf{e}_1\ \mathbf{e}_2 : \tau_2}$$

In particular, we use a symmetric convertibility judgment, which should be read as "the type τ is convertible to τ". As we described in the intro, the implementor would have to define conversions, and show that those conversions correctly translated target code that behaved as one type to code that behaved as the other. Note that if soundness was the only goal, a simpler (unary) model would be possible than the (binary) one we describe in §6.

In this paper, however, we define a more informative convertibility relation that allows us to be more precise about the relationship between the two convertible types. In particular, what we would like to capture with $\tau_A \leq \tau_B$ is that τ_A is semantically more precise than τ_B: i.e., converting from τ_A to τ_B should not introduce a runtime failure but converting from τ_B to τ_A may (in §7

$$\frac{}{C_{\mathtt{unit}\mapsto\mathbf{unit}}, C_{\mathtt{unit}\mapsto\mathtt{unit}} : \mathtt{unit} \leq \mathbf{unit}} \qquad \frac{}{C_{\mathbf{unit}\mapsto\mathtt{unit}}, C_{\mathtt{unit}\mapsto\mathbf{unit}} : \mathbf{unit} \leq \mathtt{unit}}$$

$$\frac{C_{\tau_1\mapsto\tau_1}, C_{\tau_1\mapsto\tau_1} : \tau_1 \leq \tau_1 \qquad C_{\tau_2\mapsto\tau_2}, C_{\tau_2\mapsto\tau_2} : \tau_2 \leq \tau_2}{C_{\tau_1\otimes\tau_2\mapsto\tau_1\times\tau_2}, C_{\tau_1\times\tau_2\mapsto\tau_1\otimes\tau_2} : \tau_1 \otimes \tau_2 \leq \tau_1 \times \tau_2}$$

$$\frac{C_{\tau_1\mapsto\tau_1}, C_{\tau_1\mapsto\tau_1} : \tau_1 \leq \tau_1 \qquad C_{\tau_2\mapsto\tau_2}, C_{\tau_2\mapsto\tau_2} : \tau_2 \leq \tau_2}{C_{\tau_1\otimes\tau_2\mapsto\tau_1\times\tau_2}, C_{\tau_1\times\tau_2\mapsto\tau_1\otimes\tau_2} : \tau_1 \otimes \tau_2 \leq \tau_1 \times \tau_2}$$

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

$$\begin{array}{lcl}
C_{\mathtt{unit}\mapsto\mathbf{unit}}(e) & \triangleq & e \\
C_{\mathbf{unit}\mapsto\mathtt{unit}}(e) & \triangleq & e \\
C_{\tau_1\otimes\tau_2\mapsto\tau_1\times\tau_2}(e) & \triangleq & \mathit{let}\ x = e\ \mathit{in}\ (C_{\tau_1\mapsto\tau_1}(\mathit{fst}\ x), C_{\tau_1\mapsto\tau_1}(\mathit{snd}\ x)) \\
C_{\tau_1\times\tau_2\mapsto\tau_1\otimes\tau_2}(e) & \triangleq & \mathit{let}\ x = e\ \mathit{in}\ (C_{\tau_1\mapsto\tau_1}(\mathit{fst}\ x), C_{\tau_1\mapsto\tau_1}(\mathit{snd}\ x)) \\
C_{\tau_1\multimap\tau_2\mapsto(\mathtt{unit}\to\tau_1)\to\tau_2}(e) & \triangleq & \mathit{let}\ x\ =\ e\ \mathit{in}\ \lambda x_{\mathit{thunk}}.C_{\tau_2\mapsto\tau_2}(x\ (C_{\tau_1\mapsto\tau_1}(x_{\mathit{thunk}}\ ()))) \\
C_{(\mathtt{unit}\to\tau_1)\to\tau_2\mapsto\tau_1\multimap\tau_2}(e) & \triangleq & \mathit{let}\ x\ =\ e\ \mathit{in}\ \lambda \underline{x}.\mathit{let}\ x_{\mathit{check}}\ =\ \mathit{ref}\ \mathit{inl}\ ()\ \mathit{in} \\
 & & \quad \mathit{let}\ x_{\mathit{access}}\ =\ \lambda\ _.\mathit{match}\ !x_{\mathit{check}}\ _\{x_{\mathit{check}} := \mathit{inr}(); C_{\tau_1\mapsto\tau_1}(\underline{x})\}\ _\{\mathit{fail}\}\ \mathit{in} \\
 & & \quad C_{\tau_2\mapsto\tau_2}(x\ x_{\mathit{access}})
\end{array}$$

Fig. 5. Convertibility rules for MiniML and Affi.

we show to exploit this extra information in the convertibility judgment to reason about code migration).

We define convertibility between MiniML and Affi in Figure 5, along with the conversions (glue code) that realizes the translations. In the figure, we only show the five convertibility rules that we have chosen to define and prove sound for this paper.

Our first two rules are symmetric, and capture the fact that unit and unit can be converted between one another: indeed, they translate to exactly the same value in the target, so conversions between the two, $C_{\mathtt{unit}\mapsto\mathbf{unit}}(\cdot)$ and $C_{\mathbf{unit}\mapsto\mathtt{unit}}(\cdot)$, are the identity. These form our base case, though one could easily add cases for integers, etc.

Our next two cases show that strict affine products can be converted to unrestricted products provided the subterms can be converted. So, for example, a pair of units can be converted to a pair of units, but so could a pair of pairs. The conversions, $C_{\tau_1\otimes\tau_2\mapsto\tau_1\times\tau_2}$ and $C_{\tau_1\times\tau_2\mapsto\tau_1\otimes\tau_2}$, use conversions for the composite types.

For our last case, we convert between arrow types. In this case, it is only safe to convert from a Affi function to a MiniML function, and it must be to a MiniML function that expects its argument to be a thunk (unit argument function). The conversion wrappers use the well-understood technique described in [Tov and Pucella 2010], where we create a mutable reference (bound to x_{check}, that essentially stores a boolean flag that indicates whether the affine argument passed to the function has already been used once. If the flag is *inr* (), indicating the argument has been used, all subsequent uses lead to *fail*.

In general, many more rules are possible, and there is no canonical set of rules. Moreover, one can add convertibility rules to an FFI over time, proving the new glue code implementations sound with respect to the system. For instance, if we had booleans in Affi we could relate them to unit + unit.

Using the convertibility judgment, we can define typing rules for both foreign calls. Recall from §2 that we write $\tau_1 \gtrless \tau_2$ to mean that we have *either* $\tau_1 \leq \tau_2$ or $\tau_2 \leq \tau_1$. We do this because function calls should be allowed even if they have the possibility of introducing (defined) runtime failures.

$$\frac{\cdot \vdash \mathbf{e}_1 : \tau_1 \multimap \tau_2 \qquad \Delta; \Gamma \vdash \mathbf{e}_2 : \tau_1 \qquad \_ : \tau_1 \gtrless \tau_1 \qquad \_ : \tau_2 \gtrless \tau_2}{\Delta; \Gamma \vdash \mathbf{e}_1\ \mathbf{e}_2 : \tau_2}$$

And similarly:

$$\frac{\cdot \vdash \mathbf{e}_1 : \tau_1 \to \tau_2 \qquad \Gamma; \Omega \vdash \mathbf{e}_2 : \tau_1 \qquad \_ : \tau_1 \gtrless \tau_1 \qquad \_ : \tau_2 \gtrless \tau_2}{\Gamma; \Omega \vdash \mathbf{e}_1\ \mathbf{e}_2 : \tau_2}$$

6 SEMANTIC TYPE SOUNDNESS

To prove type soundness for both of our source languages (without the foreign calls), we start (§6.1) by defining a logical relation indexed by source-language types and inhabited by target-language terms that semantically behave as prescribed by the source type. Our logical relation captures the notion of soundness that we desire—i.e., that a term in the relation only errors in well-defined ways. To prove type soundness of the source languages (§6.2), we show that the translation (compilation) of every well-typed source term is in this relation. This is the well-understood technique of semantic type soundness [Ahmed et al. 2010; Ahmed 2004; Appel and McAllester 2001]. We then prove soundness of our conversions (§6.3), and finally use that to show the semantic type soundness of our foreign call typing rule (§6.4).

Note that for the purpose of proving semantic type soundness (which is the stated goal of this section), it would suffice to give a *unary* logical relation, i.e., to give an interpretation of source-language types as *sets* of target-language terms. Nonetheless, we instead define a binary logical relation that interprets that source-language types as *relations* of target-language terms. We do this so we can use the same logical relation to reason about code migration in §7.

6.1 Logical Relation

While most logical relations in the literature are indexed by types of a given language and inhabited by terms of the *same* language, ours is a realizability model, which means that our logical relation is indexed by source-language types and inhabited by target terms that *realize* or implement that type. These target terms that realize a source type don't necessarily have to be in the image of the compiler for the source language in question. But certainly, assuming a correct compiler, we would want the compilation of any source terms of type τ to belong to the logical relation at that type. Examples of realizability models include: [].

Following the basic setup of most step-indexed Kripke logical relations, we give a value interpretation of source types $\mathcal{V}[\![\tau]\!]$ and $\mathcal{V}[\![\tau]\!]$, roughly as sets of a world

Another aspect of our relation that is somewhat non-standard, though by no means novel, is that we pair open terms with environments of bindings—a slight variation of this idea is used by Ahmed et al. [2007] and Krishnaswami et al. [2015] in their logical relations for substructural languages. This is important so we can capture the semantics of affine types, where the standard approach of using closing substitutions first and then defining the relation in terms of closed terms does not work. We define the relation for both source languages at once, because they can be mixed: the sets of terms are uniform. First, we define a set of supporting definitions, relating to worlds and step indexing. This is all standard for a logical relation for state, albeit a simplified one.

$$
\begin{array}{l}
\mathit{RelAtom}_n = \{(\mathbb{W}, (\gamma_1, e_1), (\gamma_2, e_2)) \mid \mathbb{W} \in \mathit{World}_n\} \\
\mathit{World}_n = \{(k, \Psi_k) \mid k < n \wedge \Psi_k \in \mathit{HeapTy}_k\} \\
\mathit{HeapTy}_n = \{(\ell_1, \ell_2) \mapsto \mathit{RelAtom}_n\} \cup \{(\ell_1, \bullet) \mapsto \boxdot\} \cup \{(\bullet, \ell_2) \mapsto \boxdot\} \\
\mathit{RelAtom} = \bigcup_n \mathit{RelAtom}_n \\
\mathit{World} = \bigcup_n \mathit{World}_n \\
\mathit{Rel} = \{R \in 2^{\mathit{RelAtom}}\}
\end{array}
$$

$$
\begin{array}{l}
\lfloor R \rfloor_k = \{(\lfloor \mathbb{W} \rfloor_k, (\gamma_1, e_1), (\gamma_2, e_2)) \mid (\mathbb{W}, (\gamma_1, e_1), (\gamma_2, e_2)) \in R\} \\
\lfloor \Psi \rfloor_k = \{(\ell_1, \ell_2) \mapsto \lfloor R \rfloor_k \mid (\ell_1, \ell_2) \mapsto R \in \Psi\} \cup \{(\ell_1, \bullet) \mapsto \boxdot \in \Psi\} \cup \{(\bullet, \ell_2) \mapsto \boxdot \in \Psi\} \\
\triangleright \mathbb{W} = \lfloor \mathbb{W} \rfloor_{\mathbb{W}.k} \\
\mathbb{W}_1 \sqsubseteq \mathbb{W}_2 \triangleq \mathbb{W}_1.k \geq \mathbb{W}_2.k \wedge \ \forall d \in \mathrm{dom}(\mathbb{W}_1.\Psi). \lfloor \mathbb{W}_1.\Psi(d) \rfloor_{\mathbb{W}_2.k+1} = \mathbb{W}_2.\Psi(d)
\end{array}
$$

$$
\begin{array}{lll}
(\sigma_1, \sigma_2) : \mathbb{W} \triangleq & (\forall (\ell_1, \ell_2) \in \mathrm{dom}(\mathbb{W}.\Psi).\ (\mathbb{W}, (\cdot, \sigma_1(\ell_1)), (\cdot, \sigma_2(\ell_2))) \in \mathbb{W}.\Psi(\ell_1, \ell_2)) \wedge \\
& (\forall (\ell_1, \bullet) \in \mathrm{dom}(\mathbb{W}.\Psi).\ \exists v. \ell_1 \mapsto v \in \sigma_1 \wedge \\
& (\forall (\bullet, \ell_2) \in \mathrm{dom}(\mathbb{W}.\Psi).\ \exists v. \ell_2 \mapsto v \in \sigma_2
\end{array}
$$

Fig. 6. Logical Relation: Supporting definitions.

Our worlds $\mathbb{W}$ are composed of step indexes and heap typing Ψ, where the latter either map bijected locations or locations that do not have a matching location in the other program. This is simpler than state-of-the-art logical relations for state, (e.g., [Ahmed et al. 2009; Dreyer et al. 2010]), but sufficient for our motivation: it essentially says in that case, we make no claims about what is stored at these private locations, permitting, among other things, strong updates. The case where locations are in a bijection, which is the first possibility, we map them to a relation, which has the shape that our value and expression relations (defined below) have. That is, it is a triple of a world $\mathbb{W}$ and two terms, each made up of a local environment γ and an open term e that may rely on bindings in that environment. We also define *Rel*, which is the set of arbitrary relations from *RelAtom*. This will be used for polymorphism.

We also define standard restrictions (written $\lfloor \cdot \rfloor_k$) based on the step indexes, over relations and heap typings, for completeness this is an identity on the placeholder $\boxdot$ that indicates no information in the heap typing. As is standard, we define a $\triangleright$ (later) modality defined as restricting the index to the current one, which forces the worlds "forward" one step (as it cuts out everything with the current step index). On a world $\mathbb{W}$, $\lfloor \cdot \rfloor_k$, and $\triangleright$ naturally extends to other definitions with step indexes.

Using those definitions, we can define a well-formed world, $\vdash \mathbb{W}$, which is one in which the heap typings do not refer to step indexes past that specified in the world. Unless otherwise specified, we will always assume that when we write "consider a world $\mathbb{W}$", we mean "consider a world $\mathbb{W}$ such that $\vdash \mathbb{W}$". We next define world extension, $\mathbb{W}_1 \sqsubseteq \mathbb{W}_2$ (between well-formed worlds), as one in which the step index can shrink, and existing locations map to the same thing in the future world (including keeping non-bijected locations non-bijected), modulo loss of information due to decreasing step index.

Finally, we define heaps, and how a pair of heaps satisfy a world $(\sigma_1, \sigma_2) : \mathbb{W}$: essentially, for the case of bijected locations, the heaps must point to (closed) related values, otherwise, the location must just point to *a* value, but we make no restrictions on what the value should be.

Expression and Value Relation. We define our expression and value relation below.

While not novel, one unusual aspect of our expression relation is that it packages environments with terms, and thus the terms themselves are still open (though only with respect to this local environment). This means that our interpretation of environments is atypical, as while the overall

$$\Theta(x,\gamma) = \gamma[x] \qquad \underline{\Theta}(\underline{x},\gamma) = \gamma[x], \gamma \setminus x \qquad \gamma_1, \underline{x}{\mapsto}v \uplus \gamma_2, \underline{y}{\mapsto}v' \triangleq \gamma_1 \uplus \gamma_2, \underline{x}{\mapsto}v, \underline{y}{\mapsto}v'$$
$$\Theta(v,\gamma) = v \qquad \underline{\Theta}(x,\gamma) = \gamma[x], \gamma \qquad \gamma_1, x{\mapsto}v \uplus \gamma_2, x{\mapsto}v \triangleq \gamma_1 \uplus \gamma_2, x{\mapsto}v$$
$$\underline{\Theta}(v,\gamma) = v, \gamma$$

$$\begin{array}{lcl}
\mathcal{E}[\![\tau]\!]_\rho & = & \{(\mathbb{W},(\gamma_1,e_1),(\gamma_2,e_2)) \mid \forall(\sigma_1,\sigma_2):\mathbb{W},\ j<k.\ \langle\sigma_1,\gamma_1,e_1\rangle \xrightarrow{j} \langle\sigma_1',\gamma_1',v\rangle \nrightarrow \\
& & \quad \implies v_1 = \mathit{fail} \vee (\exists \mathbb{W}'\ \sigma_2'\ \gamma_2'.\mathbb{W} \sqsubseteq \mathbb{W}' \wedge (\sigma_1',\sigma_2'):\mathbb{W}' \wedge \langle\sigma_2,\gamma_2,e_2\rangle \xrightarrow{*} \langle\sigma_2',\gamma_2',v_2\rangle \\
& & \quad\quad \wedge (\mathbb{W}',(\gamma_1',v_1),(\gamma_2',v_2)) \in \mathcal{V}[\![\tau]\!]_\rho)\} \\
\\
\mathcal{V}[\![\texttt{unit}]\!]_\rho & = & \{(\mathbb{W},(\gamma_1,v_1),(\gamma_2,v_2)) \mid \Theta(v_i,\gamma_i) = ()\} \\
\mathcal{V}[\![\texttt{int}]\!]_\rho & = & \{(\mathbb{W},(\gamma_1,v_1),(\gamma_2,v_2)) \mid \Theta(v_i,\gamma_i) \in \{0,1,\ldots,2^{64}\}\} \\
\mathcal{V}[\![\tau_1 \times \tau_2]\!]_\rho & = & \{(\mathbb{W},(\gamma_1,v_1),(\gamma_2,v_2)) \mid \Theta(v_i,\gamma_i) = (v_{ia},v_{ib}) \wedge \gamma_i = \gamma_{ia} \uplus \gamma_{ib} \wedge \\
& & \quad (\mathbb{W},(\gamma_{1a},v_{1a}),(\gamma_{2a},v_{2a})) \in \mathcal{V}[\![\tau_1]\!]_\rho \wedge (\mathbb{W},(\gamma_{1b},v_{1b}),(\gamma_{2b},v_{2b})) \in \mathcal{V}[\![\tau_2]\!]_\rho\} \\
\mathcal{V}[\![\tau_1 + \tau_2]\!]_\rho & = & \{(\mathbb{W},(\gamma_1,v_1),(\gamma_1,v_2)) \mid \Theta(v_i,\gamma_i) = \mathit{inl}\ v_i \wedge (\mathbb{W},v_1,v_2) \in \mathcal{V}[\![\tau_1]\!]_\rho\} \\
& & \cup \{(\mathbb{W},(\gamma_1,v_1),(\gamma_1,v_2)) \mid \Theta(v_i,\gamma_i) = \mathit{inr}\ v_i \wedge (\mathbb{W},v_1,v_2) \in \mathcal{V}[\![\tau_1]\!]_\rho\} \\
\mathcal{V}[\![\tau_1 \to \tau_2]\!]_\rho & = & \{(\mathbb{W},(\gamma_1,f_1),(\gamma_2,f_2)) \mid \Theta(f_i,\gamma_i) = \lambda x.e_i \wedge \forall v_1\ v_2\ \mathbb{W}' j.j<k \wedge \mathbb{W}'.k = j \wedge \mathbb{W} \sqsubseteq \mathbb{W}' \\
& & \quad \wedge (\mathbb{W}',(\gamma_1',v_1),(\gamma_2',v_2)) \in \mathcal{V}[\![\tau_1]\!]_\rho \implies \\
& & \quad\quad (\mathbb{W}',(\gamma_1 \uplus \gamma_1'\{x{\mapsto}v_1\},e_1),(\gamma_2 \uplus \gamma_2'\{x{\mapsto}v_2\},e_2)) \in \mathcal{E}[\![\tau_2]\!]_\rho\} \\
\mathcal{V}[\![\texttt{ref}\ \tau]\!]_\rho & = & \{(\mathbb{W},(\gamma_1,v_1),(\gamma_2,v_2)) \mid \Theta(v_i,\gamma_i) = \ell_i \wedge \triangleright\mathbb{W}.\Psi(\ell_1,\ell_2) = \triangleright\mathcal{V}[\![\tau]\!]_\rho\} \\
\mathcal{V}[\![\forall\alpha.\tau]\!]_\rho & = & \{(\mathbb{W},(\gamma_1,v_1),(\gamma_2,v_2)) \mid \Theta(v_i,\gamma_i) = \lambda\alpha.e_i \wedge \forall R \in \mathit{Rel}. \\
& & \quad (\triangleright\mathbb{W},(\gamma_1,e_1),(\gamma_2,e_2)) \in \mathcal{E}[\![\tau]\!]_{\rho[\alpha\mapsto R]}\} \\
\mathcal{V}[\![\alpha]\!]_\rho & = & \rho(\alpha) \\
\\
\mathcal{V}[\![\textbf{unit}]\!]. & = & \{(\mathbb{W},(\gamma_1,v_1),(\gamma_2,v_2)) \mid \underline{\Theta}(v_i,\gamma_i) = (),_\} \\
\mathcal{V}[\![\textbf{int}]\!]. & = & \{(\mathbb{W},(\gamma_1,v_1),(\gamma_2,v_2)) \mid \underline{\Theta}(v_i,\gamma_i) = n,_ \wedge n \in \{0,1,\ldots,2^{64}\}\} \\
\mathcal{V}[\![\tau_1 \multimap \tau_2]\!]. & = & \{(\mathbb{W},(\gamma_1,v_1),(\gamma_2,v_2)) \mid \underline{\Theta}(v_i,\gamma_i) = \lambda\underline{x}.\{e_i\},\gamma_i^\dagger \wedge \forall v_1\ \gamma_1'\ v_2\ \gamma_2'\ \mathbb{W}' j.j<k \wedge \\
& & \quad \mathbb{W}'.k = j \wedge \mathbb{W} \sqsubseteq \mathbb{W}' \wedge (\mathbb{W}',(\gamma_1',v_1),(\gamma_2',v_2)) \in \mathcal{V}[\![\tau_1]\!]. \implies \\
& & \quad\quad (\mathbb{W}',(\gamma_1^\dagger \uplus \gamma_1'\{x{\mapsto}\underline{v_1}\},e_1),(\gamma_2^\dagger \uplus \gamma_2'\{x{\mapsto}\underline{v_2}\},e_2)) \in \mathcal{E}[\![\tau_2]\!].\} \\
\mathcal{V}[\![!\tau]\!]. & = & \{(\mathbb{W},(\gamma_1,v_1),(\gamma_2,v_2)) \mid \mathrm{unrestricted}(\gamma_i) \wedge (\mathbb{W},(\gamma_1,v_1),(\gamma_2,v_2)) \in \mathcal{V}[\![\tau]\!].\} \\
\mathcal{V}[\![\tau_1 \otimes \tau_2]\!]. & = & \{(\mathbb{W},(\gamma_1,v_1),(\gamma_2,v_2)) \mid \underline{\Theta}(v_i,\gamma_i) = (v_{ia},v_{ib}),\gamma_{ia} \uplus \gamma_{ib} \wedge \\
& & \quad (\mathbb{W},(\gamma_{1a},v_{1a}),(\gamma_{2a},v_{2a})) \in \mathcal{V}[\![\tau_1]\!]. \wedge (\mathbb{W},(\gamma_{1b},v_{1b}),(\gamma_{2b},v_{2b})) \in \mathcal{V}[\![\tau_2]\!].\} \\
\mathcal{V}[\![\tau_1 \& \tau_2]\!]. & = & \{(\mathbb{W},(\gamma_1,v_1),(\gamma_2,v_2)) \mid \underline{\Theta}(v_i,\gamma_i) = (\lambda_.e_{ia},\lambda_.e_{ib}),\gamma_i^\dagger \wedge \\
& & \quad (\mathbb{W},(\gamma_1^\dagger,e_{1a}),(\gamma_2^\dagger,e_{2a})) \in \mathcal{E}[\![\tau_1]\!]. \wedge (\mathbb{W},(\gamma_1^\dagger,e_{1b}),(\gamma_2^\dagger,e_{2b})) \in \mathcal{E}[\![\tau_2]\!].\}
\end{array}$$

Fig. 7. Logical Relation: Value and Expression Relation.

shape of an environment (determined by its type) indeed describes "closed" terms, those closed "terms" are pairs of environments and terms that depend on those environments. Note also that since our target environments distinguish between affine (single-use) bindings and unrestricted ones, we carry that to the environments we take in to this interpretation, which means we need to (minimally) transform our source environments (marking which bindings are affine) before interpreting them:

Our value relation is indexed by the source types of both MiniML and Affi. Note, however, that what inhabits the relation is just the target: these source types are purely logical constructs. We define the relation for both source languages at once, and then describe it below.

Note the following metafunctions for looking up in our stack (local environment) definition, and a modified disjoint union on environments that matches on unrestricted bindings (thus duplicating them) and is disjoint on affine bindings (thus splitting them):

$$
\begin{array}{lcl}
\mathcal{G}[\![\cdot]\!]_\rho & = & \{(\mathbb{W}, \cdot, \cdot)\} \\
\mathcal{G}[\![\Gamma, x : \tau]\!]_\rho & = & \{(\mathbb{W}, \gamma_1; \gamma_1'; x \mapsto v_1, \gamma_2; \gamma_2'; x \mapsto v_2) \mid (\mathbb{W}, (\gamma_1', v_1), (\gamma_2', v_2)) \in \mathcal{V}[\![\tau]\!]_\wedge (\mathbb{W}, \gamma_1, \gamma_2) \in \mathcal{G}[\![\Gamma]\!]_\rho \\
\mathcal{G}[\![\Gamma, \underline{x} : \tau]\!]_\rho & = & \{(\mathbb{W}, \gamma_1; \gamma_1'; \underline{x} \mapsto v_1, \gamma_2; \gamma_2'; \underline{x} \mapsto v_2) \mid (\mathbb{W}, (\gamma_1', v_1), (\gamma_2', v_2)) \in \mathcal{V}[\![\tau]\!]_\wedge (\mathbb{W}, \gamma_1, \gamma_2) \in \mathcal{G}[\![\Gamma]\!]_\rho \\
\\
\mathcal{D}[\![\cdot]\!] & = & \{\cdot\} \\
\mathcal{D}[\![\Delta, \alpha]\!] & = & \{\rho[\alpha \mapsto \mathrm{R}] \mid R \in Rel \wedge \rho \in \mathcal{D}[\![\Delta]\!]\}
\end{array}
$$

$$
\begin{array}{l}
[\![\Delta; \Gamma \vdash \mathtt{e}_1 \leq \mathtt{e}_2 : \tau]\!] \equiv \forall \mathbb{W}. \forall \gamma_1\ \gamma_2\ \rho.\ \rho \in \mathcal{D}[\![\Delta]\!] \wedge (\mathbb{W}, \gamma_1, \gamma_2) \in \mathcal{G}[\![\Gamma^+]\!]_\rho \implies \\
\qquad\qquad (\mathbb{W}, (\gamma_1, \mathtt{e}_1{}^+), (\gamma_2, \mathtt{e}_2{}^+)) \in \mathcal{E}[\![\tau]\!]_\rho \\
[\![\Gamma; \Omega \vdash \mathtt{e}_1 \leq \mathtt{e}_2 : \tau]\!] \equiv \forall \mathbb{W}. \forall \gamma_1 \gamma_2. (\mathbb{W}, \gamma_1, \gamma_2) \in \mathcal{G}[\![\Gamma^+, \Omega^+]\!].\ \implies \\
\qquad\qquad (\mathbb{W}, (\gamma_1, \mathtt{e}_1{}^+), (\gamma_2, \mathtt{e}_2{}^+)) \in \mathcal{E}[\![\tau]\!].
\end{array}
$$

Fig. 8. Logical Relation: Open Terms.

Our expression relation $\mathcal{E}[\![\tau]\!]_\rho$ is defined as a standard error approximation relation: two terms are related if the one on the left errors more than the one on the right. That is, if the term on the left does not error (reduce to *fail*) then the term on the right must reduce to a related value with a related heap.

Our value relation $\mathcal{V}[\![\tau]\!]_\rho$ is based on the compiler, and $()$, which is the only value of type `unit`, compiles to $()$. Since our relation defined over open terms, combined with environments that close them, we use the metafunction $\Theta(\cdot, \cdot)$ defined above to show that the values are either $()$ or variables that are bound to that in the environment. This is the same for integers.

The relation on pairs is similar to how logical relations for pairs are usually defined, with a pair of values where the values should be in the relation for the component types. We also have to handle the environments: even though $\tau_1 \times \tau_2$ is an unrestricted pair, it is possible that there are affine bindings in the environment so we still split the environments with $\uplus$: since this separates affine variables into the two sides and includes unrestricted bindings on both sides, if there are only unrestricted bindings, both environments will be the same.

The relation on sums is defined by a union of *inl* and *inr* applied to values in the relation at the appropriate types.

The relation on arrow types is relatively standard: the only difference is that we combine the environment used for the argument with the environment for the function when checking that the body is in the relation.

The relation on references is standard, relying on the heap typing to ensure the invariant, and using the later ($\triangleright$) modality to ensure it is well-defined.

Polymorphic functions are defined as is standard, choosing a relation from *Rel* and extending the relational substitution, which is then used for the interpretation of polymorphic variables.

Our relation for types from **Affi** is similar, though uses the metafunction $\underline{\Theta}(\cdot, \cdot)$, which produces not only a value but the environment, as it will remove affine variables.

The interpretation of the $!\tau$ type requires that there be no affine variables, which we can check directly using the environment, as our environments track that; otherwise we just pass it through to the interpretation of the value relation for type τ.

The interpretation of strict affine products is very similar to unrestricted products: the primary difference will show up in the environments and the behavior of $\uplus$.

Our interpretation of lazy products is based on our compilation, which translates them to pairs of thunks. Only one component cant be used, so we don't split the environment.

Interpreting Open Terms. We define interpretations for environments (both term and type), and use that to define interpretations for an open type judgment for each source language.

The term environment interpretation $\mathcal{G}[\![\cdot]\!]_\rho$ captures both the "public" bindings (described in Γ) and the private bindings that are used for the values that those public bindings point to.

Our interpretation of type environments $\mathcal{D}[\![\cdot]\!]$ is more typical. Note that we write this in MiniML colors because our other language does not have polymorphism, so the only type environment we will be interpreting will be a MiniML one.

Finally we define a interpretation of a type refinement on an open term, once for each source typing judgment (as these judgments have different shapes). Note that Γ^+ and Γ^+ are both identity transformations, whereas Ω^+ adds the affine identifying underlines, as Ω is the affine value environment from **Affi**. We will use these shorthands frequently when stating compatibility rules about our type systems.

6.2 Soundness of Source Languages

We prove the two source languages, without the foreign calls, sound via the standard method. First, for every typing rules in both languages, we prove a semantic analog to the static rule. For example, from the typing rule:

$$\frac{\Gamma; \Omega_1 \vdash \mathbf{e}_1 : \tau_1 \multimap \tau_2 \qquad \Gamma; \Omega_2 \vdash \mathbf{e}_2 : \tau_1}{\Gamma; \Omega_1 \uplus \Omega_2 \vdash \mathbf{e}_1\ \mathbf{e}_2 : \tau_2}$$

We prove the compatibility lemma:

$$[\![\Gamma; \Omega_1 \vdash \mathbf{e}_1 \leq \mathbf{e}_1' : \tau_1 \multimap \tau_2]\!] \wedge [\![\Gamma; \Omega_2 \vdash \mathbf{e}_2 \leq \mathbf{e}_2' : \tau_1]\!] \implies [\![\Gamma; \Omega_1 \uplus \Omega_2 \vdash \mathbf{e}_1\ \mathbf{e}_2 \leq \mathbf{e}_1'\ \mathbf{e}_2' : \tau_2]\!]$$

Where, as described above, the notation is defined as:

$$[\![\Gamma; \Omega \vdash \mathbf{e}_1 \leq \mathbf{e}_2 : \tau]\!] \equiv \forall \mathbb{W}. \forall \gamma_1 \gamma_2. (\mathbb{W}, \gamma_1, \gamma_2) \in \mathcal{G}[\![\Gamma^+, \Omega^+]\!]. \implies (\mathbb{W}, (\gamma_1, \mathbf{e}_1{}^+), (\gamma_2, \mathbf{e}_2{}^+)) \in \mathcal{E}[\![\tau]\!].$$

The full proofs, which follow standard techniques, are in our supplementary materials. None are particularly challenging, but to prove them requires that our relations encode sufficient information about our type invariants: for example, proving the **let!** compatibility lemma requires the affine bindings be tracked in the target.

Once we have proved all of the individual compatibility lemmas, we can prove the fundamental property of the logical relation, which says that a term that is well-typed is related to itself:

Theorem 6.1 (Fundamental Property).
If $\Delta; \Gamma \vdash \mathsf{e} : \tau$ *then* $[\![\Delta; \Gamma \vdash \mathsf{e} : \tau]\!]$ *and if* $\Gamma; \Omega \vdash \mathbf{e} : \tau$ *then* $[\![\Gamma; \Omega \vdash \mathbf{e} : \tau]\!]$.

Proof. By induction on typing derivation, relying on all of the compatibility lemmas, which exist for every typing rule in both source languages. Note that this depends on a compatibility lemma for the foreign call rule, which the key aspect of our system and the focus of the rest of this section. □

6.3 Soundness of Conversions

Our foreign call typing rules depend on the convertibility relation, so in order to prove semantic soundness of the foreign call rules, first we have to show an analogous result for the convertibility judgment. In particular, for each of our convertibility rules, we prove that the conversions respect our logical relation as shown in the following theorem.

Theorem 6.2 (Convertibility Soundness). *If* $\tau_A \leq \tau_B$ *then*
$\forall\ (\mathbb{W}, (\gamma_1, e_1), (\gamma_2, e_2)) \in \mathcal{E}[\![\tau_A]\!]. \implies (\mathbb{W}, C_{\tau_A \mapsto \tau_B}(e_1), C_{\tau_A \mapsto \tau_B}(e_2)) \in \mathcal{E}[\![\tau_B]\!]. \wedge$
$\forall\ (\mathbb{W}, (\gamma_1, e_1), (\gamma_2, e_2)) \in \mathcal{E}[\![\tau_B]\!]. \implies (\mathbb{W}, C_{\tau_B \mapsto \tau_A}(e_1), C_{\tau_B \mapsto \tau_A}(e_2)) \in \mathcal{E}[\![\tau_A]\!]..$

We prove the same property of both conversions: while the conversion going *up* (from τ_A to τ_B) cannot introduce more possibility of failures and going *down* (from τ_B to τ_A) may introduce more failures. Note that the logical relation allows the possibility of failure and thus the soundness results does not distinguish between the two directions.

The proofs are straightforward and included in our supplementary materials.

In order to prove these and other theorems about our conversions, we require two key properties to hold about them:

Property 1 (Conversions Are Evaluation Context (Strictness)).

$$\langle \sigma, \gamma, C(e)\rangle \xrightarrow{*} \langle \sigma', \gamma', v\rangle \iff (\langle \sigma, \gamma, \mathit{let}\ x_{fresh} = e\ \mathit{in}\ C(x_{fresh})\rangle) \xrightarrow{*} (\sigma', \gamma'[x_{fresh}\mapsto\_], v)$$

This property says that conversions behave like evaluation contexts – i.e., they evaluate their arguments (they do not ignore them), which is also sometimes called strictness. It also indicates they evaluate them first, but this is less critical given the second property we require:

Property 2 (Conversion Purity).

$$\langle \sigma, \gamma, (\lambda x.e)C(v)\rangle \xrightarrow{*} \langle \sigma', \gamma', v\rangle \iff (\langle \sigma, \gamma, e[C(v)/x]\rangle) \xrightarrow{*} (\sigma', \gamma' \setminus x, v)$$

There are various ways to characterize that the conversions themselves must be pure, but the one that we choose happens to be convenient for our proofs: that converted values can be inlined into functions, rather than first being evaluated. Note that the purity requirement does not mean that conversions cannot return impure suspended computations, as we did with our conversion for arrows, just that the conversion itself is pure.

6.4 Soundness of Foreign Call

Once we have proved Theorem 6.2, we can prove the last two compatibility lemmas: for the foreign calls from MiniML to Affi and the reverse.

We have to prove the following semantic compatibility lemmas:

$$[\![\cdot;\cdot \vdash \mathbf{e}_1 \le \mathbf{e}_1' : \tau_1 \multimap \tau_2]\!] \wedge [\![\Delta;\Gamma \vdash \mathbf{e}_2 \le \mathbf{e}_2' : \tau_1]\!] \wedge \_ : \tau_1 \gtrless \tau_1 \wedge \_ : \tau_2 \gtrless \tau_2 \implies [\![\Delta;\Gamma \vdash \mathbf{e}_1\ \mathbf{e}_2 \le \mathbf{e}_1'\ \mathbf{e}_2' : \tau_2]\!]$$

And:

$$[\![\cdot;\cdot \vdash \mathbf{e}_1 \le \mathbf{e}_1' : \tau_1 \to \tau_2]\!] \wedge [\![\Gamma;\Omega \vdash \mathbf{e}_2 \le \mathbf{e}_2' : \tau_1]\!] \wedge \_ : \tau_1 \gtrless \tau_1 \wedge \_ : \tau_2 \gtrless \tau_2 \implies [\![\Gamma;\Gamma \vdash \mathbf{e}_1\ \mathbf{e}_2 \le \mathbf{e}_1'\ \mathbf{e}_2' : \tau_2]\!]$$

We need to show:

$$\forall \mathbb{W}.\forall \gamma_1\gamma_2.(\mathbb{W},\gamma_1,\gamma_2) \in \mathcal{G}[\![\Gamma^+,\Omega^+]\!]. \implies (\mathbb{W}, (\gamma_1, C_{\tau_2\mapsto\tau_2}(\mathbf{e_1}^+\ C_{\tau_1\mapsto\tau_1}(\mathbf{e_2}^+))), (\gamma_2, C_{\tau_2\mapsto\tau_2}(\mathbf{e_1'}^+\ C_{\tau_1\mapsto\tau_1}(\mathbf{e_2'}^+))) \in \mathcal{E}[\![\tau_2]\!].$$

The full proofs are included in our supplementary materials, but we provide a sketch here. This essentially follows by combining the induction hypotheses with the soundness lemmas for the conversion judgments: we know from our second hypothesis that $\mathbf{e}_2$ and $\mathbf{e}_2'$ are related at type $\mathcal{E}[\![\tau_1]\!].$, so the soundness of conversion (Theorem 6.2) means the converted terms are related at type $\mathcal{E}[\![\tau_1]\!].$. That means that our first induction hypotheses can be applied to them, meaning the application is in $\mathcal{E}[\![\tau_2]\!].$, and soundness of conversion for the outer conversion means the overall terms are related at $\mathcal{E}[\![\tau_2]\!].$, as needed.

7 LANGUAGE MIGRATION

Suppose we want to migrate code written one language to code written in another. In our setting, this would involve replacing a native function `f` with a foreign function `f` that we hope behaves the same. Parts of the program that apply this new function will now typecheck using the foreign call typing rule rather than the native application rule.

Consider arbitrary languages A and B, where we wish to replace a function f_A of type τ_A with a function f_B of type τ_B. In order to do this we would need to have a convertibility derivation relating τ_A to τ_B. Suppose that derivation shows that $\tau_A \leq \tau_B$. Then we can use our logical relation to prove that the conversion $C_{\tau_B \to \tau_A}(\cdot)$ applied to the compilation of f_B (f_B^+) error-approximates the compilation of f_A (f_A^+) at the type τ_A. Note that we cannot show that f_A^+ error-approximates $C_{\tau_B \to \tau_A}(f_B^+)$ since we are converting f_B^+ from τ_B to a more precise type τ_A and this may introduce errors.

If we instead were replacing a native function f_B of type τ_B with a foreign function f_A of type τ_A, still assuming that $\tau_A \leq \tau_B$, then we would be able to show that both $C_{\tau_B \to \tau_A}(f_A^+)$ error-approximates f_B^+ and f_B^+ error approximates $C_{\tau_B \to \tau_A}(f_A^+)$ at the type τ_B. This is because we are converting f_A to a less precise type, and that cannot introduce errors.

The fact that should be able prove about error approximation are a consequence of the ep-pair property, which ensures the directionality that we have thus far assumed in our convertibility rules. Note that we have proved the following ep-pair property for the conversions for `MiniML` and **Affi** shown in Figure 5 (see supplementary materials for proofs).

Theorem 7.1 (EP-Pair Property). *If* $\tau_A \leq \tau_B$,
$\forall \mathbb{W}\ e\ \gamma\ e'\ \gamma'\ \rho.\ (\mathbb{W}, (\gamma, e), (\gamma', e')) \in \mathcal{E}[\![\tau_A]\!]_\rho \implies (\mathbb{W}, (\gamma, e), (\gamma', C_{\tau_B \mapsto \tau_A}(C_{\tau_A \mapsto \tau_B}(e')))) \in \mathcal{E}[\![\tau_A]\!]_\rho$.
and
$\forall \mathbb{W}\ e\ \gamma\ e'\ \gamma'\ \rho.\ (\mathbb{W}, (\gamma, e), (\gamma', e')) \in \mathcal{E}[\![\tau_B]\!]_\rho \implies (\mathbb{W}, (\gamma, C_{\tau_A \mapsto \tau_B}(C_{\tau_B \mapsto \tau_A}(e))), (\gamma', e')) \in \mathcal{E}[\![\tau_B]\!]_\rho$.

We prove this for `MiniML` and **Affi** by induction over the convertibility judgment, which mostly follow by unfolding definitions. One interesting detail involves showing that, in the arrow case, when we convert from `MiniML` to **Affi** and then back, the function that guards access to the resource (introduced by the round trip) is an error approximation to the double conversion without the guard. This makes sense, as the only additional behavior that the guard introduces is the possibility of failure, which is exactly what our error approximation relation captures.

8 RELATED WORK

Much of the research on FFIs has focused on tools to either reduce the amount of boilerplate or improve the performance of the resulting code. With some exceptions, we will not discuss those here, focusing instead on work that somehow addresses the issue of reasoning about interoperability and questions of soundness.

Multi-language semantics. Matthews and Findler [2007] studied the question of language interoperability from a source perspective, developing the idea of a syntactic multi-language with boundary terms that mediated between the two languages. Many papers (e.g., [Ahmed and Blume 2011; New et al. 2016; Osera et al. 2012; Patterson et al. 2017; Perconti and Ahmed 2014; Scherer et al. 2018; Tov and Pucella 2010]) have modeled language interoperability via multi-language semantics, whether directly to understand how features interact or in order to prove compiler transformations correct and/or fully abstract.

Barrett et al. [2016] took a slightly different path, directly mixing languages, in their case PHP and Python, and allowing bindings from one to be used in the other. In all of these cases, the approach is based on an idealized version of what the interaction would be: not what any actual

implementation might do, as real implementations interact by translation to common intermediates or targets, inserting appropriate wrappers as needed.

Interop via typed targets. Shao and Trifonov [1998]; Trifonov and Shao [1999] studied interoperability much earlier, and much closer to our context: they explicitly were considering interoperability mediated by translation to a common target. As in our context, they expected the source languages that are each themselves safe. They tackled the problem that one language had access to control effects and the other did not. Their approach, however, is quite different: it relies upon a target language with an effect-based type system that is sufficient to capture both the safety invariants and support interoperation between code that came from each language. This would fit into our framework, but in some sense what we are trying to do is a bit broader, as we are trying to come up with a general approach that both richly typed intermediate languages (which, as a general rule, do not exist outside of research) and untyped (or poorly typed) intermediate languages that rely on runtime checks to enforce invariants can both fit into. While we think typed intermediate languages obviously offer some real benefits, there are also unaddressed problems, foremost of which is that it is quite difficult to design a type system that is sufficiently general to allow (efficient) compilation from all the languages you want to support. We can see this idea somewhat realized in a more practical setting with the TruffleVM project ([Grimmer et al. 2015]), with the expected challenges in terms of what the target can support and as-yet little meta-theory done.

Proving particular FFIs sound. There has been significant work on how to augment existing unsafe FFIs in order to make them safe, primarily by adding type systems, inference, static analysis, etc. For example, Furr and Foster [2005] study interactions between OCaml and C via the C FFI, and specifically, how to ensure safety by doing type inference on the C code as it operates over a structural representation of OCaml types. There is some faint similarity to our work, if you assume the type systems could be made sound and take the two interacting languages to not be OCaml and C but rather those languages augmented with the richer types and inference systems.

Along the same lines, Tan et al. [2006] studied safe interoperability between Java and C via the JNI. They did this by first ensuring safety for C via CCured [Necula et al. 2002], and then extending that with static and dynamic checks to ensure that the invariants of Java pointers and APIs can not be violated in C. This is more directly an instantiation of our framework, once you take the two languages to be Java and the modified C (CCured), which has a richer, safe, type system. They prove a subset of their system correct by extending the CCured proof. Hirzel and Grimm [2007] also built a system for safe interoperability between Java and C, though their system, Jeanie, relies on a novel syntax that embedded both languages and was responsible for analyzing both together before compiling to Java and C with appropriate JNI usage. There has been plenty of other research studying how to make the JNI safer by analyzing C for various properties (e.g., looking for exception behavior in [Li and Tan 2014]).

Rich FFIs. There has been lots of work exploring how to make existing FFIs safer, usually by extending the annotations that are written down so that there is less hand-written (and thus error-prone) code to write. Some of this was done in the context of the Haskell FFI, including work by Chakravarty [1999]; Finne et al. [1998]; Jones et al. [1997]. This is in some sense relevant because much of what they explored was how to preserve type invariants from Haskell, or how to express type invariants via different mechanisms (whether in separate description languages, or extracted from headers, or some combination). However, while the authors were obviously concerned about questions of soundness, it is not clear from these papers whether they proved any soundness properties of the systems in question. Similar work has also been done in other languages, for example, for Standard ML Blume [2001] embedded C types into ML such that ML

programs could safely operate over low-level C representations. This approach fits in well to our semantic framework, as they have ML types that have the same interpretation as corresponding C types (to minimize copying/conversions), realized by minimal wrapper code.

Another approach to having rich FFIs is to co-design both languages, as has been done in the much more recent verification project Everest ([Bhargavan et al. 2017]), where a low-level C-like language Low* has been designed ([Protzenko et al. 2017]) to interoperate with an embedding of a subset of assembly suitable for cryptography ([Fromherz et al. 2019]). By embedding both languages into the verification framework F*, they are able to prove rich properties about the interactions between the two languages.

An abstract framework for unsafe FFIs. Turcotte et al. [2019] advocate a more general approach to type soundness by extending languages with abstract versions of the foreign language in order to prove soundness without building a full multi-language. They demonstrate this by proving a modified type safety proof of Lua and C interacting via the C FFI, modeling the C as essentially code that can do arbitrary unsound behavior. Thus, the result allows them to state that if there is an error, it must have come from the C side. While this approach seems promising in the context of unsound languages whose behavior can be collapsed, it is less clear how it applies to sound languages interacting.

Modeling FFIs via State Machines. Lee et al. [2010] specify the type (and other) constraints that exist in both the JNI and Python/C FFI via state machines and use that to generate runtime checks to enforce these at runtime. While this is practical work and so they do not prove properties about their system, Jinn, there are many similarities between their approach and ours. In particular, the idea that invariants that cannot be expressed via the languages themselves and should instead be checked via inserted code. We would expect that if their approach were applied to safe languages, we would be able to prove that the code that they inserted satisfied semantic interpretations of the respective types.

Semantics foundations of gradual typing. Gradual typing and language interoperability have long been thought to be deeply connected: indeed, that gradual typing is a particularly compelling example of language interoperability is an oft-mentioned statement. In this work, we make this much more concrete, by drawing much of our theoretical foundation from the application of embedding-projection pairs to gradual typing by New and Ahmed [2018], and while some of it is future work that we discuss in §9.4, much of our exploration of language migration was inspired by the static and dynamic gradual guarantees described by Siek et al. [2015].

9 DISCUSSION AND FUTURE WORK

9.1 Other Language Features

In this paper, in order to clearly demonstrate the idea we intentionally kept the languages small, while retaining a non-trivial difference of features: mutable references and polymorphism on the MiniML side, affine bindings on the Affi. However, the technique should extend naturally to different features, different source languages, and different targets (which we will discuss later in this section). For example, modeling a common feature like foreign pointers should be straightforward, as foreign pointers are a native data type that is then convertible to a pointer type in the foreign language.

Further, we chose to model the only direct interaction as foreign function calls, as this is a representative method of interaction, but there is nothing in the technique that prevents us from modeling other direct interactions: for example, invoking methods on foreign objects. What this would require in our framework is adding these elimination forms and then proving that they are

sound in the same way that we showed that the foreign function call rule is. Essentially, features are either converted to native equivalents (in the case of data), which are proved sound via convertibility rules or are eliminated via new operational rules, which are proved sound directly.

In addition to exploring other individual language features, we would like to apply this to more realistic languages, both source and target.

9.2 Target Language

We chose an untyped target language to show that we did not need to encode rich static information in the target, but kept it high-level in order to keep the relations and compilers simple, so as to not distract from the techniques being illustrated. However, as has been shown in prior work on realizability models (e.g., [Benton and Hur 2009; Benton and Tabareau 2009]), it is possible to build logical relations in the same style even when the target language is much lower-level and further from the source. Indeed, the only potential restrictions lie in needing to capture invariants used by the logical relation. In the case study shown in this paper, we needed to be able to be able to distinguish between affine and unrestricted bindings in the target in order to prove compatibility lemmas about the $!\tau$ type. However, there is no reason that the particular method we used, augmenting the operational semantics, is necessary. We could, for example, have instead added a type system to our target to capture this. This possibly could be easier in the setting of languages without substructural features, e.g., an ML-like language interacting with a Java-like language. Finally, it is important to point out that this framework requires that target that is used to build the logical relation is shared between languages, so it will naturally need to be at least as low-level than the lowest-level language you are dealing with.

9.3 Invoking Closed Functions

Our foreign function calls expect closed terms of function type. There are some limitations to this approach: while dependencies between functions can be solved by code duplication (i.e., inlining all of the dependencies), if there is shared mutable data, there is no way that multiple foreign functions can access it. To get around this, we would need to instead provide a notion of imports/exports, and require that the functions only depend on the export environment. It is important for them not to be open, because we don't want to end up with environments of mixed bindings from different languages. This is a problem that has been explored in the context of multilanguage programming (e.g., by [Barrett et al. 2016]), but is quite different from the way that foreign function interfaces are implemented, since they rely on compiling each languages code and then transferring control between those large portions of code, rather than the much finer grained mixing that shows up when you can using a variable defined in another language.

9.4 Term Precision and Migrationality

In §7 we discussed the connection to gradual typing and graduality. While we have the building blocks of type conversions and ep-pairs, we do not have a way of directly stating an analogous property. The problem is that in order to state this we would need to define a term precision judgment, and then show that it satisfied a cross-language relation indexed by our type precision (i.e., conversion) judgment e.g., $\mathcal{E}[\![\tau \leq \tau]\!]_\rho$. That is, migrationality would say that a more precise type had more errors than a less precise, which would be captured by the cross-language relation. The reason we didn't do this is that, as far as we understand, this would require defining a completely separate relation, which would capture much of what was in the compilers and convertibility relations again: essentially duplicating what we already need to do in order to prove soundness. By requiring that the conversion wrappers satisfy the ep-pair property that is at the heart of term precision, we believe that prevents us from ending up with conversions that do not make sense,

while avoiding building significant (possibly redundant) formalism. It is possible, in future work, researchers will figure out a different way of formulating migrationality directly that does not require this separate logical relation, or figure out how to formulate soundness in terms of such a cross-language relation, such that both statements could be formulated with a single relation, but as-yet, we believe the best option is to prove rich properties within the context of a single-language logical relation.

9.5 Compiler Correctness

Throughout this paper, we have not addressed the question of whether our compilers are correct! This is obviously of some concern to researchers, and it is not a trivial extension. In particular, in our setup we do not have a source semantics, as it is only defined in terms of its translation to the target. What's more, giving a source semantics including the foreign calls may require defining a multi-language, which is something that we dismissed at the outset, as multi-languages are often unrelated to the actual implementations that exist. However, correctness requires semantics to preserve, and it's not clear what other semantics would make sense.

9.6 Incremental Approach

The framework described throughout this paper can not only be used to prove soundness of FFI implementations, as we have demonstrated with out example, but it can also be used to guide improvements to implementations even without proving full soundness. Even practical implementations should have convertibility rules and the properties that they should have, even without a logical relation defined, should still hold. A common source of errors is that while FFI implementations must convert values with different representations (as without that, errors will almost certainly result), it is less common for them to assert that values conform to expectations. For example, a tagged sum might use two bits for the tag but only have three variants. In this case, to be sound the wrapper code should check that the tag is not the fourth possible value, and that interpretation, including understanding the set of target terms that make up the interpretation of the source type, can be understood even in the absence of a full formalism.

10 CONCLUSION

In this paper, we have described a general framework that can be used to prove the soundness of foreign function interfaces by building a logical relation that interprets source-language types as relations on error-approximating target-language terms. We have also shown how to define a convertibility relation between types that is inspired by the idea of type precision from gradual typing. We use the logical relation to prove strong properties about the glue code that implements convertibility, which is then inserted by the compiler between components from different languages.

Adding a convertibility judgment to source type systems gives us a means of implementing type checking of foreign function calls. Such type checking provides programmers sound feedback about what conversions are safely implementable. Moreover, extending the convertibility relation over time, would allow for future FFI data representations to be supported.

We have shown that this framework, since it is built by modeling interoperation after compilation to a target language, more closely matches how actual FFIs are implemented than prior work on semantics of language interoperability.

REFERENCES

Amal Ahmed, Andrew W. Appel, Christopher D. Richards, Kedar N. Swadi, Gang Tan, and Daniel C. Wang. 2010. Semantic Foundations for Typed Assembly Languages. 32, 3 (March 2010), 1–67.

Amal Ahmed and Matthias Blume. 2011. An equivalence-preserving CPS translation via multi-language semantics. In *Proceeding of the 16th ACM SIGPLAN international conference on Functional Programming, ICFP 2011, Tokyo, Japan, September 19-21, 2011*, Manuel M. T. Chakravarty, Zhenjiang Hu, and Olivier Danvy (Eds.). ACM, 431–444. https://doi.org/10.1145/2034773.2034830

Amal Ahmed, Derek Dreyer, and Andreas Rossberg. 2009. State-dependent representation independence. In *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009, Savannah, GA, USA, January 21-23, 2009*, Zhong Shao and Benjamin C. Pierce (Eds.). ACM, 340–353. https://doi.org/10.1145/1480881.1480925

Amal Ahmed, Matthew Fluet, and Greg Morrisett. 2007. L3 : A Linear Language with Locations. *Fundamenta Informaticae* 77, 4 (June 2007), 397–449.

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

Andrew W. Appel and David A. McAllester. 2001. An indexed model of recursive types for foundational proof-carrying code. *ACM Trans. Program. Lang. Syst.* 23, 5 (2001), 657–683. https://doi.org/10.1145/504709.504712

Edd Barrett, Carl Friedrich Bolz, Lukas Diekmann, and Laurence Tratt. 2016. Fine-grained Language Composition: A Case Study. In *30th European Conference on Object-Oriented Programming (ECOOP 2016) (Leibniz International Proceedings in Informatics (LIPIcs))*, Shriram Krishnamurthi and Benjamin S. Lerner (Eds.), Vol. 56. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 3:1–3:27. https://doi.org/10.4230/LIPIcs.ECOOP.2016.3

David M. Beazley. 1996. SWIG: An Easy to Use Tool for Integrating Scripting Languages with C and C++. In *Fourth Annual USENIX Tcl/Tk Workshop 1996, Monterey, California, USA, July 10-13, 1996*, Mark Diekhans and Mark Roseman (Eds.). USENIX Association. https://www.usenix.org/legacy/publications/library/proceedings/tcl96/beazley.html

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.

Karthikeyan Bhargavan, Barry Bond, Antoine Delignat-Lavaud, Cédric Fournet, Chris Hawblitzel, Catalin Hritcu, Samin Ishtiaq, Markulf Kohlweiss, Rustan Leino, Jay R. Lorch, Kenji Maillard, Jianyang Pan, Bryan Parno, Jonathan Protzenko, Tahina Ramananandro, Ashay Rane, Aseem Rastogi, Nikhil Swamy, Laure Thompson, Peng Wang, Santiago Zanella Béguelin, and Jean Karim Zinzindohoue. 2017. Everest: Towards a Verified, Drop-in Replacement of HTTPS. In *2nd Summit on Advances in Programming Languages, SNAPL 2017, May 7-10, 2017, Asilomar, CA, USA (LIPIcs)*, Benjamin S. Lerner, Rastislav Bodík, and Shriram Krishnamurthi (Eds.), Vol. 71. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 1:1–1:12. https://doi.org/10.4230/LIPIcs.SNAPL.2017.1

Matthias Blume. 2001. No-longer-foreign: Teaching an ML compiler to speak C "natively". *Electronic Notes in Theoretical Computer Science* 59, 1 (2001), 36–52.

Manuel MT Chakravarty. 1999. C->HASKELL, or Yet Another Interfacing Tool. In *Symposium on Implementation and Application of Functional Languages*. Springer, 131–148.

Derek Dreyer, Georg Neis, and Lars Birkedal. 2010. The impact of higher-order state and control effects on local relational reasoning. In *Proceeding of the 15th ACM SIGPLAN international conference on Functional programming, ICFP 2010, Baltimore, Maryland, USA, September 27-29, 2010*, Paul Hudak and Stephanie Weirich (Eds.). ACM, 143–156. https://doi.org/10.1145/1863543.1863566

Sigbjorn Finne, Daan Leijen, Erik Meijer, and Simon Peyton Jones. 1998. H/Direct: a binary foreign language interface for Haskell. In *Proceedings of the third ACM SIGPLAN international conference on Functional programming*. 153–162.

Aymeric Fromherz, Nick Giannarakis, Chris Hawblitzel, Bryan Parno, Aseem Rastogi, and Nikhil Swamy. 2019. A verified, efficient embedding of a verifiable assembly language. *PACMPL* 3, POPL (2019), 63:1–63:30. https://doi.org/10.1145/3290376

Michael Furr and Jeffrey S. Foster. 2005. Checking Type Safety of Foreign Function Calls. *SIGPLAN Not.* 40, 6 (June 2005), 62–72. https://doi.org/10.1145/1064978.1065019

Matthias Grimmer, Chris Seaton, Roland Schatz, Thomas Würthinger, and Hanspeter Mössenböck. 2015. High-performance cross-language interoperability in a multi-language runtime. In *Proceedings of the 11th Symposium on Dynamic Languages*. 78–90.

Martin Hirzel and Robert Grimm. 2007. Jeannie: granting java native interface developers their wishes. In *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2007, October 21-25, 2007, Montreal, Quebec, Canada*, Richard P. Gabriel, David F. Bacon, Cristina Videira Lopes, and Guy L. Steele Jr. (Eds.). ACM, 19–38. https://doi.org/10.1145/1297027.1297030

Simon Peyton Jones, Thomas Nordin, and Alastair Reid. 1997. GreenCard: a foreign-language interface for Haskell. In *Proc. Haskell Workshop*.

Neelakantan R. Krishnaswami, Pierre Pradic, and Nick Benton. 2015. Integrating Linear and Dependent Types. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*, Sriram K. Rajamani and David Walker (Eds.). ACM, 17–30. https://doi.org/10.1145/2676726.2676969

Byeongcheol Lee, Ben Wiedermann, Martin Hirzel, Robert Grimm, and Kathryn S. McKinley. 2010. Jinn: synthesizing dynamic bug detectors for foreign language interfaces. In *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2010, Toronto, Ontario, Canada, June 5-10, 2010*, Benjamin G. Zorn and Alexander Aiken (Eds.). ACM, 36–49. https://doi.org/10.1145/1806596.1806601

Siliang Li and Gang Tan. 2014. Exception analysis in the java native interface. *Science of Computer Programming* 89 (2014), 273–297.

Jacob Matthews and Robert Bruce Findler. 2007. Operational semantics for multi-language programs. In *Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2007, Nice, France, January 17-19, 2007*, Martin Hofmann and Matthias Felleisen (Eds.). ACM, 3–10. https://doi.org/10.1145/1190216.1190220

George C Necula, Scott McPeak, and Westley Weimer. 2002. CCured: Type-safe retrofitting of legacy code. In *Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 128–139.

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, William J. Bowman, and Amal Ahmed. 2016. Fully abstract compilation via universal embedding. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, ICFP 2016, Nara, Japan, September 18-22, 2016*, Jacques Garrigue, Gabriele Keller, and Eijiro Sumii (Eds.). ACM, 103–116. https://doi.org/10.1145/2951913.2951941

Peter-Michael Osera, Vilhelm Sjöberg, and Steve Zdancewic. 2012. Dependent interoperability. In *Proceedings of the sixth workshop on Programming Languages meets Program Verification, PLPV 2012, Philadelphia, PA, USA, January 24, 2012*, Koen Claessen and Nikhil Swamy (Eds.). ACM, 3–14. https://doi.org/10.1145/2103776.2103779

Daniel Patterson, Jamie Perconti, Christos Dimoulas, and Amal Ahmed. 2017. FunTAL: reasonably mixing a functional language with assembly. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*, Albert Cohen and Martin T. Vechev (Eds.). ACM, 495–509. https://doi.org/10.1145/3062341.3062347

James T. Perconti and Amal Ahmed. 2014. Verifying an Open Compiler Using Multi-language Semantics. In *Programming Languages and Systems - 23rd European Symposium on Programming, ESOP 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014, Proceedings (Lecture Notes in Computer Science)*, Zhong Shao (Ed.), Vol. 8410. Springer, 128–148. https://doi.org/10.1007/978-3-642-54833-8_8

Jonathan Protzenko, Jean Karim Zinzindohoué, Aseem Rastogi, Tahina Ramananandro, Peng Wang, Santiago Zanella Béguelin, Antoine Delignat-Lavaud, Catalin Hritcu, Karthikeyan Bhargavan, Cédric Fournet, and Nikhil Swamy. 2017. Verified low-level programming embedded in F. *PACMPL* 1, ICFP (2017), 17:1–17:29. https://doi.org/10.1145/3110261

Gabriel Scherer, Max S. New, Nick Rioux, and Amal Ahmed. 2018. FabULous Interoperability for ML and a Linear Language. In *Foundations of Software Science and Computation Structures - 21st International Conference, FOSSACS 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings (Lecture Notes in Computer Science)*, Christel Baier and Ugo Dal Lago (Eds.), Vol. 10803. Springer, 146–162. https://doi.org/10.1007/978-3-319-89366-2_8

Zhong Shao and Valery Trifonov. 1998. Type-directed continuation allocation. In *International Workshop on Types in Compilation*. Springer, 116–135.

Jeremy G Siek, Michael M Vitousek, Matteo Cimini, and John Tang Boyland. 2015. Refined criteria for gradual typing. In *1st Summit on Advances in Programming Languages (SNAPL 2015)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

Gang Tan, Andrew W Appel, Srimat Chakradhar, Anand Raghunathan, Srivaths Ravi, and Daniel Wang. 2006. Safe Java native interface. In *Proceedings of IEEE International Symposium on Secure Software Engineering*, Vol. 97. Citeseer, 106.

Jesse Tov and Riccardo Pucella. 2010. Stateful Contracts for Affine Types.

Valery Trifonov and Zhong Shao. 1999. Safe and principled language interoperation. In *European Symposium on Programming*. Springer, 128–146.

Alexi Turcotte, Ellen Arteca, and Gregor Richards. 2019. Reasoning About Foreign Function Interfaces Without Modelling the Foreign Language. In *33rd European Conference on Object-Oriented Programming (ECOOP 2019) (Leibniz International Proceedings in Informatics (LIPIcs))*, Alastair F. Donaldson (Ed.), Vol. 134. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 16:1–16:32. https://doi.org/10.4230/LIPIcs.ECOOP.2019.16

Jeremy Yallop, David Sheets, and Anil Madhavapeddy. 2018. A modular foreign function interface. *Science of Computer Programming* 164 (2018), 82–97.