Semantic Soundness for Language Interoperability

Daniel Patterson
Northeastern University
Boston, MA, USA
dbp@dbpmail.net

Noble Mushtak
Northeastern University
Boston, MA, USA
mushtak.n@northeastern.edu

Andrew Wagner
Northeastern University
Boston, MA, USA
ahwagner@ccs.neu.edu

Amal Ahmed
Northeastern University
Boston, MA, USA
amal@ccs.neu.edu

Abstract

Programs are rarely implemented in a single language, and thus questions of type soundness should address not only the semantics of a single language, but how it interacts with others. Even between type-safe languages, disparate features can frustrate interoperability, as invariants from one language can easily be violated in the other. In their seminal 2007 paper, Matthews and Findler [32] proposed a multi-language construction that augments the interoperating languages with a pair of *boundaries* that allow code from one language to be embedded in the other. While this technique has been widely applied, their syntactic source-level interoperability doesn't reflect practical implementations, where the behavior of interaction is only defined after compilation to a common target, and any safety must be ensured by target invariants or inserted target-level "glue code."

In this paper, we present a novel framework for the design and verification of sound language interoperability that follows an interoperation-after-compilation strategy. Language designers specify what data can be converted between types of the two languages via a convertibility relation $\tau_A \sim \tau_B$ ("τ_A is convertible to τ_B") and specify target-level glue code implementing the conversions. Then, by giving a semantic model of source-language types as sets of target-language terms, we can establish not only the meaning of our source types, but also *soundness of conversions*: i.e., whenever $\tau_A \sim \tau_B$, 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. With this, we can prove semantic type soundness for the entire system. We illustrate our framework via a series of case studies that demonstrate how our semantic interoperation-after-compilation approach allows us both to account for complex differences in language semantics and make efficiency trade-offs based on particularities of compilers or targets.

Keywords language interoperability, type soundness, semantics, logical relations

PL'18, January 01–03, 2018, New York, NY, USA
2018.

1 Introduction

All practical language implementations come with some way of interoperating with code written in a different language, usually via a foreign-function interface (FFI). This enables development of software systems with components written in different languages, whether to support legacy libraries or different programming paradigms. For instance, you might have a system with a high-performance data layer written in Rust interoperating with business logic implemented in OCaml. Sometimes, this interoperability is realized by targeting a common platform (e.g., Scala [39] and Clojure [22] for the JVM, or SML [10] and F# [46] for .NET). Other times, it is supported by libraries that insert boilerplate or "glue code" to mediate between the two languages (such as the binding generator SWIG [7], C->Haskell [16], OCaml-ctypes [51], NLFFI [13], Rust's bindgen [52], etc).

In 2007, Matthews and Findler [32] observed 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 system for abstractly modeling interactions between languages A and B by embedding the existing operational syntax and semantics into a multi-language AB and adding boundaries to mediate between the two. Specifically, a boundary ${}^{\tau_A}\mathcal{AB}^{\tau_B}(\cdot)$ allows a term $\mathrm{e_B}$ of type τ_B to be embedded in an A context that expects a term of type τ_A, and likewise for the boundary ${}^{\tau_B}\mathcal{BA}^{\tau_A}(\cdot)$. Operationally, the term ${}^{\tau_A}\mathcal{AB}^{\tau_B}(\mathrm{e_B})$ evaluates e_B using the B-language semantics to ${}^{\tau_A}\mathcal{AB}^{\tau_B}(\mathrm{v_B})$ and then a type-directed conversion takes the value $\mathrm{v_B}$ of type τ_B to an A-language term of type τ_A. There are often interesting design choices in deciding what conversions are available for a type, if any at all. One can then prove that the entire multi-language type system is sound by proving type safety for the multi-language, which includes the typing rules of both the embedded languages and the boundaries. This multi-language framework has inspired a significant amount of work on interoperability: between simple and dependently typed languages [40], between languages with unrestricted and substructural types [43, 48], between a high-level functional language and assembly [41], and between source and target languages of compilers [2, 36, 42].

Unfortunately, while Matthews-Findler-style boundaries give an elegant, abstract model for interoperability, they do not reflect reality. Indeed, a decade and a half later, there is little progress on assigning semantics to real multi-language systems. In most actual implementations, the source languages are compiled to components in a common target and glue code is inserted at the boundaries between them to account for data representations or calling conventions. While one could try to approach this problem by defining source-level boundaries, building a compiler for the multi-language, and then showing that the entire system is realized correctly, there are serious downsides to this approach. One is that if the two languages differ significantly, the multi-language may be significantly more than just an embedding of the evaluation rules of both languages (c.f. our last case study, as an implicitly garbage-collected language interoperating with a manually managed language may need to make the garbage collection explicit). And that doesn't even consider the fact that in practice, we usually have *existing* compiler implementations for one or both languages and wish to add (or extend) support for interoperability. Here, language designers' understanding of what datatypes *should* be convertible at the source level very much depends on how the sources are compiled and how data is (or could be) represented in the target, all information that is ignored by the multi-language approach. Moreover, certain conversions, even if possible, might be undesirable because the glue code needed to realize *safe* interoperability imposes too much runtime overhead.

In this paper, we present a framework for the *design* and *verification* of sound language interoperability, where both activities are connected to the actual implementation (of compilers and conversions). At the source, we still use Matthews-Findler-style boundaries, as our approach differs not in the source syntax but rather that instead of proving operational properties of that source, we instead prove semantic type soundness by defining a model of *source* types as sets of (or relations on) *target* terms. That is, the interpretation of a source type is the set of target terms that *behave as that type*. Guiding the design of these type interpretations are the compilers. This kind of model, often called a realizability model, is not a new idea — for instance, Benton and Zarfaty [12] and Benton and Tabareau [11] used such models to prove type soundness, but their work was limited to a single source language. By interpreting the types of two source languages as sets of terms in a common target, we enable rich reasoning about interoperability. Using the model, we can then give meaning to a boundary ${}^{\tau_B}\mathcal{BA}^{\tau_A}(\cdot)$: there is a bit of target code that, when given a target term that is in the model of the type τ_A, results in a target term in the model of type τ_B.

A realizability model is valuable not only for proving soundness, but for reasoning about the *design* of interoperability. For example, we can ask if a particular type in one language is *the same* as a type in the other language. This is true if the same set of target terms inhabits both types, and in this case conversions between the types should do nothing. More generally, opportunities for efficient conversions may only become apparent upon looking at how source types and invariants are represented (or realized) in the target. Since interoperability is a design challenge, with tradeoffs just like any other—performance high among them—working with the ability to understand all the pieces is a tremendous advantage.

Contributions To demonstrate the use and benefits of our framework, we present three case studies that illustrate different kinds of challenges for interoperability. In each case, we compile to an untyped target language.

1. **Shared-Memory Interoperability (§2):** We consider how mutable references can be exchanged between two languages and what properties must hold of stored data for aliasing to be safe. We show that to avoid copying mutable data — without having to wrap references in guards or chaperones [45] — convertible reference types must be inhabited by the *very same* set of target terms.
2. **Affine & Unrestricted (§3):** We consider how `MiniML`, a standard functional language with mutable references, can interact with **Affi**, an affine language. We show that affine code can be safely embedded in unrestricted code and vice versa by using runtime checks (only where necessary) to ensure that affine resources are used at most once.
3. **Memory Management & Polymorphism (§4):** We consider how `MiniML`, whose references are garbage collected, can interact with $\mathbf{L^3}$ [3], a language that uses linear capabilities to support safe strong updates to a manually managed mutable heap. We demonstrate not only when memory can be moved between languages, but also a type-level form of interoperability that allows generics to be used with $\mathbf{L^3}$ (which lacks type polymorphism) without violating any invariants of either language.

For each case study, we devise a novel realizability model. An interesting aspect of these models is that, since the target languages are untyped, statically enforced source invariants must be captured using either dynamic enforcement in target code or via invariants in the model. This demonstrates that our approach is viable even when working with existing target languages without rich static reasoning principles. For the first case study, we give a unary model (source types as *sets* of target terms). But for the next two studies (both of which include polymorphic languages), we give binary models (source types as *relations* on target terms), to allow our models to capture the relational properties of parametricity.

Definitions and proofs elided from this paper are provided in our extensive anonymous supplementary material.

2 Shared Memory

Aliased mutable data is challenging to deal with no matter the context, but aliasing across languages is especially difficult

$$
\begin{array}{llcl}
\texttt{RefHL} & \text{Type } \tau & ::= & \texttt{unit} \mid \texttt{bool} \mid \tau + \tau \mid \tau \times \tau \mid \tau \to \tau \mid \texttt{ref } \tau \\
 & \text{Expr. } e & ::= & () \mid \texttt{true} \mid \texttt{false} \mid x \mid \texttt{inl } e \mid \texttt{inr } e \mid (e, e) \\
 & & & \mid \texttt{fst } e \mid \texttt{snd } e \mid \texttt{if } e\ e\ e \mid \lambda x : \tau.e \mid e\ e \\
 & & & \mid \texttt{match } e\ x\{e\}\ y\{e\} \mid \texttt{ref } e \mid !e \mid e := e \mid (\!|e|\!)_\tau \\
\textbf{RefLL} & \text{Type } \tau & ::= & \textbf{int} \mid [\tau] \mid \tau \to \tau \mid \textbf{ref } \tau \\
 & \text{Expr. } e & ::= & n \mid x \mid [e, \ldots] \mid e[e] \mid \lambda x : \tau.e \mid e\ e \mid e + e \\
 & & & \mid \textbf{if0 } e\ e\ e \mid \textbf{ref } e \mid !e \mid e := e \mid (\!|e|\!)_\tau
\end{array}
$$

Figure 1. Syntax for RefHL and RefLL.

because giving a pointer to a foreign language can allow for *unknown* data to be written to its address. Specifically, if the pointer has a particular type in the host language, then only certain data should be written to it, but the foreign language may not respect or even know about that restriction. One existing approach to this problem is to create proxies, where data is guarded or converted before being read or written [17, 31, 45]. While sound, this comes with significant runtime overhead. Here, our framework allows a different approach.

Languages In this case study, we explore this problem using two simply-typed functional source languages with dynamically allocated mutable references, RefHL and RefLL (for "higher-level" and "lower-level"). RefHL has boolean, sum, and product types, whereas RefLL has arrays ($[e_1, \ldots, e_n] : [\tau]$). Their syntax is given in Fig. 1 and their static semantics — which are entirely standard — are elided (see supplementary material). These two languages are compiled (Fig. 3—note that we write e^+ to indicate e', where $e \rightsquigarrow e'$) into an untyped stack-based language called StackLang (inspired by [28]), whose syntax and small-step operational semantics — a relation on configurations $\langle H; S; P\rangle$ comprised of a heap, stack, and program — are given in Fig. 2; here we describe a few highlights. First, we note that StackLang values include not only numbers, thunks, and locations, but arrays of values, a simplification we made for the sake of presentation. Second, notice the interplay between thunk and lam: thunks are suspended computations, whereas lam is an instruction (not a value) responsible solely for substitution[1]. We can see how these features are combined, or used separately, in our compilers (Fig. 3). Finally, note that for any instruction where the precondition on the stack is not met, the configuration steps to a program with fail Type (a dynamic type error), although we elide most of these reduction rules for space.

Convertibility In our source languages, we may syntactically embed a term from one language into the other using the boundary forms $(\!|e|\!)_{\tau_A}$ and $(\!|e|\!)_{\tau_B}$. The typing rules for boundary terms require that the boundary types be convertible, written $\tau_A \sim \tau_B$. Those typing rules are:

$$
\frac{\Gamma; \Gamma \vdash e : \tau_A \qquad \tau_A \sim \tau_B}{\Gamma; \Gamma \vdash (\!|e|\!)_{\tau_B}}
\qquad\qquad
\frac{\Gamma; \Gamma \vdash e : \tau_B \qquad \tau_A \sim \tau_B}{\Gamma; \Gamma \vdash (\!|e|\!)_{\tau_A} : \tau_A}
$$

[1] À la Levy's Call-by-push-value [30].

$$
\begin{array}{llcl}
\text{Program P} & ::= & \cdot \mid i, P \qquad \text{Value v} ::= n \mid \text{thunk } P \mid \ell \mid [v, \ldots] \\
\text{Instruction i} & ::= & \text{push v} \mid \text{add} \mid \text{less?} \mid \text{if0 P P} \mid \text{lam x.P} \mid \text{call} \\
 & & \mid \text{idx} \mid \text{len} \mid \text{alloc} \mid \text{read} \mid \text{write} \mid \text{fail c} \\
\text{Error Code c} & ::= & \textsc{Type} \mid \textsc{Idx} \mid \textsc{Conv} \\
\text{Heap H} & ::= & \{\ell : v, \ldots\} \qquad \text{Stack S} ::= v, \ldots, v \mid \text{Fail c}
\end{array}
$$

$$
\begin{array}{lll}
\langle H; S; \text{push v}, P\rangle & \to \langle H; S, v; P\rangle & (S \neq \text{Fail c}) \\
\langle H; S, n', n; \text{add}, P\rangle & \to \langle H; S, (n + n'); P\rangle & \\
\langle H; S, n', n; \text{less?}, P\rangle & \to \langle H; S, b; P\rangle & (b = 0 \text{ if } n < n' \text{ else } 1) \\
\langle H; S, n; \text{if0 } P_1\ P_2, P\rangle & \to \langle H; S; P_i, P\rangle & (i = 1 \text{ if } n = 0 \text{ else } 2) \\
\langle H; S; \text{if0 } P_1\ P_2, P\rangle & \to \langle H; S; \text{fail } \textsc{Type}\rangle & (S \neq S', n) \\
\langle H; S, v; \text{lam x}.P_1, P_2\rangle & \to \langle H; S; [x \mapsto v]P_1, P_2\rangle & \\
\langle H; S, \text{thunk } P_1; \text{call}, P_2\rangle & \to \langle H; S; P_1, P_2\rangle & \\
\langle H; S, [v_0, \ldots, v_{n'}], n; \text{idx}, P\rangle & \to \langle H; S, v_n; P\rangle & (n \in [0, n']) \\
\langle H; S, [v_0, \ldots, v_{n'}], n; \text{idx}, P\rangle & \to \langle H; S; \text{fail } \textsc{Idx}\rangle & (n \notin [0, n']) \\
\langle H; S, [v_0, \ldots, v_n]; \text{len}, P\rangle & \to \langle H; S, (n + 1); P\rangle & \\
\langle H; S, v; \text{alloc}, P\rangle & \to \langle H \uplus \{\ell : v\}; S, \ell; P\rangle & \\
\langle H \uplus \{\ell : v\}; S, \ell; \text{read}, P\rangle & \to \langle H \uplus \{\ell : v\}; S, v; P\rangle & \\
\langle H \uplus \{\ell : \_\}; S, \ell, v; \text{write}, P\rangle & \to \langle H \uplus \{\ell : v\}; S; P\rangle & \\
\langle H; S; \text{fail c}, P\rangle & \to \langle H; \text{Fail c}; \cdot\rangle &
\end{array}
$$

Figure 2. Syntax and selected operational semantics for StackLang (most fail Type cases elided).

$$
\begin{array}{c}
\text{SWAP} \triangleq \text{lam x.(lam y.push x, push y)} \\
\text{DROP} \triangleq \text{lam x.()} \qquad \text{DUP} \triangleq \text{lam x.(push x, push x)}
\end{array}
$$

$$
\begin{array}{lcl}
() \rightsquigarrow \text{push } 0 & \mid & x \rightsquigarrow \text{push x} \\
\texttt{true} \mid \texttt{false} & \rightsquigarrow & \text{push } \langle 0 \mid 1\rangle \\
\texttt{inl } e \mid \texttt{inr } e & \rightsquigarrow & e^+, \text{lam x.(push } [\langle 0 \mid 1\rangle, x]) \\
\texttt{if } e\ e_1\ e_2 & \rightsquigarrow & e^+, \text{if0 } e_1^+\ e_2^+ \\
\texttt{match } e & \rightsquigarrow & e^+, \text{DUP, push 1, idx, SWAP, push 0,} \\
\quad x\{e_1\}\ y\{e_2\} & & \text{idx, if0 (lam x.}e_1^+\text{) (lam y.}e_2^+\text{)} \\
(e_1, e_2) & \rightsquigarrow & e_1^+, e_2^+, \text{lam } x_2, x_1.(\text{push } [x_1, x_2]) \\
\texttt{fst } e \mid \texttt{snd } e & \rightsquigarrow & e^+, \text{push } \langle 0 \mid 1\rangle, \text{idx} \\
e_1\ e_2 & \rightsquigarrow & e_1^+, e_2^+, \text{SWAP, call} \\
\texttt{ref } e & \rightsquigarrow & e^+, \text{alloc} \\
e_1 := e_2 & \rightsquigarrow & e_1^+, e_2^+, \text{write, push } 0 \\
(\!|e|\!)_\tau & \rightsquigarrow & e^+, C_{\tau \mapsto \tau} \\
n \rightsquigarrow \text{push n} & \mid & e_1 + e_2 \rightsquigarrow e_1^+, e_2^+, \text{SWAP, add} \\
[e_1, \ldots, e_n] & \rightsquigarrow & e_1^+, \ldots, e_n^+, \text{lam } x_n, \ldots, x_1. \\
 & & \quad (\text{push } [x_1, \ldots, x_n]) \\
e_1[e_2] & \rightsquigarrow & e_1^+, e_2^+, \text{idx} \\
\lambda x : \tau.e & \rightsquigarrow & \text{push (thunk lam x.}e^+\text{)} \\
!e & \rightsquigarrow & e^+, \text{read} \\
(\!|e|\!)_\tau & \rightsquigarrow & e^+, C_{\tau \mapsto \tau}
\end{array}
$$

Figure 3. Selections from compilers for RefHL and RefLL.

Note that the convertibility judgment is a declarative, extensible judgment that describes closed types in one language that are interconvertible with closed types in the other, allowing for the possibility of well-defined runtime errors. By separating this judgment from the rest of the type system, the language designer can allow additional conversions to be added later, whether by implementers or even end-users. The second thing to note is that this presentation allows for open terms to be converted, so we must maintain a type environment for both languages during typechecking (both Γ and Γ), as we have to carry information from the site of binding—possibly through conversion boundaries—to the site of variable use. A simpler system, which we have explored, would only allow closed terms to be converted. In that case, the typing rules still use the $\tau_A \sim \tau_B$ judgment but do not thread foreign environments (using only Γ for `RefHL` and only Γ for **RefLL**).

We present, in Fig. 4, some of the convertibility rules we have defined for this case study (we elide $\tau_1 \times \tau_2 \sim [\tau]$), which come with target-language instruction sequences that perform the conversions, written $C_{\tau_A \mapsto \tau_B}$ (some are no-ops). An instruction sequence $C_{\tau_A \mapsto \tau_B}$, while ordinary target code, when appended to a program in the model at type τ_A, should result in a program in the model at type τ_B. An implementer can write these conversions based on understanding of the sets of target terms that inhabit each source type, before defining a proper semantic model (or possibly, without defining one, if formal soundness is not required). They would do this based on inspection of the compiler and the target.

From Fig. 3, we see that `bool` and **int** both compile to target integers, and importantly, that `if` compiles to if0, which means the compiler interprets `false` as any non-zero integer. Hence, conversions between `bool` and **int** are identities.

For sums, we use the tags 0 and 1, and as for `if`, we use if0 to branch in the compilation of `match`. Therefore, we can choose if the `inl` and `inr` tags should be represented by 0 and 1, or by 0 and any other integer n. Given that tags could be added later, we choose the former, thus converting a sum to an array of integers is mostly a matter of converting the payload. In the other direction, we have to handle the case that the array is too short, and error.

The final case, between `ref bool` and **ref int**, is the reason for this case study. Intuitively, if you exchange pointers, any value of the new type can now be written at that address, and thus must have been compatible with the old type (as aliases could still exist). Thus, we require that `bool` and **int** are somehow "identical" in the target, so conversions are unnecessary.

Semantic Model Declaring that a type `bool` is "identical" to **int** or that τ is convertible to τ and providing the conversion code is not sufficient for soundness. In order to show that these conversions are sound, and indeed to understand which conversions are even possible, we define a model for source types that is inhabited by target terms. Since both languages compile to the same target, the range of their relations will be the same (i.e., composed of terms and values from StackLang), and thus we will be able to easily and directly compare the inhabitants of two types, one from each language.

Our model, which is a standard step-indexed unary logical relation for a language with mutable state (essentially

$$\frac{}{C_{\mathtt{bool}\mapsto\mathbf{int}}, C_{\mathbf{int}\mapsto\mathtt{bool}} : \mathtt{bool} \sim \mathbf{int}}$$

$$\frac{}{C_{\mathtt{ref\ bool}\mapsto\mathbf{ref\ int}}, C_{\mathbf{ref\ int}\mapsto\mathtt{ref\ bool}} : \mathtt{ref\ bool} \sim \mathbf{ref\ int}}$$

$$\frac{C_{\tau_1\mapsto\mathbf{int}}, C_{\mathbf{int}\mapsto\tau_1} : \tau_1 \sim \mathbf{int} \qquad C_{\tau_2\mapsto\mathbf{int}}, C_{\mathbf{int}\mapsto\tau_2} : \tau_2 \sim \mathbf{int}}{C_{\tau_1+\tau_2\mapsto[\mathbf{int}]}, C_{[\mathbf{int}]\mapsto\tau_1+\tau_2} : \tau_1 + \tau_2 \sim [\mathbf{int}]}$$

$$C_{\mathtt{bool}\mapsto\mathbf{int}} \triangleq C_{\mathbf{int}\mapsto\mathtt{bool}} \triangleq C_{\mathtt{ref\ bool}\mapsto\mathbf{ref\ int}} \triangleq C_{\mathbf{ref int}\mapsto\mathtt{ref\ bool}} \triangleq \cdot$$

$C_{\tau_1 + \tau_2\mapsto[\mathbf{int}]} \triangleq$
  DUP, push 1, idx, SWAP,push 0, idx, DUP,
  if0 (SWAP, $C_{\tau_1\mapsto\mathbf{int}}$)(SWAP, $C_{\tau_2\mapsto\mathbf{int}}$),lam x_v.lam x_t.push $[x_t, x_v]$
$C_{[\mathbf{int}]\mapsto\tau_1 + \tau_2} \triangleq$
  DUP, len, push 2, SWAP,less?, if0 fail Conv,
  DUP, push 1, idx, SWAP,push 0, idx, DUP,
  if0 (SWAP, $C_{\mathbf{int}\mapsto\tau_1}$)(DUP, push −1, add,
   if0 (SWAP, $C_{\mathbf{int}\mapsto\tau_2}$)(fail Conv)),lam x_v.lam x_t.push $[x_t, x_v]$

Figure 4. Conversions for `RefHL` and **RefLL**.

following Ahmed [4]), is presented with some parts elided in Fig. 5 (see supplementary material).

We give value interpretations for each source type τ, written $\mathcal{V}[\![\tau]\!]$ as sets of target *values* v paired with *worlds* W that inhabit that type. A world W is comprised of a step index k and a *heap typing* Ψ, which maps locations to type interpretations in *Typ*. As is standard, *Typ* is the set of valid type interpretations, which must be closed under world extension. A future world W' extends W, written $W' \sqsupseteq W$, if W' has a potentially lower step budget $j \le W.k$ and all locations in $W.\Psi$ still have the same types (to approximation j).

Intuitively, $(W, \mathrm{v}) \in \mathcal{V}[\![\tau]\!]$ says that the target value v belongs to (or behaves like a value of) type τ in world W. For example, $\mathcal{V}[\![\mathtt{unit}]\!]$ is inhabited by 0 in any world. A more interesting case is $\mathcal{V}[\![\mathtt{bool}]\!]$, which is the set of all target integers, not just 0 and 1, though we could choose to define our model that way (provided we compiled `bool`s to 0 or 1). An array $\mathcal{V}[\![[\tau]]\!]$ is inhabited by an array of target values v_i in world W if each v_i is in $\mathcal{V}[\![\tau]\!]$ with W.

Functions follow the standard pattern for logical relations, appropriately adjusted for our stack-based target language: $\mathcal{V}[\![\tau_1 \to \tau_2]\!]$ is inhabited by values thunk lam x.P in world W if, for any future world W' and argument v in $\mathcal{V}[\![\tau_1]\!]$ at that world, the result of substituting the argument into the body ($[x\mapsto\mathrm{v}]\mathrm{P}$) is in the expression relation at the result type $\mathcal{E}[\![\tau_2]\!]$. Reference types $\mathcal{V}[\![\mathtt{ref}\ \tau]\!]$ are inhabited by a location ℓ in world W if the current world's heap typing $W.\Psi$ maps ℓ to the value relation $\mathcal{V}[\![\tau]\!]$ approximated to the step index in the world $W.k$. (The j-approximation of a type, written $\lfloor\mathcal{V}[\![\tau]\!]\rfloor_j$, restricts $\mathcal{V}[\![\tau]\!]$ to inhabitants with worlds in $World_j$.)

Our expression relation $\mathcal{E}[\![\tau]\!]$ defines when a program P in world W behaves as a computation of type τ. It says that for any heap H that satisfies the current world W, written $H : W$, and any non-Fail stack S, if the machine $\langle \mathsf{H}; \mathsf{S}; \mathsf{P}\rangle$ terminates in j steps (where j is less than our step budget $W.k$), then either it ran to a non-type error or there exists some value v and some future world W' such that the resulting stack S' is the original stack with v on top, the resulting heap H' satisfies the future world W' and W' and v are in $\mathcal{V}[\![\tau]\!]$.

At the bottom of Fig. 5, we show a syntactic shorthand, $[\![\Gamma;\Gamma \vdash \mathsf{e} : \tau]\!]$, for showing that well-typed source programs, when compiled and closed off with well-typed substitutions γ that map variables to target values, are in the expression relation. Note $\mathcal{G}[\![\Gamma]\!]$ contains closing substitutions γ in world W that assign every $x : \tau \in \Gamma$ to a v such that $(W, \mathsf{v}) \in \mathcal{V}[\![\tau]\!]$.

With our logical relation in hand, we can now state formal properties about our convertibility judgments.

Lemma 2.1 (Convertibility Soundness).
If $\tau \sim \tau$, *then* $\forall (W,P) \in \mathcal{E}[\![\tau]\!].(W,(P, C_{\tau\mapsto\tau})) \in \mathcal{E}[\![\tau]\!] \wedge \forall (W,P) \in \mathcal{E}[\![\tau]\!].(W,(P, C_{\tau\mapsto\tau})) \in \mathcal{E}[\![\tau]\!]$.

Proof. We sketch the `ref bool` ∼ **ref int** case; (rest elided, see supplementary material). For `ref bool` ∼ **ref int**, what we need to show is that given any expression in $\mathcal{E}[\![$`ref bool`$]\!]$, if we apply the conversion (which does nothing), the result will be in $\mathcal{E}[\![$**ref int**$]\!]$. That requires we show $\mathcal{V}[\![$`ref bool`$]\!] = \mathcal{V}[\![$**ref int**$]\!]$.

The value relation at a reference type says that if you look up the location ℓ in the heap typing of the world ($W.\Psi$), you will get the value interpretation of the type. That means a `ref bool` must be a location ℓ that, in the model, points to the value interpretation of `bool` (i.e., $\mathcal{V}[\![$`bool`$]\!]$). In our model, this must be true for all future worlds, which makes sense for ML-style references. Thus, for this proof to go through, $\mathcal{V}[\![$`bool`$]\!]$ must be the same as $\mathcal{V}[\![$**int**$]\!]$, which it is. □

Once we have proved Lemma 2.1, we can prove semantic type soundness in the standard two-step way for our entire system. First, for each source typing rule, we define a compatibility lemma that is a semantic analog to that rule. For example, the compatibility lemma for the conversion typing rule, shown here, requires the proof of Lemma 2.1 to go through:

$$[\![\Gamma;\Gamma \vdash \mathsf{e} : \tau]\!] \wedge \tau \sim \tau \implies [\![\Gamma;\Gamma \vdash (\!|\mathsf{e}|\!)_{\tau} : \tau]\!]$$

Once we have all compatibility lemmas we can prove the following theorems:

Theorem 2.2 (Fundamental Property).
If $\Gamma;\Gamma \vdash \mathsf{e} : \tau$ *then* $[\![\Gamma;\Gamma \vdash \mathsf{e} : \tau]\!]$ *and if* $\Gamma;\Gamma \vdash \mathsf{e} : \tau$ *then* $[\![\Gamma;\Gamma \vdash \mathsf{e} : \tau]\!]$.

Theorem 2.3 (Type Safety for **RefLL**). *If* $\cdot;\cdot \vdash \mathsf{e} : \tau$ *then for any* $\mathsf{H} : W$, *if* $\langle \mathsf{H}; \cdot; \mathsf{e}^+\rangle \xrightarrow{*} \langle \mathsf{H}'; \mathsf{S}'; \mathsf{P}'\rangle$, *then either* $\langle \mathsf{H}'; \mathsf{S}'; \mathsf{P}'\rangle \longrightarrow \langle \mathsf{H}''; \mathsf{S}''; \mathsf{P}''\rangle$, *or* $\mathsf{P}' = \cdot$ *and either* $\mathsf{S}' = \mathsf{Fail}\ \mathsf{c}$ *for some* $\mathsf{c} \in \{\textsc{Conv}, \textsc{Idx}\}$ *or* $\mathsf{S}' = \mathsf{v}$.

Theorem 2.4 (Type Safety for `RefHL`). *If* $\cdot;\cdot \vdash \mathsf{e} : \tau$ *then for any* $\mathsf{H} : W$, *if* $\langle \mathsf{H}; \cdot; \mathsf{e}^+\rangle \xrightarrow{*} \langle \mathsf{H}'; \mathsf{S}'; \mathsf{P}'\rangle$, *then either*

$$
\begin{aligned}
AtomVal_n &= \{(W, \mathsf{v}) \mid W \in World_n\}\\
World_n &= \{(k, \Psi) \mid k < n \wedge \Psi \subset HeapTy_k\}\\
HeapTy_n &= \{\ell \mapsto Typ_n, \ldots\}\\
Typ_n &= \{R \in 2^{AtomVal_n} \mid \forall (W,\mathsf{v}) \in R.\\
&\qquad \forall W'.\ W \sqsubseteq W' \implies (W', \mathsf{v}) \in R\}
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{V}[\![\mathtt{bool}]\!] &= \{(W, \mathsf{n})\}\\
\mathcal{V}[\![\mathtt{unit}]\!] &= \{(W, 0)\}\\
\mathcal{V}[\![\tau_1 + \tau_2]\!] &= \{(W, [0, \mathsf{v}]) \mid (W, \mathsf{v}) \in \mathcal{V}[\![\tau_1]\!]\}\\
&\quad \cup \{(W, [1, \mathsf{v}]) \mid (W, \mathsf{v}) \in \mathcal{V}[\![\tau_2]\!]\}\\
\mathcal{V}[\![\tau_1 \longrightarrow \tau_2]\!] &= \{(W, \mathsf{thunk\ lam\ x.P}) \mid\\
&\qquad \forall \mathsf{v}, W' \sqsupset W.\ (W', \mathsf{v}) \in \mathcal{V}[\![\tau_1]\!]\\
&\qquad\quad \implies (W', [\mathsf{x}\mapsto \mathsf{v}]\mathsf{P}) \in \mathcal{E}[\![\tau_2]\!]\}\\
\mathcal{V}[\![\mathtt{ref}\ \tau]\!] &= \{(W, \ell) \mid W.\Psi(\ell) = \lfloor \mathcal{V}[\![\tau]\!] \rfloor_{W.k}\}\\
\mathcal{V}[\![\mathbf{int}]\!] &= \{(W, \mathsf{n})\}\\
\mathcal{V}[\![[\tau]]\!] &= \{(W, [\mathsf{v}_1, \ldots, \mathsf{v}_n]) \mid (W, \mathsf{v}_i) \in \mathcal{V}[\![\tau]\!]\}\\
\mathcal{V}[\![\tau_1 \rightarrow \tau_2]\!] &= \{(W, \mathsf{thunk\ lam\ x.P}) \mid\\
&\qquad \forall \mathsf{v}, W' \sqsupset W.\ (W', \mathsf{v}) \in \mathcal{V}[\![\tau_1]\!]\\
&\qquad\quad \implies (W', [\mathsf{x}\mapsto \mathsf{v}]\mathsf{P}) \in \mathcal{E}[\![\tau_2]\!]\}\\
\mathcal{V}[\![\mathbf{ref}\ \tau]\!] &= \{(W, \ell) \mid W.\Psi(\ell) = \lfloor \mathcal{V}[\![\tau]\!] \rfloor_{W.k}\}
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{E}[\![\tau]\!] = \{(W, P) \mid{}& \forall \mathsf{H}{:}W, S \neq \mathsf{Fail}\ \_, \mathsf{H}', S', j < W.k.\\
&\langle \mathsf{H}; \mathsf{S}; P\rangle \xrightarrow{j} \langle \mathsf{H}'; \mathsf{S}'; \cdot\rangle \implies \mathsf{S}' = \mathsf{Fail}\ \mathsf{c} \wedge \mathsf{c} \in \{\textsc{Conv}, \textsc{Idx}\}\\
&\vee \exists \mathsf{v}, W' \sqsupseteq W.\ \big(\mathsf{S}' = \mathsf{S}, \mathsf{v} \wedge \mathsf{H}' : W' \wedge (W', \mathsf{v}) \in \mathcal{V}[\![\tau]\!]\big)\}
\end{aligned}
$$

$$
\begin{aligned}
[\![\Gamma;\Gamma \vdash \mathsf{e} : \tau]\!] \equiv{}& \forall W\, \gamma_\Gamma\, \gamma_\Gamma.(W, \gamma_\Gamma) \in \mathcal{G}[\![\Gamma]\!] \wedge (W, \gamma_\Gamma) \in \mathcal{G}[\![\Gamma]\!]\\
&\implies (W, \mathsf{close}(\gamma_\Gamma, \mathsf{close}(\gamma_\Gamma, \mathsf{e}^+))) \in \mathcal{E}[\![\tau]\!]
\end{aligned}
$$

$$
\begin{aligned}
[\![\Gamma;\Gamma \vdash \mathsf{e} : \tau]\!] \equiv{}& \forall W\, \gamma_\Gamma\, \gamma_\Gamma.(W, \gamma_\Gamma) \in \mathcal{G}[\![\Gamma]\!] \wedge (W, \gamma_\Gamma) \in \mathcal{G}[\![\Gamma]\!]\\
&\implies (W, \mathsf{close}(\gamma_\Gamma, \mathsf{close}(\gamma_\Gamma, \mathsf{e}^+))) \in \mathcal{E}[\![\tau]\!]
\end{aligned}
$$

Figure 5. Logical relation for `RefHL` and **RefLL**.

$\langle \mathsf{H}'; \mathsf{S}'; \mathsf{P}'\rangle \longrightarrow \langle \mathsf{H}''; \mathsf{S}''; \mathsf{P}''\rangle$, *or* $\mathsf{P}' = \cdot$ *and either* $\mathsf{S}' = \mathsf{Fail}\ \mathsf{c}$ *for some* $\mathsf{c} \in \{\textsc{Conv}, \textsc{Idx}\}$ *or* $\mathsf{S}' = \mathsf{v}$.

Discussion In addition to directly passing across pointers, there are two alternative conversion strategies, both of which our framework would accommodate: first, we could create a new location and copy and convert the data. This would allow the more flexible convertibility which does not require references to "identical" types, but would not allow aliasing, which may be desirable. Second, rather than converting `ref` τ and **ref** τ, we could instead convert (`unit` ⟶ τ) × (τ ⟶ `unit`) and (**unit** → τ) × (τ → **unit**) (assuming we had pairs)—i.e., read/write proxies to the reference (similar to that used in [17]). This allows aliasing, i.e., both languages reading / writing to the same location, and is sound for arbitrary convertibility relations, but it comes at a significant runtime cost, as we introduce overhead at each read / write.

The choice to use the encoding described in this case study, or either of these options, is not, of course, exclusive—we could provide different options for different types in the same system, depending on the performance characteristics we need.

$$
\begin{array}{lll}
\textsc{Affi} & & \\
\text{Type } \tau & ::= & \mathbf{unit} \mid \mathbf{bool} \mid \mathbf{int} \mid \tau \multimap \tau \mid \tau \mathrel{-\!\!\bullet} \tau \mid !\tau \mid \tau \& \tau \mid \tau \otimes \tau \\
\text{Expr. } \mathbf{e} & ::= & () \mid \mathbf{true} \mid \mathbf{false} \mid \mathbf{n} \mid \mathbf{x} \mid \mathbf{a}_{\LEFTcircle} \mid \lambda \mathbf{a}_{\LEFTcircle} : \tau.\mathbf{e} \\
 & & \mid \mathbf{e}\ \mathbf{e} \mid \llparenthesis \mathtt{e} \rrparenthesis_{\tau} \mid !\mathbf{v} \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}_{\bullet}, \mathbf{a}'_{\bullet}) = \mathbf{e}\ \mathbf{in}\ \mathbf{e}' \\
\text{Value } \mathbf{v} & ::= & () \mid \lambda \mathbf{a}_{\LEFTcircle} : \tau.\mathbf{e} \mid !\mathbf{v} \mid \langle \mathbf{e}, \mathbf{e}' \rangle \mid (\mathbf{v}, \mathbf{v}') \\
\text{Mode } \LEFTcircle & ::= & \circ \mid \bullet \\
\mathtt{MiniML} & & \\
\text{Type } \tau & ::= & \mathtt{unit} \mid \mathtt{int} \mid \tau \times \tau \mid \tau + \tau \mid \tau \to \tau \mid \forall \alpha.\tau \mid \alpha \mid \mathtt{ref}\ \tau \\
\text{Expr. } \mathtt{e} & ::= & () \mid \mathtt{n} \mid \mathtt{x} \mid (\mathtt{e}, \mathtt{e}) \mid \mathtt{fst}\ \mathtt{e} \mid \mathtt{snd}\ \mathtt{e} \mid \mathtt{inl}\ \mathtt{e} \mid \mathtt{inr}\ \mathtt{e} \\
 & & \mid \mathtt{match}\ \mathtt{e}\ \mathtt{x}\{\mathtt{e}\}\ \mathtt{y}\{\mathtt{e}\} \mid \lambda \mathtt{x} : \tau.\mathtt{e} \mid \mathtt{e}\ \mathtt{e} \mid \Lambda \alpha.\mathtt{e} \mid \mathtt{e}[\tau] \\
 & & \mid \mathtt{ref}\ \mathtt{e} \mid !\mathtt{e} \mid \mathtt{e} := \mathtt{e} \mid \llparenthesis \mathbf{e} \rrparenthesis_{\tau} \\
\text{LCVM} & & \\
\text{Expr e} & ::= & () \mid n \mid \ell \mid x \mid (e, e) \mid \text{fst } e \mid \text{snd } e \mid \text{inl } e \mid \text{inr } e \\
 & & \mid \text{if } e\ \{e\}\ \{e\} \mid \text{match } e\ x\{e\}\ y\{e\} \mid \text{let } x = e \text{ in } e \\
 & & \mid \lambda x\{e\} \mid e\ e \mid \text{ref } e \mid !e \mid e := e \mid \text{fail } c \\
\text{Values v} & ::= & () \mid n \mid \ell \mid (v, v) \mid \lambda x.e \\
\text{Err Code c} & ::= & \textsc{Type} \mid \textsc{Conv}
\end{array}
$$

Figure 6. Syntax for MiniML, Affi, and LCVM.

$$
\frac{\mathbf{a}_{\LEFTcircle} : \tau \in \Omega}{\Delta; \Gamma; \Gamma; \Omega \vdash \mathbf{a}_{\LEFTcircle} : \tau}
\qquad
\frac{\Delta; \Gamma; \Gamma; \Omega[\mathbf{a}_{\circ} := \tau_1] \vdash \mathbf{e} : \tau_2 \quad \mathrm{no}_{\bullet}(\Omega)}{\Delta; \Gamma; \Gamma; \Omega \vdash \lambda \mathbf{a}_{\circ} : \tau_1.\mathbf{e} : \tau_1 \multimap \tau_2}
$$

$$
\frac{\Delta; \Gamma; \Gamma; \Omega[\mathbf{a}_{\bullet} := \tau_1] \vdash \mathbf{e} : \tau_2}{\Delta; \Gamma; \Gamma; \Omega \vdash \lambda \mathbf{a}_{\bullet} : \tau_1.\mathbf{e} : \tau_1 \mathrel{-\!\!\bullet} \tau_2}
$$

$$
\frac{\Omega = \Omega_1 \uplus \Omega_2 \quad \Delta; \Gamma; \Gamma; \Omega_1 \vdash \mathbf{e}_1 : \tau_1 \mathrel{-\!\!\LEFTcircle} \tau_2 \quad \Delta; \Gamma; \Gamma; \Omega_2 \vdash \mathbf{e}_2 : \tau_1}{\Delta; \Gamma; \Gamma; \Omega \vdash \mathbf{e}_1\ \mathbf{e}_2 : \tau_2}
$$

$$
\frac{\begin{array}{c}\Omega = \Omega_1 \uplus \Omega_2 \\ \Delta; \Gamma; \Gamma; \Omega_1 \vdash \mathbf{e} : \tau_1 \otimes \tau_2 \quad \Delta; \Gamma; \Gamma; \Omega_2[\mathbf{a} := \tau_1, \mathbf{a}' := \tau_1] \vdash \mathbf{e}' : \tau'\end{array}}{\Delta; \Gamma; \Gamma; \Omega \vdash \mathbf{let}\ (\mathbf{a}_{\bullet}, \mathbf{a}'_{\bullet}) = \mathbf{e}\ \mathbf{in}\ \mathbf{e}' : \tau'}
$$

$$
\frac{\Omega = \Omega_{\mathbf{e}} \uplus \Omega' \quad \mathrm{no}_{\bullet}(\Omega_{\mathbf{e}}) \quad \Delta; \Gamma; \Gamma; \Omega_{\mathbf{e}} \vdash \mathbf{e} : \tau \quad \_ : \tau \sim \tau}{\Gamma; \Omega; \Delta; \Gamma \vdash \llparenthesis \mathbf{e} \rrparenthesis_{\tau} : \tau}
$$

$$
\frac{\Omega = \Omega_1 \uplus \Omega_2 \quad \Gamma; \Omega_1; \Delta; \Gamma \vdash \mathtt{e}_1 : \tau_1 \to \tau_2 \quad \Gamma; \Omega_2; \Delta; \Gamma \vdash \mathtt{e}_2 : \tau_1}{\Delta; \Gamma; \Gamma; \Omega \vdash \mathtt{e}_1\ \mathtt{e}_2 : \tau_2}
$$

Figure 7. Selected statics for Affi and MiniML.

3 Affine & Unrestricted

In our second case study, we consider an affine language, Affi, interacting with an unrestricted one, MiniML. We enforce Affi's at-most-once variable use dynamically in the target using the well-known technique described, e.g., in [48], where affine resources are protected behind thunks with stateful flags that raise runtime errors the second time the thunk is forced. However, an interesting and challenging aspect of our case study is that we only want to use dynamic enforcement when we lack static assurance that an affine variable will be use at most once.

Languages We present the syntax of Affi, MiniML, and our untyped Scheme-like functional target LCVM in Fig. 6 and selected static semantics in Fig. 7 (see supplementary material). Our target LCVM is untyped, with functions, pattern matching, mutable references, and a standard operational semantics defined via steps $\langle H, e \rangle \to \langle H', e' \rangle$ over heap and expression pairs. As in the previous case study, we will support open terms across language boundaries, and thus need to carry environments for both languages throughout our typing judgments.

To avoid unnecessary dynamic enforcement, we have two kinds of affine function types in Affi: $\multimap$ and $\mathrel{-\!\!\bullet}$.[2] We introduce a distinction between Affi functions (and thus bindings) that may be passed across the boundary (our "dynamic" affine arrows $\multimap$, written with a hollow circle and bind dynamic affine variables $\mathbf{a}_{\circ}$), and ones that will only ever be used within Affi (our "static" affine arrows $\mathrel{-\!\!\bullet}$, written with a solid circle and bind static affine variables $\mathbf{a}_{\bullet}$).

[2] In our supplementary materials, we also present a complete case study with a simpler variant of Affi, which does not distinguish $\multimap$/$\mathrel{-\!\!\bullet}$ and thus does dynamic enforcement even on affine variables that have no interaction with unrestricted code.

The intention is that this distinction is to be the target of inference — i.e., a procedure should be able to, starting from boundaries, determine which functions (and their clients) should be dynamic, and the rest can remain static. To simplify our presentation, we do not show such an inference procedure.

We can see in Fig. 7 how Affi's affine-variable environment Ω is maintained: variables are introduced by lambda and tensor-destructuring let, and environments are split across subterms, but all bindings are not required to be used, as we can see, in the variable rule. (In the full rules in supplementary material, a similar pattern shows up for base types). Since affine resources can exist within unrestricted MiniML terms, our affine environments Ω need to be split, even in MiniML typing rules.

Note that we do not allow a dynamic function $\lambda \mathbf{a}_{\circ} : \_.\mathbf{e}$ to close over static resources, as it may be duplicated if passed to MiniML, and thus the static resources would be unprotected. However, we do allow a dynamic function to accept a static closure as argument. This is safe because the dynamic guards will ensure that the static closure is called at most once. Once called, any static resources in its body will be used safely because the static closure typechecked.

We present selections of our compilers in Fig. 8 that highlight the interesting cases: how we compile variables, binders, and application. In the application cases, we can see that static variables do not introduce the overhead that dynamic variables have (see the thunk macro at the top of the figure that errors on second invocation).

Convertibility We define convertibility relations and conversions for Affi and MiniML, highlighting selections in Fig. 9 (see supplementary material for elided $\mathbf{unit} \sim \mathtt{unit}$

$$\mathsf{thunk}(e) \triangleq \mathsf{let}\ r_{fr} = \mathsf{ref}\ 1\ \mathsf{in}\ \lambda\_.\{\mathsf{if}\ !r_{fr}\ \{\mathsf{fail}\ \textsc{Conv}\}\ \{r_{fr} := 0; e\}\}$$

$$() \rightsquigarrow () \quad n \rightsquigarrow n \quad \lambda x : \tau.e \rightsquigarrow \lambda x.\{e^+\} \quad \mathbf{true}/\mathbf{false} \rightsquigarrow 0/1$$

$$a_\circ \rightsquigarrow a_\circ\ () \quad a_\bullet \rightsquigarrow a_\bullet \quad \lambda a_{\circ/\bullet} : \tau.e \rightsquigarrow \lambda a_{\circ/\bullet}.\{e^+\}$$

$$\begin{array}{lcl} (e_1 : \tau_1 \multimap \tau_2)\ e_2 & \rightsquigarrow & e_1^+\ (\mathsf{let}\ x = e_2^+\ \mathsf{in}\ \mathsf{thunk}(x)) \\ (e_1 : \tau_1 \multimap\!\bullet\ \tau_2)\ e_2 & \rightsquigarrow & {e_1}^+\ {e_2}^+ \\ \mathbf{let}\ (a_\bullet, a'_\bullet) = e_1\ \mathbf{in}\ e_2 & \rightsquigarrow & \mathsf{let}\ x_{fresh} = e^+, \\ & & a_\bullet = \mathsf{fst}\ x_{fresh}, \\ & & a'_\bullet = \mathsf{snd}\ x_{fresh}\ \mathsf{in}\ {e'}^+ \end{array}$$

Figure 8. Selected cases for MiniML and Affi compilers.

$$\frac{}{C_{\mathtt{int}\mapsto\mathbf{bool}}, C_{\mathbf{bool}\mapsto\mathtt{int}} : \mathtt{int} \sim \mathbf{bool}}$$

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

$$C_{\mathbf{bool}\mapsto\mathtt{int}}(e) \triangleq e \qquad C_{\mathtt{int}\mapsto\mathbf{bool}}(e) \triangleq \mathsf{if}\ e\ 0\ 1$$

$$\begin{array}{l} C_{\tau_1\multimap\tau_2\mapsto(\mathtt{unit} \to \tau_1) \to \tau_2}(e) \triangleq \\ \quad \mathsf{let}\ x = e\ \mathsf{in}\ \lambda x_{thnk}.\mathsf{let}\ x_{conv} = C_{\tau_1\mapsto\tau_1}(x_{thnk}\ ())\ \mathsf{in} \\ \qquad\qquad\qquad \mathsf{let}\ x_{acc} = \mathsf{thunk}(x_{conv})\ \mathsf{in}\ C_{\tau_2\mapsto\tau_2}(x\ x_{acc}) \\ C_{(\mathtt{unit} \to \tau_1) \to \tau_2\mapsto\tau_1\multimap\tau_2}(e) \triangleq \mathsf{let}\ x = e\ \mathsf{in} \\ \quad \lambda x_{thnk}.\mathsf{let}\ x_{acc} = \mathsf{thunk}(C_{\tau_1\mapsto\tau_1}(x_{thnk}\ ()))\ \mathsf{in}\ C_{\tau_2\mapsto\tau_2}(x\ x_{acc}) \end{array}$$

Figure 9. Selected convertibility rules for MiniML and Affi.

and $\tau_1 \otimes \tau_2 \sim \tau_1 \times \tau_2$). We focus on the conversion between $\to$ and $\multimap$ (note, of course, that it is impossible to safely convert $\multimap\!\bullet$ to MiniML). Our compiler is designed to support affine code being mixed directly with unrestricted code. Intuitively, an affine function should be able to behave as an unrestricted one, but the other direction is harder to accomplish, and higher-order functions mean both must be addressed at once. In order to account for this, we convert $\tau_1 \multimap \tau_2$ not to $\tau_1 \to \tau_2$, but rather to $(\mathtt{unit} \to \tau_1) \to \tau_2$. That is, to a MiniML function that expects its argument to be a thunk containing a τ_1 rather than a τ_1 directly. Provided that the thunk fails if invoked more than once, we can ensure, dynamically, that a MiniML function with that type behaves as an Affi function of a related type. These invariants are ensured by appropriate wrapping and use of the compiler macro thunk(·) (see top of Fig. 8).

Semantic Model The most interesting part of this case study is the logical relation because we must build a model that allows us to show that the dynamic and static affine bindings within Affi are used at most once. For a dynamic binding, this is tracked in target code by the dynamic reference flag created by thunk. For a static binding, we use a similar strategy of tracking use via a flag, but rather than a target-level dynamic runtime flag, we create a *phantom* flag that exists only within our model. Specifically, we define an augmented target operational semantics that exists solely for the model, and any program that runs without getting stuck under the augmented semantics has a trivial erasure to a program that runs under the standard semantics. This means we are using the model to identify a subset of target programs (the erasures of well-behaved augmented programs) that behave sensibly and do not violate source type constraints (i.e., do not use static variables more than once), even if there is nothing in the target programs that actually witnesses those constraints (i.e., dynamic checks or static types).

We build the model as follows. First, we extend our machine configurations to keep track of *phantom flags* f — i.e., in addition to a heap H and term e, we have a *phantom flag set* Φ. Second, the augmented semantics uses one additional term, protect, which consumes one of the aforementioned phantom flags when it reduces:

$$\text{Expressions } e ::= \ldots \mathsf{protect}(e, f)$$

$$\langle \Phi \uplus \{f\}, H, \mathsf{protect}(e, f) \rangle \dashrightarrow \langle \Phi, H, e \rangle$$

And finally, we modify the two rules that introduce bindings such that whenever a binding in the syntactic category $\bullet$ is introduced, we create a new phantom flag (where "f fresh" means f is disjoint from all the flags generated thus far during this execution):

$$\frac{f \text{ fresh}}{\langle \Phi, H, \mathsf{let}\ a_\bullet = v\ \mathsf{in}\ e \rangle \dashrightarrow \langle \Phi \uplus \{f\}, H, [a_\bullet \mapsto \mathsf{protect}(v, f)]e \rangle}$$

$$\frac{f \text{ fresh}}{\langle \Phi, H, \lambda a_\bullet.e\ v \rangle \dashrightarrow \langle \Phi \uplus \{f\}, H, [a_\bullet \mapsto \mathsf{protect}(v, f)]e \rangle}$$

Note that we write $\dashrightarrow$ for a step in this augmented semantics, to distinguish it from the true operational step $\longrightarrow$. While phantom flags in the augmented operational semantics play a similar role in protecting static affine resources as dynamic reference flags in the dynamic case, the critical difference is that in the augmented semantics, a protect(·)ed resource for which there is no phantom flag will get stuck, and thus be excluded from the logical relation by construction. This is very different from the dynamic case, where we want — and, in fact, need — to include terms that can fail in order to mix MiniML and Affi without imposing an affine type system on MiniML itself. What this means for the model is that dynamic reference flags are a *shared resource* that can be accessed from many parts of the program and therefore tracked in the world, while phantom flags are an *unique resource* which our type system ensures is owned/used by at most one part of the program, which is what allows us to prove that the augmented semantics will not get stuck.

While the full definitions are in our supplementary materials, we give a high-level description of our expression and value relations, shown in Fig. 10, noting that the high-level structure is similar to the first case study.

Our expression relation, $\mathcal{E}[\![\tau]\!]_\rho$, is made up of tuples with worlds W and pairs of phantom flag stores and terms (Φ_i, e_i), where each flag store represents the phantom variables owned by the expression. Our worlds W keep the step index, a standard heap typing Ψ (see §2), but also an affine flag store Θ, which maps pairs of dynamic flags (ℓ_1, ℓ_2) to *either* a marker that indicates a dynamic affine variable has been used (0, written USED), *or* the phantom flags (Φ_1, Φ_2) that it closes over if it has not been used. These dynamic flags ℓ_1, ℓ_2 are a subset of the heap, disjoint from Ψ (which tracks the rest of the heap, i.e., all the normal/non-dynamic-flag references). The expression relation then says that, given heaps that satisfy the world and arbitrary "rest" of phantom flag stores Φ_{r1} and Φ_{r2} (disjoint from those closed over by the world and the owned portions), the term e_1 will either: (i) run longer than the step index accounts for, (ii) fail CONV (error while converting a value), or (iii) terminate at some value e_2, where the flag store Φ_1 has been modified to $\Phi_{f1} \uplus \Phi_{g1}$, the heap has changed to H_1', and the new world W' is an extension of W. World extension ($\sqsubseteq_{\Phi_{r1},\Phi_{r2}}$) is defined over worlds that do not contain phantom flags from Φ_{r1}, Φ_{r2}. It allows the step index to decrease, the heap typing to gain (but not overwrite or remove) entries, and the affine store to mark (but not unmark) bindings as USED.

At that future world, we know that the other side will have similarly run to a value with modified heap, flag stores, and we know that the resulting values, along with their Φ_{f1} and Φ_{f2}, will be in the value relation $\mathcal{V}[\![\tau]\!]_\rho$. The phantom flag stores Φ_{gi} are "garbage" that are no longer needed, and the "rest" is unchanged. Note that, while running, some phantom flags may have moved into the world, which has changed, but the world cannot have absorbed what was in the "rest".

Our value relation cases are now mostly standard, so we will focus only on the interesting ones: $\multimap$ and $-\!\!\bullet$. $\mathcal{V}[\![\tau_1 \multimap \tau_2]\!]_\cdot$ is defined to take arbitrary arguments from $\mathcal{V}[\![\tau_1]\!]_\cdot$, which may own static phantom flags in Φ_1 and Φ_2, and add both new locations ℓ_1, ℓ_2 that will be used in the thunks that prevent multiple uses, but also store the phantom flags in the affine store. The idea is that a function $\lambda \mathbf{a}_\circ : \_.\mathbf{e}$ can be applied to an expression that closes over static phantom flags, like **let** $(\mathbf{b}_\bullet, \mathbf{c}_\bullet) = (1, 2)$ **in** $\lambda \mathbf{a}_\bullet.\mathbf{b}_\bullet$—the latter will have phantom flags for both $\mathbf{b}_\bullet$ and $\mathbf{c}_\bullet$. The bodies are then run with the argument substituted with guarded expressions. Now, consider what happens when the variable is used: the guard$(\cdot)$ wrapper will update the location to USED, which means that in the world, the phantom flags that were put at that location are no longer there — i.e., they are no longer returned by flags(W'), which returns all phantom flags closed over by dynamic flags. That means, for the reduction to be well-formed, the phantom flags have to move somewhere else—either back to being owned by the term (in Φ_{fi}) or in the discarded "garbage" Φ_{gi}. Once the phantom flag set has been moved back out of the world, the flags can again be used by protect$(\cdot)$ expressions.

The static function, $\mathcal{V}[\![\tau_1 -\!\!\bullet\, \tau_2]\!]_\cdot$, has a similar flavor, but it may itself own static phantom flags. That means that the phantom flag sets for the arguments must be disjoint, and when we run the bodies, we combine the two sets along with a fresh pair of phantom flags f_1, f_2 for the argument, which are then put inside the protect$(\cdot)$ expressions.

$$
\begin{array}{l}
\mathrm{guard}(\mathrm{e}, \ell) \triangleq \lambda\_.\{\mathrm{if}\ !\ell\ \{\mathrm{fail\ C{\small ONV}}\}\ \{\ell := \mathrm{\small USED}; \mathrm{e}\}\} \\
\mathcal{V}[\![\tau_1 \rightarrow \tau_2]\!]_\rho = \{(W, (\emptyset, \lambda \mathrm{x}.\{\mathrm{e}_1\}), (\emptyset, \lambda \mathrm{x}.\{\mathrm{e}_2\})) \mid \forall \mathrm{v}_1\ \mathrm{v}_2\ W'. \\
\qquad W \sqsubset_{\emptyset,\emptyset} W' \ \wedge\ (W', (\emptyset, \mathrm{v}_1), (\emptyset, \mathrm{v}_2)) \in \mathcal{V}[\![\tau_1]\!]_\rho \\
\qquad \implies (W', (\emptyset, [x {\mapsto} \mathrm{v}_1]\mathrm{e}_1), (\emptyset, [x {\mapsto} \mathrm{v}_2]\mathrm{e}_2)) \in \mathcal{E}[\![\tau_2]\!]_\rho\} \\
\mathcal{V}[\![\tau_1 \multimap \tau_2]\!]_\cdot = \{(W, (\emptyset, \lambda\, \mathrm{x}\{\mathrm{e}_1\}), (\emptyset, \lambda\, \mathrm{x}\{\mathrm{e}_2\})) \mid \forall \Phi_1\ \mathrm{v}_1\ \Phi_2\ \mathrm{v}_2\ W'. \\
\qquad W \sqsubset_{\emptyset,\emptyset} W' \wedge (W', (\Phi_1, \mathrm{v}_1), (\Phi_2, \mathrm{v}_2)) \in \mathcal{V}[\![\tau_1]\!]_\cdot \\
\qquad \implies ((W'.k, W'.\Psi, W'.\Theta \uplus (\ell_1, \ell_2) \mapsto (\Phi_1, \Phi_2)), \\
\qquad\qquad (\emptyset, [x {\mapsto} \mathrm{guard}(\mathrm{v}_1, \ell_1)]\mathrm{e}_1), \\
\qquad\qquad (\emptyset, [x {\mapsto} \mathrm{guard}(\mathrm{v}_2, \ell_2)]\mathrm{e}_2)) \in \mathcal{E}[\![\tau_2]\!]_\cdot\} \\
\mathcal{V}[\![\tau_1 -\!\!\bullet\, \tau_2]\!]_\cdot = \{(W, (\Phi_1, \lambda\, \mathrm{a}_\bullet.\{\mathrm{e}_1\}), (\Phi_2, \lambda\, \mathrm{a}_\bullet.\{\mathrm{e}_2\})) \mid \\
\qquad \forall \Phi_1'\ \Phi_2'\ f_1\ f_2\ \mathrm{v}_1\ \mathrm{v}_2\ W'. W \sqsubset_{\Phi_1,\Phi_2} W' \\
\qquad \wedge\ (W', (\Phi_1', \mathrm{v}_1), (\Phi_2', \mathrm{v}_2)) \in \mathcal{V}[\![\tau_1]\!]_\cdot \\
\qquad \wedge \Phi_1 \cap \Phi_1' = \Phi_2 \cap \Phi_2' = \emptyset \\
\qquad \wedge\ f_1 \notin \Phi_1 \uplus \Phi_1' \uplus \mathrm{flags}(W', 1) \\
\qquad \wedge f_2 \notin \Phi_2 \uplus \Phi_2' \uplus \mathrm{flags}(W', 2) \\
\qquad \implies (W', (\Phi_1 \uplus \Phi_1' \uplus \{f_1\}, [\mathrm{a}_\bullet {\mapsto} \mathrm{protect}(\mathrm{v}_1, f_1)]\mathrm{e}_1), \\
\qquad\qquad (\Phi_2 \uplus \Phi_2' \uplus \{f_2\}, [\mathrm{a}_\bullet {\mapsto} \mathrm{protect}(\mathrm{v}_2, f_2)]\mathrm{e}_2)) \\
\qquad\qquad \in \mathcal{E}[\![\tau_2]\!]_\cdot\} \\
\mathcal{E}[\![\tau]\!]_\rho = \{(W, (\Phi_1, \mathrm{e}_1), (\Phi_2, \mathrm{e}_2)) \mid \mathrm{freevars}(\mathrm{e}_1) = \mathrm{freevars}(\mathrm{e}_2) = \emptyset\ \wedge \\
\quad \forall \Phi_{r1}, \Phi_{r2}, \mathrm{H}_1, \mathrm{H}_2{:}W,\ \mathrm{e}_1',\ \mathrm{H}_1',\ j < W.k.\ \Phi_{r1} \# \Phi_1 \\
\quad \wedge \Phi_{r2} \# \Phi_2 \wedge \Phi_{r1} \uplus \Phi_1, \Phi_{r2} \uplus \Phi_2 : W \wedge \\
\quad \langle \Phi_{r1} \uplus \mathrm{flags}(W, 1) \uplus \Phi_1, \mathrm{H}_1, \mathrm{e}_1 \rangle \overset{j}{\dashrightarrow} \langle \Phi_1', \mathrm{H}_1', \mathrm{e}_1' \rangle \nrightarrow \\
\quad \implies \mathrm{e}_1' = \mathrm{fail\ C{\small ONV}} \vee (\exists \Phi_{f1}\ \Phi_{g1}\ \Phi_{f2}\ \Phi_{g2}\ \mathrm{v}_2 \mathrm{H}_2' W'. \\
\qquad \langle \Phi_{r2} \uplus \mathrm{flags}(W, 2) \uplus \Phi_2, \mathrm{H}_2, \mathrm{e}_2 \rangle \\
\qquad \overset{*}{\dashrightarrow} \langle \Phi_{r2} \uplus \mathrm{flags}(W', 2) \uplus \Phi_{f2} \uplus \Phi_{g2}, \mathrm{H}_2', \mathrm{v}_2 \rangle \nrightarrow \\
\qquad \wedge\ \Phi_1' = \Phi_{r1} \uplus \mathrm{flags}(W', 1) \uplus \Phi_{f1} \uplus \Phi_{g1} \wedge \\
\qquad \wedge\ W \sqsubseteq_{\Phi_{r1},\Phi_{r2}} W' \wedge\ \mathrm{H}_1', \mathrm{H}_2' : W' \\
\qquad \wedge\ (W', (\Phi_{f1}, \mathrm{e}_1'), (\Phi_{f2}, \mathrm{v}_2)) \in \mathcal{V}[\![\tau]\!]_\rho)\}
\end{array}
$$

Figure 10. Selections of MiniML & AFFI Logical Relation.

With the logical relation in hand, we can prove analogous theorems to Lemma 2.1 (Convertibility Soundness), Theorem 2.2 (Fundamental Property), Theorem 2.3 (Type Safety for Lang A), and Theorem 2.4 (Type Safety for Lang B).

Note that to prove our type safety theorems, we prove a lemma which states that, if $\langle \mathrm{H}, \mathrm{e} \rangle \overset{*}{\rightarrow} \langle \mathrm{H}', \mathrm{e}' \rangle \nrightarrow$, then for any Φ, $\langle \Phi, \mathrm{H}, \mathrm{e} \rangle \overset{*}{\dashrightarrow} \langle \Phi_1', \mathrm{H}_1', \mathrm{e}_1' \rangle \nrightarrow$. This lemma is necessary because the given assumption of the type safety theorem is that the configuration $\langle \mathrm{H}, \mathrm{e} \rangle$ steps under the normal operational semantics, but to apply the expression relation, we need that a corresponding configuration steps to an irreducible configuration under the phantom operational semantics.

Although our phantom flag realizability model was largely motivated by efficiency concerns with the dynamic enforcement of affinity, more broadly, it demonstrates how one can build complex static reasoning into the model even if such reasoning is absent from the target. Indeed, the actual target language, which source programs are compiled to and run in, has not changed; the augmentations exist *only in the model*. In this way, the preservation of source invariants is subtle: it is not that the types actually exist in the target (via runtime invariants or actual target types), but rather that the operational behavior of the target is exactly what the type interpretations characterize.

4 Memory Management & Polymorphism

For our third case study, we consider how MiniML, whose references are garbage collected, can interoperate with core $\mathbf{L^3}$, a language with safe strong updates despite memory aliasing, supported via linear capabilities [3]. This case study primarily highlights how different memory management strategies can interoperate safely, in particular, that manually managed linear references can be converted to garbage-collected references without copying. This is of particular interest as more low-level code is written in Rust, a language with an ownership discipline on memory that similarly could allow safe transfer of memory to garbage-collected languages.

We also use this case study to explore how polymorphism/generics in one language can be used, via a form of interoperability, from the other. This is interesting because significant effort has gone into adding generics to languages that did not originally support them, in order to more easily build certain re-usable libraries.[3] While we are not claiming that interoperability could entirely replace built-in polymorphism, sound support for cross-language type instantiation and polymorphic libraries presents a possible alternative, especially for smaller, perhaps more special-purpose, languages. This would allow us to write something like:

$$\mathtt{map}(\!|(\lambda \mathbf{x} : \mathbf{int}.\mathbf{x} + \mathbf{1})|\!)_{\langle \mathbf{int}\rangle \to \langle \mathbf{int}\rangle}\,(\!|[\mathbf{1}, \mathbf{2}, \mathbf{3}]|\!)_{\mathtt{list}\ \langle \mathbf{int}\rangle}$$

where the blue language supports polymorphism, and has a generic map function, while the **pink** language does not. Of course, since convertibility is still driving this, in addition to using a concrete **intlist**, $\mathbf{[1, 2, 3]}$, as above, the language without polymorphism could convert entirely different (non-list) concrete representations into similar polymorphic ones — i.e., implementing a sort of polymorphic interface at the boundary. For example, rather than an **intlist** (or a **stringlist**), in the example above, one could start with an **intarray** or **intbtree**, or any number of other traversable data structures that could be converted to `list int` (or any `list` α).

Languages We present the syntax of $\mathbf{L^3}$, augmented with forms for interoperability, in Fig. 11. $\mathbf{L^3}$ has linear capability types $\mathbf{cap}\ \zeta\ \tau$ (capability for abstract location ζ storing data of type τ), unrestricted pointer types $\mathbf{ptr}\ \zeta$ to support aliasing, and location abstraction ($\Lambda\zeta.\mathbf{e} : \forall\zeta.\tau$ and $\ulcorner\zeta, \mathbf{v}\urcorner : \exists\zeta.\tau$). The key insight to $\mathbf{L^3}$ is that the pointer can be separated from the capability and passed around in the program separately. At runtime, the capabilities will be erased, but the static discipline only allows pointers to be used with their capabilities (tied together with the type variables ζ), and only allows capabilities to be used linearly. This enables safe in-place updates and low-level manual memory management while still supporting some flexibility in terms of pointer manipulation. We refer the reader to our supplementary materials, or the original paper on $\mathbf{L^3}$ ([3]) for more details on its precise static semantics, but present highlights here. In particular, **new** allocates memory and returns an existential package containing a capability and pointer ($\exists\zeta.\mathbf{cap}\ \zeta\ \tau \otimes \mathbf{ptr}\ \zeta$). **swap** takes a matching capability ($\mathbf{cap}\ \zeta\ \tau_1$) and pointer $\mathbf{ptr}\ \zeta$ and a value (of a possibly different type τ_2) and replaces what is stored, returning the capability and old value $\mathbf{cap}\ \zeta\ \tau_2 \otimes \tau_1$. Note that since capabilities record the type of what is in the heap and are unique, strong updates are safe. Finally, **free** takes a package of a capability and pointer ($\exists\zeta.\mathbf{cap}\ \zeta\ \tau \otimes \mathbf{ptr}\ \zeta$) and frees the memory, consuming both in the process and returning what was stored there—any lingering pointers are harmless, as the necessary capability is now gone.

We compile both $\mathbf{L^3}$ and MiniML to an extension of the Scheme-like target LCVM that we used in the previous case study (see Fig. 13 for $\mathbf{L^3}$; MiniML is standard). Our additions to LCVM, shown in Fig. 12, add manual memory allocation (alloc), free (which will error on a garbage-collected location), an instruction (gcmov) to convert a manually managed location to garbage collected, and an instruction (callgc) to explicitly invoke the garbage collector. The last allows the compiler to decide where the GC can intercede (before allocation, in our compiler), and in doing so simplifies our model slightly. The memory management itself is captured in our heap definition, which allows the same location names to be used as either GC'd ($\overset{gc}{\mapsto}$) or manually managed ($\overset{m}{\mapsto}$), and reused after garbage collection or manual free. Dereference (!e) and assignment (e := e) work on both types of reference (failing, of course, if it is manually managed and has been freed). This strategy of explicitly invoking the garbage collector and using a single pool of locations retains significant challenging aspects about garbage collectors while remaining simple enough to expose the interesting aspects of interoperation.

As in the previous case study, we have boundary terms, $(\!|\mathbf{e}|\!)_\tau$ and $(\!|\mathtt{e}|\!)_{\boldsymbol{\tau}}$, for *converting* a term and using it in the other language. Now, we also add new types $\langle\boldsymbol{\tau}\rangle$, pronounced "foreign type", and allow conversions from $\boldsymbol{\tau}$ to $\langle\boldsymbol{\tau}\rangle$ for *opaquely embedding*[4] types for use in polymorphic functions.

If a language supports polymorphism, then its type abstractions should be agnostic to the types that instantiate them, allowing them to range over not only host types, but indeed any foreign types as well. Doing so should not violate parametricity. However, the non-polymorphic language may need to make restrictions on how this power can be used, so as to not allow the polymorphic language to violate

[3] e.g., Java 1.5/5, C# 2.0 [27] and more recently, in the Go programming language

[4] Similar to "lumps" in Matthews-Findler[32], though they give a *single* lump type for all foreign types, i.e., they would have only $\langle\rangle$, rather than $\langle\tau\rangle$.

$$
\begin{array}{lcl}
\mathbf{L^3} & & \\
\text{Type } \tau & ::= & \mathbf{unit} \mid \mathbf{bool} \mid \tau \otimes \tau \mid \tau \multimap \tau \mid !\tau \\
 & & \mid \mathbf{ptr}\ \zeta \mid \mathbf{cap}\ \zeta\ \tau \mid \forall \zeta.\tau \mid \exists \zeta.\tau \\
\text{Value } \mathbf{v} & ::= & \lambda \mathbf{x} : \tau.\mathbf{e} \mid () \mid \mathbb{B} \mid (\mathbf{v}, \mathbf{v}) \mid !\mathbf{v} \mid \Lambda \zeta.\mathbf{e} \mid \ulcorner \zeta,\ \mathbf{v} \urcorner \\
\text{Expr. } \mathbf{e} & ::= & \mathbf{v} \mid \mathbf{x} \mid (\mathbf{e},\ \mathbf{e}) \mid \mathbf{e}\ \mathbf{e} \mid \mathbf{let}\ () = \mathbf{e}\ \mathbf{in}\ \mathbf{e} \mid \mathbf{if}\ \mathbf{e}\ \mathbf{e}\ \mathbf{e} \\
 & & \mid \mathbf{let}\ (\mathbf{x},\ \mathbf{x}) = \mathbf{e}\ \mathbf{in}\ \mathbf{e} \mid \mathbf{let}\ !\mathbf{x} = \mathbf{e}\ \mathbf{in}\ \mathbf{e} \mid \mathbf{dupl}\ \mathbf{e} \\
 & & \mid \mathbf{drop}\ \mathbf{e} \mid \mathbf{new}\ \mathbf{e} \mid \mathbf{free}\ \mathbf{e} \mid \mathbf{swap}\ \mathbf{e}\ \mathbf{e}\ \mathbf{e} \mid \mathbf{e}\ [\zeta] \\
 & & \mid \ulcorner \zeta,\ \mathbf{e} \urcorner \mid \mathbf{let}\ \ulcorner \zeta,\ \mathbf{x} \urcorner = \mathbf{e}\ \mathbf{in}\ \mathbf{e} \mid (\!|\mathtt{e}|\!)_\tau \mid \langle \mathtt{e} \rangle_\tau \\
\textsc{Duplicable} & = & \{\mathbf{unit}, \mathbf{bool}, \mathbf{ptr}\ \zeta, !\tau\}
\end{array}
$$

Figure 11. Syntax for $\mathbf{L^3}$.

$$
\begin{array}{lcl}
\text{Expr e} & ::= & \ldots \mid \text{alloc e} \mid \text{free e} \mid \text{gcmov e} \mid \text{callgc} \\
\text{Heap H} & ::= & \ell \overset{m}{\mapsto} v, \text{H} \mid \ell \overset{gc}{\mapsto} v, \text{H} \mid \cdot \\
\text{Err Code c} & ::= & \ldots \mid \textsc{Ptr}
\end{array}
$$

Figure 12. Additions to LCVM (see Fig. 6 for base LCVM).

$$
\mathbf{x} \rightsquigarrow \text{x} \quad () \rightsquigarrow () \quad \mathbf{true/false} \rightsquigarrow 0/1 \quad !\mathbf{v} \rightsquigarrow \mathbf{v}^+ \quad \lambda \mathbf{x} : \tau.\mathbf{e} \rightsquigarrow \lambda \text{x}.\mathbf{e}^+
$$

$$
\begin{array}{lcl}
\mathbf{e_1 e_2} & \rightsquigarrow & \mathbf{e_1}^+ \mathbf{e_2}^+ \\
\mathbf{let}\ () = \mathbf{e_1}\ \mathbf{in}\ \mathbf{e_2} & \rightsquigarrow & \text{let \_} = \mathbf{e_1}^+ \text{ in } \mathbf{e_2}^+ \\
\mathbf{if}\ \mathbf{e_1}\ \mathbf{e_2}\ \mathbf{e_3} & \rightsquigarrow & \text{if } \mathbf{e_1}^+\ \mathbf{e_2}^+\ \mathbf{e_3}^+ \\
(\mathbf{e_1},\ \mathbf{e_2}) & \rightsquigarrow & (\mathbf{e_1}^+,\ \mathbf{e_2}^+) \\
\mathbf{let}\ (\mathbf{x_1},\ \mathbf{x_2}) = \mathbf{e_1}\ \mathbf{in}\ \mathbf{e_2} & \rightsquigarrow & \text{let p} = \mathbf{e_1}^+ \text{ in let } \text{x}_1 = \text{fst p in} \\
 & & \text{let } \text{x}_2 = \text{snd p in } \mathbf{e_2}^+ \\
\mathbf{let}\ !\mathbf{x} = \mathbf{e_1}\ \mathbf{in}\ \mathbf{e_2} & \rightsquigarrow & \text{let x} = \mathbf{e_1}^+ \text{ in } \mathbf{e_2}^+ \\
\mathbf{dupl}\ \mathbf{e} & \rightsquigarrow & \text{let x} = \mathbf{e}^+ \text{ in (x, x)} \\
\mathbf{drop}\ \mathbf{e} & \rightsquigarrow & \text{let \_} = \mathbf{e}^+ \text{ in ()} \\
\mathbf{new}\ \mathbf{e} & \rightsquigarrow & \text{let \_} = \text{callgc in let } \text{x}_\ell = \text{alloc } \mathbf{e}^+ \text{ in} \\
 & & ((),\ \text{x}_\ell) \\
\mathbf{free}\ \mathbf{e} & \rightsquigarrow & \text{let x} = \mathbf{e}^+ \text{ in let } \text{x}_\text{r} = !(\text{snd x}) \text{ in} \\
 & & \text{let \_} = \text{free (snd x) in } \text{x}_\text{r} \\
\mathbf{swap}\ \mathbf{e_c}\ \mathbf{e_p}\ \mathbf{e_v} & \rightsquigarrow & \text{let } \text{x}_\text{p} = \mathbf{e_p}^+ \text{ in let \_} = \mathbf{e_c} \text{ in let } \text{x}_\text{v} = !\text{x}_\text{p} \\
 & & \text{in let \_} = (\text{x}_\text{p} := \mathbf{e_v}+) \text{ in } ((), \text{x}_\text{v}) \\
\Lambda \zeta.\mathbf{e} & \rightsquigarrow & \lambda\_.\mathbf{e}^+ \\
\mathbf{e}\ [\zeta] & \rightsquigarrow & \mathbf{e}^+\ () \\
\ulcorner \zeta,\ \mathbf{e} \urcorner & \rightsquigarrow & \mathbf{e}^+ \\
\mathbf{let}\ \ulcorner \zeta,\ \mathbf{x} \urcorner = \mathbf{e_1}\ \mathbf{in}\ \mathbf{e_2} & \rightsquigarrow & \text{let x} = \mathbf{e_1}^+ \text{ in } \mathbf{e_2}^+ \\
(\!|\mathtt{e}|\!)_\tau & \rightsquigarrow & \text{C}_{\tau \mapsto \tau}(\mathtt{e}^+)
\end{array}
$$

Figure 13. Compiler for $\mathbf{L^3}$.

its invariants. To make this challenge material, our non-polymorphic language in this case study has linear resources (heap capabilities) that cannot, if we are to maintain soundness, be duplicated. This means, in particular, that whatever interoperability strategy we come up with cannot allow a linear capability from $\mathbf{L^3}$ to flow over to a `MiniML` function that duplicates it, even if such function is well-typed (and parametric) in `MiniML`.

Convertibility The first conversion that we want to highlight is between references. In $\mathbf{L^3}$, pointers have capabilities that convey ownership, and thus to convert a pointer we also need the corresponding capability. For brevity, we may use $\mathbf{REF}\ \tau$ to abbreviate a capability+pointer package type.

$$
\frac{\text{C}_{\tau \mapsto \tau}, \text{C}_{\tau \mapsto \tau} : \tau \sim \tau}{\text{C}_{\mathbf{REF}\ \tau \mapsto \mathtt{ref}\ \tau}, \text{C}_{\mathtt{ref}\ \tau \mapsto \mathbf{REF}\ \tau} : \mathtt{ref}\ \tau \sim \exists \zeta.\mathbf{cap}\ \zeta\ \tau \otimes !\mathbf{ptr}\ \zeta}
$$

$$
\begin{array}{rl}
\text{C}_{\mathbf{REF}\ \tau \mapsto \mathtt{ref}\ \tau}(\text{e}) \triangleq & \text{let x} = \text{snd e in} \\
 & \text{let \_} = (\text{x} := \text{C}_{\tau \mapsto \tau}(!\text{x})) \text{ in gcmov x} \\
\text{C}_{\mathtt{ref}\ \tau \mapsto \mathbf{REF}\ \tau}(\text{e}) \triangleq & \text{let x} = \text{alloc } \text{C}_{\tau \mapsto \tau}(!\text{e}) \text{ in } ((), \text{x})
\end{array}
$$

The glue code itself is quite interesting: going from $\mathbf{L^3}$ to `MiniML`, since the $\mathbf{L^3}$ type system guarantees that this is the only capability to this pointer, we can safely directly convert the pointer into a `MiniML` pointer with gcmov after in-place replacing the contents with the result of converting (a less general rule that had a different premise might not need to do this, e.g., if the data was already compatible—see the first case study for more details). Going the other direction, from `MiniML` to $\mathbf{L^3}$, there is no way for us to know if there are other aliases to the reference, so we can't convert the pointer: instead we copy and convert data into a freshly allocated manually managed location (note how at runtime capabilities are erased to unit).

We account for interoperability of polymorphism in two parts. First, we have a *foreign type*, $\langle \tau \rangle$, which embeds an $\mathbf{L^3}$ type into the type grammar of `MiniML`. This foreign type, like any `MiniML` type, can be used to instantiate type abstractions, define functions, etc, but `MiniML` has no introduction or elimination rules for it—terms of foreign type must come across from, and then be sent back to, $\mathbf{L^3}$. These come by way of the conversion rule $\langle \tau \rangle \sim \tau$, which allow terms of the form $(\!|\mathtt{e}|\!)_{\langle \tau \rangle}$ (to bring an $\mathbf{L^3}$ term to `MiniML`) and $(\!|\mathtt{e}|\!)_\tau$ (the reverse). Moreover, the conversion rule for foreign types restricts τ to a safe $\textsc{Duplicable}$ subset of types, but has no runtime consequences:

$$
\frac{\tau \in \textsc{Duplicable}}{\text{C}_{\langle \tau \rangle \mapsto \tau}, \text{C}_{\tau \mapsto \langle \tau \rangle} : \langle \tau \rangle \sim \tau} \qquad
\begin{array}{l}
\text{C}_{\langle \tau \rangle \mapsto \tau}(\text{e}) \triangleq \text{e} \\
\text{C}_{\tau \mapsto \langle \tau \rangle}(\text{e}) \triangleq \text{e}
\end{array}
$$

To prove soundness we need to show that $\textsc{Duplicable}$ types are indeed safe to embed. The soundness condition depends on the expressive power of the two languages when viewed through the lens of polymorphism. In our case, since the non-polymorphic language is linear but the polymorphic one is not, we need to show is that a $\textsc{Duplicable}$ type can be copied (i.e., none of its values own linear capabilities)—this includes $\mathbf{unit}$ and $\mathbf{bool}$, but also $\mathbf{ptr}\ \zeta$ and any type of the form $!\tau$. Now, consider examples using this:

$$
(\Lambda\alpha.\lambda \mathtt{x}{:}\alpha.\lambda \mathtt{y}{:}\alpha.\mathtt{y})[\langle \mathbf{bool} \rangle]\ (\!|\mathbf{true}|\!)_{\langle \mathbf{bool} \rangle}\ (\!|\mathbf{false}|\!)_{\langle \mathbf{bool} \rangle} \qquad (1)
$$

$$
(\lambda \mathtt{x} : \mathtt{BOOL}.\mathtt{x})(\!|\mathbf{true}|\!)_{\mathtt{BOOL}} \text{ where } \mathtt{BOOL} \triangleq \forall \alpha.\alpha \to \alpha \to \alpha \qquad (2)
$$

In (1), the leftmost expression is a polymorphic `MiniML` function that returns the second of its two arguments. It is instantiated it with a foreign type, $\langle \mathbf{bool} \rangle$. Next, two terms of type $\mathbf{bool}$ in $\mathbf{L^3}$ are embedded via the foreign conversion, $(\!|\cdot|\!)_{\langle \mathbf{bool} \rangle}$, which requires that $\mathbf{bool} \in \textsc{Duplicable}$. Not only does this mechanism allow $\mathbf{L}^3$ programmers to use polymorphic functions, but also `MiniML` programmers to use new base types. Of course, we could also convert the actual values, as in (2). To do so, we can define conversions between Church booleans in `MiniML` (which has no booleans) and ordinary booleans in $\mathbf{L}^3$:

$$\frac{}{\forall\alpha.\alpha \rightarrow \alpha \rightarrow \alpha \sim \mathbf{bool}} \qquad \begin{array}{l} \mathsf{C}_{\mathsf{BOOL}\mapsto\mathbf{bool}}(\mathsf{e}) \triangleq \mathsf{e}\ ()\ 0\ 1 \\ \mathsf{C}_{\mathbf{bool}\mapsto\mathsf{BOOL}}(\mathsf{e}) \triangleq \mathsf{if0}\ \mathsf{e}\ \{\Lambda\alpha.\lambda\mathsf{x}{:}\alpha.\lambda\mathsf{y}{:}\alpha.\mathsf{x}\} \\ \qquad\qquad\qquad\qquad \{\Lambda\alpha.\lambda\mathsf{x}{:}\alpha.\lambda\mathsf{y}{:}\alpha.\mathsf{y}\} \end{array}$$

Semantic Model In Fig. 14, we present parts of the binary logical relation that we use to prove our conversions and entire languages sound (see supplementary material).

Our model is inspired by that of core $\mathbf{L}^3$ [3], though ours is binary and significantly more complex to account for garbage collection and interoperation with `MiniML`. The key is a careful distinction between owned (linear) manual memory, which is *local* and described by heap fragments associated with terms, and garbage-collected memory, which is *global* and described by the world W. Since memory can be freed (via garbage collection or manual free), reused, and moved from manual memory to garbage-collected memory, there are several constraints on how heap fragments and worlds may evolve so we can ensure safe memory usage.

With that in mind, our value interpretation of source types $\mathcal{V}[\![\tau]\!]_\rho$ are sets of worlds and related heap-fragment-and-value pairs $(\mathsf{H}_1, \mathsf{v}_1)$, $(\mathsf{H}_2, \mathsf{v}_2)$ where the heap fragment H_i paired with value v_i is the portion of the manually managed heap *owned* by that value.

The relational substitution ρ maps type variables α to arbitrary type interpretations R and location variables ζ to concrete locations (ℓ_1, ℓ_2). Since `MiniML` cannot own manual (linear) memory, all cases $\mathcal{V}[\![\tau]\!]_\rho$ have empty $\emptyset$ heap fragments. However, during evaluation, memory could be allocated and subsequently freed so the expression relation does not have that restriction. In $\mathbf{L}^3$, pointer types $\mathbf{ptr}\ \zeta$ do not own locations, so they can be freely copied. Rather, linear capabilities $\mathbf{cap}\ \zeta\ \tau$ convey ownership of the location ℓ_i that ζ maps to and the heap fragment H_i pointed to by ℓ_i.

In the expression relation $\mathcal{E}[\![\tau]\!]_\rho$, we run the expressions with sets of pinned locations (L_i) that the garbage collector should not touch (which may come from an outer context if we are evaluating a subterm), garbage-collected heap fragments that satisfy the world (H_{ig+}), arbitrary disjoint manually allocated $(MHeap)$ "rest" of the heap (H_{ir}), composed with the owned fragment (H_i). Then, assuming e_1 terminates, we expect the "rest" heap is unchanged, the garbage-collected portion has been transformed to H'_{ig}, the owned portion has been transformed into H'_1, and that e_2 terminates in an analogous configuration, so $(W', (\mathsf{H}'_1, \mathsf{v}_1), (\mathsf{H}'_2, \mathsf{v}_2)) \in \mathcal{V}[\![\tau]\!]_\rho$, where W' is a world the transformed GC'd portions of the heap H'_{ig} must satisfy.

Critically, the relationship between the original world W and the new world W' (world extension, written $\sqsubseteq_{\mathbb{L},\eta}$) indicates the future world does not overwrite manual locations $\mathbb{L}$ (made up of sets of locations relevant to the left and right side respectively) and preserves the contents of garbage-collected locations η (a set of related pairs of locations). We compute a relevant set of garbage-collected locations η either directly using rchgclocs(W, L_1, L_2), which is the locations in $W.\Psi$ in either L_1 or L_2 or, using our syntactic shorthand, $\sqsubseteq_{\mathsf{H}_1,\mathsf{H}_2,\mathsf{e}_1,\mathsf{e}_2}$, which sets $\mathbb{L} = (\mathrm{dom}(\mathsf{H}_1), \mathrm{dom}(\mathsf{H}_2))$ and η to be those locations free in e_1 and e_2, plus any garbage-collected locations stored in the owned heaps $\mathsf{H}_1/\mathsf{H}_2$. We need this to be able to support freeing, garbage collecting, and re-using locations while ensuring the locations relevant to the term (i.e., free locations and GC'd locations stored in the term's owned memory) remain in the world.

While our target supports dynamic failure (in the form of the fail term), our logical relation rules out that possibility, ensuring that there are no errors from the source nor from the conversion. This is, of course, a choice we made, which may be stronger than desired for some languages (and, indeed, for our previous two case studies), but given our choice of conversions, it is possible.

With the logical relation in hand, we prove analogous theorems to Lemma 2.1 (Convertibility Soundness), Theorem 2.2 (Fundamental Property), Theorem 2.3 (Type Safety for Lang A), and Theorem 2.4 (Type Safety for Lang B).

Our convertibility soundness result proves that our conversions above between garbage-collected and manual references, as well as $\mathbf{L}^3$ booleans and `MiniML` Church booleans (described above) are sound. We also show that $\tau_1 \rightarrow \tau_2 \sim \mathbf{!(!}\boldsymbol{\tau}_1 \multimap \boldsymbol{\tau}_2\mathbf{)}$ assuming $\tau_1 \sim \boldsymbol{\tau}_1$ and $\tau_2 \sim \boldsymbol{\tau}_2$.

5 Related Work and Conclusion

Most research on interoperability has focused either on reducing boilerplate or improving performance. We will not discuss those, focusing instead on work addressing reasoning and soundness.

Multi-language semantics. Matthews and Findler [32] studied the question of the interoperability of source languages, developing the idea of a syntactic multi-language with *boundary terms* (c.f., contracts [18, 19]) that mediate between the two languages. They focused on a static language interacting with a dynamic one, but similar techniques have been applied widely (e.g., object-oriented [20, 21], affine and unrestricted [48], simple and dependently typed [40], functional language and assembly [41], linear and unrestricted [43]) and used to prove compiler properties (e.g., correctness [42], full abstraction [2, 36]). More recently, there has been an effort understand this construction from a denotational [15] and categorical [14] perspective.

$$
\begin{aligned}
\mathcal{V}[\![\alpha]\!]_\rho &= \rho.\mathsf{F}(\alpha)\\
\mathcal{V}[\![\mathtt{unit}]\!]_\rho &= \{(W, (\emptyset, ()), (\emptyset, ()))\}\\
\mathcal{V}[\![\tau_1 \to \tau_2]\!]_\rho &= \{(W, (\emptyset, \lambda \mathrm{x}_1.\mathrm{e}_1), (\emptyset, \lambda \mathrm{x}_2.\mathrm{e}_2)) \mid \forall W', \mathrm{v}_1, \mathrm{v}_2.\\
&\qquad W \sqsubseteq_{\emptyset,\emptyset,\mathrm{e}_1,\mathrm{e}_2} W' \wedge (W', (\emptyset, \mathrm{v}_1), (\emptyset, \mathrm{v}_2)) \in \mathcal{V}[\![\tau_1]\!]_\rho \implies\\
&\qquad (W', (\emptyset, [\mathrm{x}_1 \mapsto \mathrm{v}_1]\mathrm{e}_1), (\emptyset, [\mathrm{x}_2 \mapsto \mathrm{v}_2]\mathrm{e}_2)) \in \mathcal{E}[\![\tau_2]\!]_\rho\}\\
\mathcal{V}[\![\forall \alpha.\tau]\!]_\rho &= \{(W, (\emptyset, \lambda\_.\mathrm{e}_1), (\emptyset, \lambda\_.\mathrm{e}_2)) \mid \forall R \in \mathit{RelT}, W'.\\
&\quad W \sqsubseteq_{\emptyset,\emptyset,\mathrm{e}_1,\mathrm{e}_2} W' \implies (W', (\emptyset, \mathrm{e}_1), (\emptyset, \mathrm{e}_2)) \in \mathcal{E}[\![\tau]\!]_{\rho[\mathsf{F}(\alpha) \mapsto R]}\}\\
\mathcal{V}[\![\mathtt{ref}\ \tau]\!]_\rho &= \{(W, (\emptyset, \ell_1), (\emptyset, \ell_2)) \mid W.\Psi(\ell_1, \ell_2) = \lfloor \mathcal{V}[\![\tau]\!]_\rho \rfloor_{W.k}\}\\
\mathcal{V}[\![\langle \boldsymbol{\tau} \rangle]\!]_\rho &= \mathcal{V}[\![\boldsymbol{\tau}]\!]_\rho\\
\mathcal{V}[\![\mathbf{unit}]\!]_\rho &= \{(W, (\emptyset, ()), (\emptyset, ()))\}\\
\mathcal{V}[\![\mathbf{bool}]\!]_\rho &= \{(W, (\emptyset, \mathrm{b}), (\emptyset, \mathrm{b})) \mid \mathrm{b} \in \{0, 1\}\}\\
\mathcal{V}[\![\boldsymbol{\tau}_1 \otimes \boldsymbol{\tau}_2]\!]_\rho &= \{(W, (\mathrm{H}_{1l} \uplus \mathrm{H}_{1r}, (\mathrm{v}_{1\mathrm{l}}, \mathrm{v}_{1\mathrm{r}})), (\mathrm{H}_{2l} \uplus \mathrm{H}_{2r}, (\mathrm{v}_{2\mathrm{l}}, \mathrm{v}_{2\mathrm{r}}))) \mid\\
&\qquad (W, (\mathrm{H}_{1l}, \mathrm{v}_{1\mathrm{l}}), (\mathrm{H}_{2l}, \mathrm{v}_{2\mathrm{l}})) \in \mathcal{V}[\![\boldsymbol{\tau}_1]\!]_\rho \wedge\\
&\qquad (W, (\mathrm{H}_{1r}, \mathrm{v}_{1\mathrm{r}}), (\mathrm{H}_{2r}, \mathrm{v}_{2\mathrm{r}})) \in \mathcal{V}[\![\boldsymbol{\tau}_2]\!]_\rho\}\\
\mathcal{V}[\![\boldsymbol{\tau}_1 \multimap \boldsymbol{\tau}_2]\!]_\rho &= \{(W, (\mathrm{H}_1, \lambda \mathrm{x}_1.\mathrm{e}_1), (\mathrm{H}_2, \lambda \mathrm{x}_2.\mathrm{e}_2)) \mid \forall W', \mathrm{H}_{1v}, \mathrm{v}_1, \mathrm{H}_{2v}, \mathrm{v}_2.\\
&\quad W \sqsubseteq_{\mathrm{H}_1,\mathrm{H}_2,\mathrm{e}_1,\mathrm{e}_2} W' \wedge (W', (\mathrm{H}_{1v}, \mathrm{v}_1), (\mathrm{H}_{2v}, \mathrm{v}_2)) \in \mathcal{V}[\![\boldsymbol{\tau}_1]\!]_\rho \implies\\
&\quad (W', (\mathrm{H}_1 \uplus \mathrm{H}_{1v}, [\mathrm{x}_1 \mapsto \mathrm{v}_1]\mathrm{e}_1), (\mathrm{H}_2 \uplus \mathrm{H}_{2v}, [\mathrm{x}_2 \mapsto \mathrm{v}_2]\mathrm{e}_2)) \in \mathcal{E}[\![\boldsymbol{\tau}_2]\!]_\rho\}\\
\mathcal{V}[\![!\boldsymbol{\tau}]\!]_\rho &= \{(W, (\emptyset, \mathrm{v}_1), (\emptyset, \mathrm{v}_2)) \mid (W, (\emptyset, \mathrm{v}_1), (\emptyset, \mathrm{v}_2)) \in \mathcal{V}[\![\boldsymbol{\tau}]\!]_\rho\}\\
\mathcal{V}[\![\mathbf{ptr}\ \zeta]\!]_\rho &= \{(W, (\emptyset, \ell_1), (\emptyset, \ell_2)) \mid \rho.\mathbf{L3}(\zeta) = (\ell_1, \ell_2)\}\\
\mathcal{V}[\![\mathbf{cap}\ \zeta\ \boldsymbol{\tau}]\!]_\rho &= \{(W, (\mathrm{H}_1 \uplus \{\ell_1 \mapsto \mathrm{v}_1\}, ()), (\mathrm{H}_2 \uplus \{\ell_2 \mapsto \mathrm{v}_2\}, ())) \mid\\
&\qquad \rho.\mathbf{L3}(\zeta) = (\ell_1, \ell_2) \wedge (W, (\mathrm{H}_1, \mathrm{v}_1), (\mathrm{H}_2, \mathrm{v}_2)) \in \mathcal{V}[\![\boldsymbol{\tau}]\!]_\rho\}\\
\mathcal{V}[\![\forall \zeta.\boldsymbol{\tau}]\!]_\rho &= \{(W, (\mathrm{H}_1, \lambda\_.\mathrm{e}_1), (\mathrm{H}_2, \lambda\_.\mathrm{e}_2)) \mid\\
&\quad \forall \ell_1 \ell_2.\, (W, (\mathrm{H}_1, \mathrm{e}_1), (\mathrm{H}_2, \mathrm{e}_2)) \in \mathcal{E}[\![\boldsymbol{\tau}]\!]_{\rho[\mathbf{L3}(\zeta) \mapsto (\ell_1, \ell_2)]}\}\\
\mathcal{V}[\![\exists \zeta.\boldsymbol{\tau}]\!]_\rho &= \{(W, (\mathrm{H}_1, \mathrm{v}_1), (\mathrm{H}_2, \mathrm{v}_2) \mid\\
&\quad \exists \ell_1 \ell_2.\, (W, (\mathrm{H}_1, \mathrm{v}_1), (\mathrm{H}_2, \mathrm{v}_2)) \in \mathcal{V}[\![\boldsymbol{\tau}]\!]_{\rho[\mathbf{L3}(\zeta) \mapsto (\ell_1, \ell_2)]}\}
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{E}[\![\tau]\!]_\rho = \{&(W, (\mathrm{H}_1, \mathrm{e}_1), (\mathrm{H}_2, \mathrm{e}_2)) \mid\\
&\forall L_1, L_2, \mathrm{v}_1, \mathrm{H}_{1g+}, \mathrm{H}_{2g+} : W, \mathrm{H}_{1r} : \mathit{MHeap}, \mathrm{H}_{1*}.\\
&(\mathrm{H}_{1g+} \uplus \mathrm{H}_1 \uplus \mathrm{H}_{1r}, \mathrm{e}_1) \xrightarrow{*}_{L_1} (\mathrm{H}_{1*}, \mathrm{v}_1) \nrightarrow_{L_1}\\
&\implies \exists \mathrm{H}'_1, \mathrm{H}'_{1g}.\forall \mathrm{H}_{2r} : \mathit{MHeap}.\exists \mathrm{H}'_2, W', \mathrm{H}'_{2g}, \mathrm{v}_2.\\
&\quad \mathrm{H}_{1*} = \mathrm{H}'_{1g} \uplus \mathrm{H}'_1 \uplus \mathrm{H}_{1r} \ \wedge \mathrm{H}'_{1g}, \mathrm{H}_{2g'} : W' \wedge\\
&W \sqsubseteq_{(\mathrm{dom}(\mathrm{H}_{1r}),\mathrm{dom}(\mathrm{H}_{2r})),\mathrm{rchgclocs}(W,L_1 \cup \mathit{FL}(\mathrm{cod}(\mathrm{H}_{1r})),L_2 \cup \mathit{FL}(\mathrm{cod}(\mathrm{H}_{2r})))} W'\\
&\quad \wedge (W', (\mathrm{H}'_1, \mathrm{v}_1), (\mathrm{H}'_2, \mathrm{v}_2)) \in \mathcal{V}[\![\tau]\!]_\rho \wedge\\
&\quad (\mathrm{H}_{2g+} \uplus \mathrm{H}_2 \uplus \mathrm{H}_{2r}, \mathrm{e}_2) \xrightarrow{*}_{L_2} (\mathrm{H}'_{2g} \uplus \mathrm{H}'_2 \uplus \mathrm{H}_{2r}, \mathrm{v}_2) \nrightarrow_{L_2}\\
&\quad \wedge \mathrm{H}_{1'} = \mathrm{H}_{2'} = \emptyset\}
\end{aligned}
$$

Note the highlighted parts only apply to MiniML types.

Figure 14. Logical Relation for MiniML and $\mathbf{L}^3$.

Barrett et al. [6] take a slightly different path, directly mixing languages (PHP and Python) and allowing bindings from one to be used in the other, though to similar ends.

Interoperability via typed targets. Shao and Trifonov [44, 49] studied interoperability much earlier, and closer to our context: they consider interoperability mediated by translation to a common target. They tackle the problem that one language has access to control effects and the other does not. Their approach, however, is different: it relies upon a target language with an effect-based type system that is sufficient to capture the safety invariants, whereas while our realizability approach can certainly benefit from typed target languages, it doesn't rely upon them.

An abstract framework for unsafe FFIs. Turcotte et al. [50] advocate a framework using an abstract version of the foreign language, so soundness can be proved 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 code that can do arbitrary unsound behavior and thus blamed for all unsoundness. While this approach seems promising in the context of unsound languages, it is less clear how it applies to sound languages.

Semantic Models and Realizability Models The use of semantic models to prove type soundness has a long history [33]. We make use of step-indexed models [4, 5], developed as part of the Foundational Proof-Carrying Code [1] project, which showed how to scale the semantic approach to complex features found in real languages such as recursive types and higher-order mutable state. While much of the recent work that uses step-indexed models is concerned with program equivalence, one recent project that focuses on type soundness is RustBelt [26]: they give a semantic model of λ_{Rust} types and use it to prove the soundness of λ_{Rust} typing rules, but also to prove that the λ_{Rust} implementation of standard library features (essentially unsafe code) are semantically sound inhabitants of their ascribed type specification.

Unlike the above, our realizability model interprets source types as sets of target terms. Our work takes inspiration from a line of work by Benton and collaborators on "low-level semantics for high-level types" (dubbed "realistic realizability") [8]. Such models were used to prove type soundness of standalone languages, specifically, Benton and Zarfaty [12] proved an imperative while language sound and Benton and Tabareau [11] proved type soundness for a simply typed functional language, both times interpreting source types as relations on terms of an idealized assembly and allowing for compiled code to be linked with a verified memory allocation module implemented in assembly [8]. Krishnaswami et al. [29] make use of a realizability model to prove consistency of LNL_D a core type theory that integrates linearity and full type dependency. The linear parts of their model, like our interpretation of $\mathbf{L}^3$ types, are directly inspired by the semantic model for $\mathbf{L}^3$ by Ahmed et al. [3]. Such realizability models have also been used by Jensen et al. [25] to verify low-level code using a high-level separation logic, and by Benton and Hur [9] to verify compiler correctness.

Finally, New et al. [35, 37, 38] make use of realizability models in their work on semantic foundations of gradual typing, work that we have drawn inspiration from, given gradual typing is a special instance of language interoperability. They compile type casts in a surface gradual language to a target Call-By-Push-Value [30] language without casts, build a realizability model of gradual types and type precision as relations on target terms, and prove properties about the gradual surface language using the model.

Conclusion and Future Work We have presented a novel framework for the design and verification of sound language interoperability where that interoperability happens, as in practical systems, after compilation. The realizability models at the heart of our technique give us powerful reasoning tools, including the ability to encode static invariants that are otherwise impossible to express in often untyped or low-level target languages. Even when it is possible to turn static source-level invariants into dynamic target-level checks, the ability to instead move these invariants into the model allows for more performant (and perhaps, realistic) compilers without losing the ability to prove soundness.

In the future, we hope to apply the framework to further explorations of the interoperability design space, e.g., to investigate interactions between lazy and strict languages (compilation to Call-By-Push-Value [30] may illuminate conversions), between single-threaded and concurrent languages (session types [23, 24, 47] may help guide interoperability with process calculi like the π-calculus [34]), between different control effects, and between Rust and a GC'ed language such as ML, Java, or Haskell compiled to a low-level target.

References

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

[2] 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

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

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

[5] 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

[6] 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), Vol. 56)*, Shriram Krishnamurthi and Benjamin S. Lerner (Eds.). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 3:1–3:27. https://doi.org/10.4230/LIPIcs.ECOOP.2016.3

[7] 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

[8] Nick Benton. 2006. Abstracting allocation: The new new thing. In *Computer Science Logic (CSL)*.

[9] 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* (Edinburgh, Scotland) *(ICFP '09)*. ACM, New York, NY, USA, 97–108. https://doi.org/10.1145/1596550.1596567

[10] Nick Benton, Andrew Kennedy, and Claudio V Russo. 2004. Adventures in interoperability: the sml. net experience. In *Proceedings of the 6th ACM SIGPLAN International conference on Principles and Practice of Declarative Programming*. 215–226.

[11] 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.

[12] Nick Benton and Uri Zarfaty. 2007. Formalizing and Verifying Semantic Type Soundness of a Simple Compiler. In *Proceedings of the 9th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming* (Wroclaw, Poland) *(PPDP '07)*. Association for Computing Machinery, New York, NY, USA, 1–12. https://doi.org/10.1145/1273920.1273922

[13] 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.

[14] Samuele Buro, Roy Crole, and Isabella Mastroeni. 2020. Equational logic and categorical semantics for multi-languages. *Electronic Notes in Theoretical Computer Science* 352 (2020), 79–103.

[15] Samuele Buro and Isabella Mastroeni. 2019. On the Multi-Language Construction.. In *ESOP*. 293–321.

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

[17] Christos Dimoulas, Sam Tobin-Hochstadt, and Matthias Felleisen. 2012. Complete Monitors for Behavioral Contracts. In *European Symposium on Programming (ESOP)*.

[18] Robert Bruce Findler and Matthias Blume. 2006. Contracts as pairs of projections. In *International Symposium on Functional and Logic Programming*. Springer, 226–241.

[19] Robert Bruce Findler and Matthias Felleisen. 2002. Contracts for higher-order functions. In *Proceedings of the seventh ACM SIGPLAN international conference on Functional programming*. 48–59.

[20] Kathryn E Gray. 2008. Safe cross-language inheritance. In *European Conference on Object-Oriented Programming*. Springer, 52–75.

[21] Kathryn E Gray, Robert Bruce Findler, and Matthew Flatt. 2005. Fine-grained interoperability through mirrors and contracts. *ACM SIGPLAN Notices* 40, 10 (2005), 231–245.

[22] Rich Hickey. 2020. A history of Clojure. *Proceedings of the ACM on programming languages* 4, HOPL (2020), 1–46.

[23] Kohei Honda. 1993. Types for dyadic interaction. In *International Conference on Concurrency Theory*. Springer, 509–523.

[24] Kohei Honda, Vasco T Vasconcelos, and Makoto Kubo. 1998. Language primitives and type discipline for structured communication-based programming. In *European Symposium on Programming*. Springer, 122–138.

[25] Jonas B. Jensen, Nick Benton, and Andrew Kennedy. 2013. High-Level Separation Logic for Low-Level Code *(POPL '13)*. Association for Computing Machinery, New York, NY, USA, 301–314. https://doi.org/10.1145/2429069.2429105

[26] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. 2018. RustBelt: Securing the Foundations of the Rust Programming Language. In *ACM Symposium on Principles of Programming Languages (POPL)*.

[27] Andrew Kennedy and Don Syme. 2001. Design and Implementation of Generics for the .NET Common Language Runtime. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation* (Snowbird, Utah, USA) *(PLDI '01)*. Association for Computing Machinery, New York, NY, USA, 1–12. https://doi.org/10.1145/378795.378797

[28] Robert Kleffner. 2017. *A Foundation for Typed Concatenative Languages.* Master's thesis. Northeastern University.

[29] 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

[30] Paul Blain Levy. 2001. *Call-by-Push-Value.* Ph. D. Dissertation. Queen Mary, University of London, London, UK.

[31] Phillip Mates, Jamie Perconti, and Amal Ahmed. 2019. Under Control: Compositionally Correct Closure Conversion with Mutable State. In *ACM Conference on Principles and Practice of Declarative Programming (PPDP)*.

[32] 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

[33] Robin Milner. 1978. A theory of type polymorphism in programming. *J. Comput. Syst. Sci.* 17 (1978), 348–375.

[34] Robin Milner, Joachim Parrow, and David Walker. 1992. A calculus of mobile processes, i. *Information and computation* 100, 1 (1992), 1–40.

[35] Max S. New and Amal Ahmed. 2018. Graduality from Embedding-Projection Pairs, In ICFP. *Proceedings of the ACM on Programming*

Languages 2, 73:1–73:30.

[36] 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

[37] Max S. New, Dustin Jamner, and Amal Ahmed. 2020. Graduality and Parametricity: Together Again for the First Time. *Proceedings of the ACM on Programming Languages* 4, POPL, 46:1–46:32.

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

[39] Martin Odersky and Matthias Zenger. 2005. Scalable component abstractions. In *Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*. 41–57.

[40] 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

[41] 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

[42] 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, Vol. 8410)*, Zhong Shao (Ed.). Springer, 128–148. https://doi.org/10.1007/978-3-642-54833-8_8

[43] 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, Vol. 10803)*, Christel Baier and Ugo Dal Lago (Eds.). Springer, 146–162. https://doi.org/10.1007/978-3-319-89366-2_8

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

[45] T. Stephen Strickland, Sam Tobin-Hochstadt, Robert Bruce Findler, and Matthew Flatt. 2012. Chaperones and Impersonators: Run-Time Support for Reasonable Interposition. In *ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)* (Tucson, Arizona, USA). Association for Computing Machinery, New York, NY, USA, 943–962. https://doi.org/10.1145/2384616.2384685

[46] Don Syme. 2006. Leveraging. NET meta-programming components from F# integrated queries and interoperable heterogeneous execution. In *Proceedings of the 2006 workshop on ML*. 43–54.

[47] Kaku Takeuchi, Kohei Honda, and Makoto Kubo. 1994. An interaction-based language and its typing system. In *International Conference on Parallel Architectures and Languages Europe*. Springer, 398–413.

[48] Jesse Tov and Riccardo Pucella. 2010. Stateful Contracts for Affine Types. In *Programming Languages and Systems, 19th European Symposium on Programming, ESOP 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus, March 20-28, 2010. Proceedings* (Paphos, Cyprus).

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

[50] 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), Vol. 134)*, Alastair F. Donaldson (Ed.). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 16:1–16:32. https://doi.org/10.4230/LIPIcs.ECOOP.2019.16

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

[52] Jyun-Yan You. 2021. Rust Bindgen. https://github.com/rust-lang/rust-bindgen