Semantic Soundness for Language Interoperability

DANIEL PATTERSON, Northeastern University, USA
NOBLE MUSHTAK, Northeastern University, USA
ANDREW WAGNER, Northeastern University, USA
AMAL AHMED, Northeastern University, USA

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 [43] proposed a 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 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-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 and show how the approach helps designers better take advantage of efficient enforcement mechanisms and opportunities for sound sharing that may not be obvious in a setting divorced from implementations.

Additional Key Words and Phrases: language interoperability, type soundness, semantics, logical relations

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 store written in Rust interoperating with business logic implemented in OCaml. Sometimes, this interoperability is realized by targeting a common platform (e.g., Scala [51] and Clojure [29] for the JVM, or SML [11] and F# [60] 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 [8], C->Haskell [18], OCaml-ctypes [66], NLFFI [15], Rust's bindgen [67], etc).

In 2007, Matthews and Findler [43] 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 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 e_B of type

Authors' addresses: Daniel Patterson, Northeastern University, 440 Huntington Avenue, Boston, MA, 02115, USA, dbp@dbpmail.net; Noble Mushtak, Northeastern University, 440 Huntington Avenue, Boston, MA, 02115, USA, mushtak.n@northeastern.edu; Andrew Wagner, Northeastern University, 440 Huntington Avenue, Boston, MA, 02115, USA, ahwagner@ccs.neu.edu; Amal Ahmed, Northeastern University, 440 Huntington Avenue, Boston, MA, 02115, USA, amal@ccs.neu.edu.

2018. 2475-1421/2018/1-ART1 $15.00
https://doi.org/

τ_B to be embedded in an A context that expects a term of type τ_A, and likewise for the boundary ${}^{\tau_B}\mathcal{B}\mathcal{A}^{\tau_A}(\cdot)$. Operationally, the term ${}^{\tau_A}\mathcal{A}\mathcal{B}^{\tau_B}(\mathrm{e_B})$ evaluates e_B using the B-language semantics to ${}^{\tau_A}\mathcal{A}\mathcal{B}^{\tau_B}(\mathrm{v_B})$ 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 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 [52], between languages with unrestricted and substructural types [57, 63], between a high-level functional language and assembly [53], and between source and target languages of compilers [2, 48, 54].

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. The job of the glue code is to convert between data representations and calling conventions so that values and code coming from one language are usable in the other. While one could 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, this is neither practical nor informative. 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. Moreover, certain conversions, even if possible, might be undesireable 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, though the framework should accommodate alternate syntax. Indeed, we differ in that rather than proving operational properties of the syntactic source, we 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, which need to be compatible with the model. This kind of model, often called a realizability model, is not a new idea — for instance, Benton and Zarfaty [13] and Benton and Tabareau [12] 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 capture the representation choices made by the compilers. With this model, we can then give meaning to a boundary ${}^{\tau_B}\mathcal{B}\mathcal{A}^{\tau_A}(\cdot)$: 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.

Of course, building realizability models can be a challenge, but it exactly reflects the challenge of reasoning about information loss during compilation, which is necessary to prove type soundness of

systems as implemented. To show that a system is sound, we must show that glue code converting between source types τ_A and τ_B either does so correctly or raises an error when correct conversion is impossible. However, this glue code does not actually operate on source data of type τ_A and τ_B, as in Matthews-Findler, but on compiled target code implementing τ_A and τ_B. If, for example, we are compiling typed languages to an untyped target, there is a non-trivial gap between the source and target that makes reasoning about these conversions hard. In fact, it may be difficult to even characterize what a τ_B is in the target, which is a prerequisite for proving any particular τ_A to τ_B conversion is correct. For instance, it is hard to characterize an affine function type as a set of untyped target terms, especially if we want to do so without changing the target language (which is rarely feasible) or abandoning static for dynamic enforcement (which, while pragmatic, is inefficient). We will show that we can characterize such functions, in §5, by pushing the reasoning into the model — in this case, reasoning about static enforcement of affinity. Even in the case that capturing invariants is straightforward, some conversions are particularly inefficient, and may motivate a different design altogether[1] or even the desire to rule it out.

A Simple Example. To illustrate how compilation influences the model, and how that, in turn, guides sound conversion, we consider a small example. Suppose we have two source languages, A and B, and an untyped target T with integers and if0. We define an inductive convertibility judgment of the form $\tau_A \sim \tau_B$ to specify what types may be converted. For example, a base case might be the rule: bool ∼ int.

Every convertibility rule requires "glue code" conversions, so for the above rule, we have conversions $C_{\texttt{bool}\mapsto\texttt{int}}$ and $C_{\texttt{int}\mapsto\texttt{bool}}$, implemented in target code. Of course, the way we implement these depends on the compilers. Suppose if is compiled to if0, structurally recurring on subterms. Naturally, we compile true to 0, but we may compile false to any non-zero integer (which are all "falsy" at the target). Indeed, our model may specify the target values that inhabit bool as follows: $\mathcal{V}[\![\texttt{bool}]\!] = \{n \mid n \in \mathbb{Z}\}$. Now, if ints are compiled to target integers — and we model them as $\mathcal{V}[\![\texttt{int}]\!] = \{n \mid n \in \mathbb{Z}\}$ — then the conversions between bool and int are no-ops.

As a second scenario, suppose if is instead compiled to target code that takes the "then" branch only when the conditional value equals 0 and takes the "else" branch only when the conditional value equals 1. Whereas before, other integers were "falsy," now, they are nonsensical as bools. Then we would have to model bool as follows: $\mathcal{V}[\![\texttt{bool}]\!] = \{0, 1\}$. For the conversions between bool and int to be sound, converting from bool to int would still be the identity, but the reverse must either fail if the value is not 0 or 1, or collapse any other value to either 0 or 1. Evidently, what the conversions do depends upon how we define the type interpretations, which in turn are constrained by how the introduction and elimination forms for each type in the source are compiled — since otherwise, we wouldn't be able to prove the stand-alone language sound.

Observing these constraints, the compiler writer from the second scenario may see an opportunity to edit the compiler for if to match the first scenario, so that both conversions between bool and int are the identity. But sometimes, legacy considerations make changes to an existing compiler infeasible. Either way, our framework helps designers understand and account for these design and implementation tradeoffs.

Once we have the convertibility rules, conversions, compilers, and logical relations for languages A and B defined, we prove type soundness in the standard semantic way. First, we show that each source typing rule for both languages entails an analogous semantic "compatibility" lemma. With these compatibility lemmas in hand, we prove the Fundamental Property of the logical relation, which says that the *compilation of any well-typed source term* is in the logical relation. Finally, we

[1]The WebAssembly Interface Types proposal [26] concerns efficiently moving between representations without violating soundness invariants. This is the sort of under-the-hood reasoning that must guide interoperability.

show that any term in the logical relation is type safe: i.e., the term runs without getting stuck and without raising any errors other than the specified set of "interoperability errors" (which arise when interoperability is enforced via runtime mechanisms).

Contributions. To demonstrate the use and benefits of our framework, we present four 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 [59] — convertible reference types must be inhabitated by the *very same* set of target terms.
(2) Pure Polymorphism & Effects (§3): We consider how System F, a pure polymorphic language, can interact with $\mathbf{L}^3$ [4], a language that uses linear capabilities to support safe strong updates to a mutable heap but lacks type abstraction. We demonstrate a type-level form of interoperability that allows generics to be used in both languages without violating any invariants of either language.
(3) Affine & Unrestricted (§4): We consider how MiniML, a standard functional language with mutable references, can interact with Affi, an affine language. We allow affine code to be safely embedded in unrestricted code and vice versa by using runtime checks to ensure that affine resources are used at most once.
(4) Affine & Unrestricted, Efficiently (§5): We consider how to *efficiently* regulate the interactions between MiniML and Affi from the previous case study. In particular, we revise the compiler to eliminate all unnecessary runtime checks for code where affine use of arguments is statically enforced and show that the entire system is still safe.

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 three studies, all of which involve polymorphic languages, we give binary models (source types as *relations* on target terms) so that we can establish parametricity as well as type soundness.

Definitions and proofs elided from this paper are provided in our 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 because giving a pointer to a foreign language can allow for *arbitrary* data to be written to its address. The specific challenge is that 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 these restrictions. One existing approach to this problem is to create proxies, where data is guarded or converted before being read or written [19, 42, 59]. However, this comes with significant runtime overhead. Here, our framework suggests 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," respectively). 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

RefHL	Type τ	::=	unit | bool | $\tau + \tau$ | $\tau \times \tau$ | $\tau \to \tau$ | ref τ
	Expression e	::=	() | true | false | x | inl e | inr e | (e, e) | fst e | snd e | if e e e | match e x{e} y{e} | λx : τ.e | e e | ref e | !e | e := e | $(\!|\mathsf{e}|\!)_\tau$
RefLL	Type τ	::=	int | $[\tau]$ | $\tau \to \tau$ | ref τ
	Expression e	::=	n | x | [e, . . .] | e[e] | λx : τ.e | e e | e + e | if0 e e e | ref e | !e | e := e | $(\!|\mathsf{e}|\!)_\tau$

Fig. 1 . Syntax for RefHL and RefLL.

Heap H	::=	$\{\ell : v, \ldots\}$	Stack S	::=	v, . . . , v | Fail c	Error Code c ::= Type | Idx | Conv
Program P	::=	· | i, P	Value v	::=	n | thunk P | ℓ | [v, . . .]	
Instruction i	::=	push v | add | less? | if0 P P | lam x.P | call | idx | len | alloc | read | write | fail c				

$\langle H; S; \text{push v}, P\rangle \to \langle H; S, v; P\rangle$

$\langle H; S, n', n; \text{add}, P\rangle \to \langle H; S, (n + n'); P\rangle$

$\langle H; S, n', n; \text{less?}, P\rangle \to \langle H; S, 0; P\rangle \quad (n < n')$

$\langle H; S, n', n; \text{less?}, P\rangle \to \langle H; S, 1; P\rangle \quad (n \geq n')$

$\langle H; S, 0; \text{if0 } P_1\ P_2, P\rangle \to \langle H; S; P_1, P\rangle$

$\langle H; S, n; \text{if0 } P_1\ P_2, P\rangle \to \langle H; S; P_2, P\rangle \quad (n \neq 0)$

$\langle H; S; \text{if0 } P_1\ P_2, P\rangle \to \langle H; S; \text{fail Type}\rangle \quad (S \neq S', n)$

$\langle H; S, v; \text{lam x.}P_1, P_2\rangle \to \langle H; S; [x \mapsto v]P_1, P_2\rangle$

$\langle H; S; \text{thunk } P_1; \text{call}, P_2\rangle \to \langle H; S; P_1, P_2\rangle$

$\langle H; S, [v_0, \ldots, v_{n'}], n; \text{idx}, P\rangle \to \langle H; S, v_n; P\rangle \quad (n \in [0, n'])$

$\langle H; S, [v_0, \ldots, v_{n'}], n; \text{idx}, P\rangle \to \langle H; S; \text{fail Idx}\rangle \quad (n \notin [0, n'])$

$\langle H; S, [v_0, \ldots, v_n]; \text{len}, P\rangle \to \langle H; S, (n + 1); P\rangle$

$\langle H; S, v; \text{alloc}, P\rangle \to \langle H \uplus \{\ell : v\}; S, \ell; P\rangle$

$\langle H \uplus \{\ell : v\}; S, \ell; \text{read}, P\rangle \to \langle H \uplus \{\ell : v\}; S, v; P\rangle$

$\langle H \uplus \{\ell : \_\}; S, \ell, v; \text{write}, P\rangle \to \langle H \uplus \{\ell : v\}; S; P\rangle$

$\langle H; S; \text{fail c}, P\rangle \to \langle H; \text{Fail c}; \cdot\rangle$

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

entirely standard — may be found in the supplementary material. These two languages are compiled (Fig. 3) into an untyped stack-based language called StackLang (inspired by [37]), 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. Note that for any instruction where the precondition on the stack is not met, the configuration steps to a program with fail Type, although these reduction rules have been elided.

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

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

We want to point out a few things about these rules. First, the convertibility judgment, which we will explain in detail later, 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.

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

SWAP ≜ lam x.(lam y.push x; push y)			DROP ≜ lam x.()		DUP ≜ lam x.(push x, push x)
`true`	⇝	push 0	$\mathbf{n}$	⇝	push n
`false`	⇝	push 1	$\mathbf{e_1 + e_2}$	⇝	$\mathbf{e_1^+}$, $\mathbf{e_2^+}$, SWAP, add
`()`	⇝	push 0			
`x`	⇝	push x	$\mathbf{x}$	⇝	push x
`inl e`	⇝	e^+, lam x.(push [0, x])			
`inr e`	⇝	e^+, lam x.(push [1, x])			
`if e` e_1 e_2	⇝	e^+, if0 e_1^+ e_2^+	**if0 e** $\mathbf{e_1}$ $\mathbf{e_2}$	⇝	$\mathbf{e^+}$, if0 $\mathbf{e_1^+}$ $\mathbf{e_2^+}$
`match e` `x{`e_1`} y{`e_2`}`	⇝	e^+, DUP, push 1, idx, SWAP, push 0, idx, if0 (lam x.e_1^+) (lam y.e_2^+)			
(e_1, e_2)	⇝	e_1^+, e_2^+, lam x_2, x_1.(push $[x_1, x_2]$)	$\mathbf{[e_1, \ldots, e_n]}$	⇝	$\mathbf{e_1^+, \ldots, e_n^+}$, lam $x_n, \ldots, x_1$. (push $[x_1, \ldots, x_n]$)
`fst e`	⇝	e^+, push 0, idx			
`snd e`	⇝	e^+, push 1, idx	$\mathbf{e_1[e_2]}$	⇝	$\mathbf{e_1^+}$, $\mathbf{e_2^+}$, idx
$\lambda x : \tau.e$	⇝	push (thunk lam x.e^+)	$\boldsymbol{\lambda x : \tau.e}$	⇝	push (thunk lam x.$\mathbf{e^+}$)
e_1 e_2	⇝	e_1^+, e_2^+, SWAP, call	$\mathbf{e_1}$ $\mathbf{e_2}$	⇝	$\mathbf{e_1^+}$, $\mathbf{e_2^+}$, SWAP, call
`ref e`	⇝	e^+, alloc	**ref e**	⇝	$\mathbf{e^+}$, alloc
`!e`	⇝	e^+, read	**!e**	⇝	$\mathbf{e^+}$, read
$e_1 := e_2$	⇝	e_1^+, e_2^+, write, push 0	$\mathbf{e_1 := e_2}$	⇝	$\mathbf{e_1^+}$, $\mathbf{e_2^+}$, write, push 0
$(\!\vert e \vert\!)_\tau$	⇝	e^+, $C_{\tau \mapsto \tau}$	$\mathbf{(\!\vert e \vert\!)_\tau}$	⇝	$\mathbf{e^+}$, $\mathbf{C_{\tau \mapsto \tau}}$

Fig. 3 . Compilers for RefHL and RefLL.

conversions, written $C_{\tau_A \mapsto \tau_B}$. 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. Note that an implementor can write these conversions based on a general understanding of the sets of target terms that inhabit each source type, before (or possibly, without ever) defining a proper semantic model. They would do this based on inspection of the compiler and the target.

We can see that `bool` and **int** both compile to target integers, and importantly, that `if` compiles to if0, which means that we choose to interpret `false` as any non-zero integer. That means that our conversions from `bool` to **int** are identities.

For sums, we can see that 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 most interesting, and the reason for this case study. Intuitively, if you exchange pointers, any value at the new type can be written, 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.

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.

$$\frac{}{C_{\texttt{bool}\mapsto\textbf{int}}, C_{\textbf{int}\mapsto\texttt{bool}} : \texttt{bool} \sim \textbf{int}} \qquad \frac{}{C_{\texttt{ref bool}\mapsto\textbf{ref int}}, C_{\textbf{ref int}\mapsto\texttt{ref bool}} : \texttt{ref bool} \sim \textbf{ref int}}$$

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

$C_{\texttt{bool}\mapsto\textbf{int}} \triangleq \cdot$

$C_{\texttt{ref bool}\mapsto\textbf{ref int}} \triangleq \cdot$

$C_{\textbf{int}\mapsto\texttt{bool}} \triangleq \cdot$

$C_{\textbf{ref}\ \textbf{int}\mapsto\texttt{ref bool}} \triangleq \cdot$

$C_{\tau_1+\tau_2\mapsto\textbf{[int]}} \triangleq$ DUP, push 1, idx, SWAP, push 0, idx, DUP, if0 (SWAP, $C_{\tau_1\mapsto\textbf{int}}$) (SWAP, $C_{\tau_2\mapsto\textbf{int}}$), lam x_v.lam x_t.push $[x_t, x_v]$

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

Fig. 4 . Conversions for RefHL and RefLL.

Our model, which is a standard step-indexed unary logical relation for a language with mutable state (essentiall following Ahmed [5]), is presented with some parts elided in Fig. 5 (the full treatment is in our supplementary materials). We construct an interpretation of source types as sets of *atoms* of the form (W, v) where v is a target term and W is a *world* comprised of a step index k and a *heap typing* Ψ, which maps locations to type interpretations in Typ. As is standard, Typ defines 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 $j \leq W.k$ and if all locations in $W.\Psi$ still have the same types (to approximation j).

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. Intuitively, $(W, v) \in \mathcal{V}[\![\tau]\!]$ says that the target value v belongs to (or behaves like a value of) type τ in world W. For example, $\mathcal{V}[\![\texttt{unit}]\!]$ is inhabited by 0 in any world. A more interesting case is $\mathcal{V}[\![\texttt{bool}]\!]$, which is the set of all target integers, not just 0 and 1 (c.f., the discussion in the Section 1) in any world. An array $\mathcal{V}[\![\textbf{[}\tau\textbf{]}]\!]$ 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↦v]P) is in the expression relation at the result type $\mathcal{E}[\![\tau_2]\!]$. Reference types $\mathcal{V}[\![\texttt{ref}\ \tau]\!]$ 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 $W.k$.

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 stack S, if the machine $\langle H; S; P\rangle$ terminates in j steps (where j is less than our step budget $W.k$), then either it ran to an 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 v and W' are in $\mathcal{V}[\![\tau]\!]$.

At the bottom of Fig. 5, we show a syntactic shorthand, $[\![\Gamma; \Gamma \vdash \mathbf{e} : \tau]\!]$, 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 that assign every $x : \tau \in \Gamma$ to a $v \in \mathcal{V}[\![\tau]\!]$.

With our logical relation, 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` $\sim$ **ref int** case; the full proof with the rest of the cases is in our supplementary materials. For `ref bool` $\sim$ **ref int**, what we need to show is that given any expression in $\mathcal{E}[\![\texttt{ref bool}]\!]$, if we apply the conversion (which does nothing), the result will be in $\mathcal{E}[\![\textbf{ref int}]\!]$. That amounts to showing that $\mathcal{V}[\![\texttt{ref bool}]\!] = \mathcal{V}[\![\textbf{ref int}]\!]$.

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 that a `ref bool` must be a location ℓ that, in the model, points to the value interpretation of `bool` (i.e., $\mathcal{V}[\![\texttt{bool}]\!]$). 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, $\mathcal{V}[\![\texttt{bool}]\!]$ must be the same as $\mathcal{V}[\![\textbf{int}]\!]$, 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:

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

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

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

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

Discussion. To construct a **ref** τ location ℓ', from the `ref` τ location ℓ, there are three choices:

(1) Pass the pointer across directly, as done above.
(2) Allocate fresh ℓ' and then copy and convert the data from ℓ to ℓ'. This requires mere convertibility between $\tau \sim \tau$ — not that their type interpretations be identical — but is inefficient (due to deep copies) and limits possibly desired aliasing.
(3) Rather than converting `ref` τ and **ref** τ, we can instead convert $(\texttt{unit} \to \tau) \times (\tau \to \texttt{unit})$ and $(\textbf{unit} \to \tau) \times (\tau \to \textbf{unit})$ (assuming we had pairs) – i.e., read/write proxies to the reference (similar to that used in [19]). This allows aliasing, i.e., both languages reading / writing to the same location, and will remain sound as long as the types are convertible, but again it comes at a significant runtime cost.

While we only showed the first, as we think it best demonstrates the power of our realizability model, our approach allows us to formalize and prove sound all three, observing the accompanying runtime cost. Indeed, it may be that with appropriate restrictions of types (to plain flat data, arrays of bytes, etc.), we can provide the first option as an performant but type-impoverished alternative to the richer-typed-but-slower latter solutions.

Another question to consider is what would happen if one were to pass a pointer from a statically type safe host language like `RefHL` to an unsound language like C. C allows arbitrary data to be

$$AtomVal_n = \{(W, \mathsf{v}) \mid W \in World_n\} \qquad World_n = \{(k, \Psi) \mid k < n \wedge \Psi \subset HeapTy_k\}$$

$$HeapTy_n = \{\ell \mapsto Typ_n, \ldots\} \qquad Typ_n = \{R \in 2^{AtomVal_n} \mid \forall (W, \mathsf{v}) \in R.\, \forall W'.\, W \sqsubseteq W' \implies (W', \mathsf{v}) \in R\}$$

$$\mathcal{V}[\![\texttt{bool}]\!] = \{(W, \mathsf{n})\} \qquad \mathcal{V}[\![\texttt{int}]\!] = \{(W, \mathsf{n})\}$$

$$\mathcal{V}[\![\texttt{unit}]\!] = \{(W, 0)\}$$

$$\mathcal{V}[\![\tau_1 + \tau_2]\!] = \{(W, [0, \mathsf{v}]) \mid (W, \mathsf{v}) \in \mathcal{V}[\![\tau_1]\!]\} \cup \{(W, [1, \mathsf{v}]) \mid (W, \mathsf{v}) \in \mathcal{V}[\![\tau_2]\!]\} \qquad \mathcal{V}[\![[\tau]]\!] = \{(W, [\mathsf{v}_1, \ldots, \mathsf{v}_n]) \mid (W, \mathsf{v}_i) \in \mathcal{V}[\![\tau]\!]\}$$

$$\mathcal{V}[\![\tau_1 \to \tau_2]\!] = \{(W, \text{thunk lam x.P}) \mid \forall \mathsf{v}, W' \sqsupseteq W.\, (W', \mathsf{v}) \in \mathcal{V}[\![\tau_1]\!] \implies (W', [\mathsf{x} \mapsto \mathsf{v}]\mathsf{P}) \in \mathcal{E}[\![\tau_2]\!]\}$$

$$\mathcal{V}[\![\texttt{ref}\ \tau]\!] = \{(W, \ell) \mid W.\Psi(\ell) = \lfloor \mathcal{V}[\![\tau]\!] \rfloor_{W.k}\} \qquad \mathcal{V}[\![\texttt{ref}\ \tau]\!] = \{(W, \ell) \mid W.\Psi(\ell) = \lfloor \mathcal{V}[\![\tau]\!] \rfloor_{W.k}\}$$

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

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

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

Fig. 5 . Logical relation for RefHL and RefLL.

written into pointers, so we would not generally expect the data to lie in the interpretation of a simple host type. However, if the host language has an untyped and unstructured data type, say, bytearray, and we send C a pointer to this type, then any data that C writes to that address (ignoring concerns about overwriting, etc.) would be interpretable in the host language, because bytearray includes all of the values that C can write.

3 PURE POLYMORPHISM & EFFECTS

For our second case study, we consider two languages: System F [25, 56] and core $\mathbf{L}^3$, a language with safe strong updates despite memory aliasing, supported via linear capabilities [4]. This case study highlights not only how a pure language can be isolated from one with effects (strong updates, no less), but more centrally, how polymorphism/generics in one language can be used, via a form of interoperability, from the other. 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.[2] While we are not claiming that interoperability could entirely replace built-in polymorphism, sound support for cross-language type instantiation and polymorphic libraries present a possible alternative, especially for smaller, perhaps more special-purpose, languages. Since this approach doesn't add generics to the language without them, any generic code must reside in the language with generics, though concrete instantiations can come from the other. This ends up being the same sort of separation of concerns that one sees when writing and using ML functors, but split across languages. For example, we could write:

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

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**, **[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

[2] e.g., Java 1.5/5, C# 2.0 [36] and more recently, in the Go programming language

System F	Type τ	::=	$\alpha \mid \tau \to \tau \mid \forall\alpha.\tau \mid \langle\boldsymbol{\tau}\rangle$
	Expression e	::=	$x \mid \lambda x:\tau.e \mid \Lambda\alpha.e \mid e\ e \mid e\ [\tau] \mid (\!|\mathbf{e}|\!)_\tau$
$\mathbf{L^3}$	Type $\boldsymbol{\tau}$	::=	$\mathbf{unit} \mid \mathbf{bool} \mid \boldsymbol{\tau} \otimes \boldsymbol{\tau} \mid \boldsymbol{\tau} \multimap \boldsymbol{\tau} \mid \mathbf{!}\boldsymbol{\tau} \mid \mathbf{ptr}\ \boldsymbol{\zeta} \mid \mathbf{cap}\ \boldsymbol{\zeta}\ \boldsymbol{\tau} \mid \forall\boldsymbol{\zeta}.\boldsymbol{\tau} \mid \exists\boldsymbol{\zeta}.\boldsymbol{\tau}$
	Value v	::=	$\boldsymbol{\lambda} x:\boldsymbol{\tau}.e \mid \mathbf{()} \mid \mathbb{B} \mid \mathbf{(v, v)} \mid \mathbf{!v} \mid \boldsymbol{\Lambda\zeta}.e \mid \ulcorner \boldsymbol{\zeta}, \mathbf{v} \urcorner$
	Expression e	::=	$\mathbf{v} \mid \mathbf{x} \mid \mathbf{(e,\ e)} \mid \mathbf{e\ e} \mid \mathbf{let\ ()\ =\ e\ in\ e} \mid \mathbf{if\ e\ e\ e} \mid \mathbf{let\ (x,\ x)\ =\ e\ in\ e}$ $\mid \mathbf{let\ !x\ =\ e\ in\ e} \mid \mathbf{dupl\ e} \mid \mathbf{drop\ e} \mid \mathbf{new\ e} \mid \mathbf{free\ e} \mid \mathbf{swap\ e\ e\ e}$ $\mid \mathbf{e}\ [\boldsymbol{\zeta}] \mid \ulcorner \boldsymbol{\zeta}, \mathbf{e} \urcorner \mid \mathbf{let} \ulcorner \boldsymbol{\zeta}, \mathbf{x} \urcorner \mathbf{\ =\ e\ in\ e} \mid (\!|\mathbf{e}|\!)_{\boldsymbol{\tau}} \mid \langle \mathbf{e} \rangle_{\boldsymbol{\tau}}$
	Foreign	::=	$\mathbf{unit} \mid \mathbf{bool} \mid \mathbf{ptr}\ \boldsymbol{\zeta} \mid \mathbf{!}\boldsymbol{\tau}$

Fig. 6 . Syntax for System F and $\mathbf{L^3}$.

Expressions e	::=	() | $\mathbb{Z}$ | ℓ | x | (e, e) | fst e | snd e | inl e | inr e | if e {e} {e} | match e x{e} y{e} | let x = e in e | λx{e} | e e | ref e | !e | e := e | fail c
Values v	::=	() | $\mathbb{Z}$ | ℓ | (v, v) | λx.e
Error Code c	::=	Type | Conv

Fig. 7 . Syntax for LCVM.

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 the source languages System F and $\mathbf{L^3}$, which have been augmented with forms for interoperability, in Fig. 6. $\mathbf{L^3}$ has linear capability types $\mathbf{cap}\ \boldsymbol{\zeta}\ \boldsymbol{\tau}$ (capability for abstract location $\boldsymbol{\zeta}$ storing data of type $\boldsymbol{\tau}$), unrestricted pointer types $\mathbf{ptr}\ \boldsymbol{\zeta}$ to support aliasing, and location abstraction ($\boldsymbol{\Lambda\zeta}.e : \forall\boldsymbol{\zeta}.\boldsymbol{\tau}$ and $\ulcorner \boldsymbol{\zeta}, \mathbf{v} \urcorner : \exists\boldsymbol{\zeta}.\boldsymbol{\tau}$). We compile (Fig. 8) both to an untyped lambda calculus, Scheme-like target LCVM, with pairs, sums, and mutable references (Fig. 7). This untyped target captures the typical challenge where the medium of interoperation supports less static reasoning than the sources, usually sitting at a lower level of abstraction. As in the previous case study, we have boundary terms, $(\!|\mathbf{e}|\!)_\tau$ and $(\!|\mathbf{e}|\!)_{\boldsymbol{\tau}}$, for *converting* a term and using it within the other language. In this case study, 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*[3] 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 to allow the polymorphic language to violate its invariants. To make this challenge material, our non-polymorphic language in this case study has linear resources in the form of 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 System F function that duplicates it, even if such function is well-typed (and parametric) in System F.

Convertibility. We solve this in two parts. First, we have a *foreign type*, $\langle\boldsymbol{\tau}\rangle$, which embeds an $\mathbf{L^3}$ type into the type grammar of System F. This foreign type, like any System F type, can be used to instantiate type abstractions, define functions, etc, but System F 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 $\langle\boldsymbol{\tau}\rangle \sim \boldsymbol{\tau}$, which allow terms of the form $(\!|\mathbf{e}|\!)_{\langle\boldsymbol{\tau}\rangle}$ (to bring an $\mathbf{L^3}$

[3]These are like "lumps" in [43].

$x, \mathbf{x}$	$\rightsquigarrow$	x	$\Lambda\alpha.e$	$\rightsquigarrow$	$\lambda\_.e^+$
$\lambda x:\tau.e$	$\rightsquigarrow$	$\lambda x.e^+$	$e\ [\tau]$	$\rightsquigarrow$	$e^+()$
$e_1 e_2$	$\rightsquigarrow$	$e_1^+ e_2^+$	$\langle\!\langle \mathbf{e} \rangle\!\rangle_\tau$	$\rightsquigarrow$	$C_{\tau\mapsto\tau}(\mathbf{e}^+)$
$\boldsymbol{\lambda x:\tau.e}$	$\rightsquigarrow$	$\lambda x.\mathbf{e}^+$	$\mathbf{drop\ e}$	$\rightsquigarrow$	$\mathbf{let\ \_ = e^+\ in\ ()}$
$\mathbf{e_1 e_2}$	$\rightsquigarrow$	$\mathbf{e_1^+ e_2^+}$	$\mathbf{new\ e}$	$\rightsquigarrow$	$\mathbf{let\ x_\ell = ref\ e^+\ in\ (x_\ell, ((), x_\ell))}$
$\mathbf{()}$	$\rightsquigarrow$	$()$	$\mathbf{free\ e}$	$\rightsquigarrow$	$\mathbf{let\ x = e^+ in\ (fst\ x, !(snd\ (snd\ x)))}$
$\mathbf{let\ () = e_1\ in\ e_2}$	$\rightsquigarrow$	$\mathbf{let\ \_ = e_1^+\ in\ e_2^+}$	$\mathbf{swap\ e_c\ e_p\ e_v}$	$\rightsquigarrow$	$\mathbf{let\ x_p = e_p^+,\ \_ = e_c^+, x_v' = !x_p,}$ $\mathbf{\_ = (x_p := e_v^+)\ in\ ((), x_v')}$
$\mathbf{true}$	$\rightsquigarrow$	0			
$\mathbf{false}$	$\rightsquigarrow$	1	$\mathbf{dupl\ e}$	$\rightsquigarrow$	$\mathbf{let\ e = e^+\ in\ (e,\ e)}$
$\mathbf{if\ e_1\ e_2\ e_3}$	$\rightsquigarrow$	$\mathbf{if\ e_1^+\ e_2^+\ e_3^+}$	$\Lambda\zeta.e$	$\rightsquigarrow$	$\lambda x_\zeta.e^+$
$\mathbf{(e_1,\ e_2)}$	$\rightsquigarrow$	$\mathbf{(e_1^+,\ e_2^+)}$	$e\ [\zeta]$	$\rightsquigarrow$	$e^+\ x_\zeta$
$\mathbf{let\ (x_1,\ x_2)\ = e_1}$ $\mathbf{\ in\ e_2}$	$\rightsquigarrow$	$\mathbf{let\ p = e_1^+, x_1 = fst\ p,}$ $\mathbf{\ x_2 = snd\ p\ in\ e_2^+}$	$\ulcorner\zeta,\ e\urcorner$	$\rightsquigarrow$	$(x_\zeta,\ e^+)$
$\mathbf{!v}$	$\rightsquigarrow$	$\mathbf{v}^+$	$\mathbf{let\ \ulcorner\zeta,\ x\urcorner = e_1}$ $\mathbf{\ in\ e_2}$	$\rightsquigarrow$	$\mathbf{let\ x_p = e_1^+,\ x_\zeta = fst\ x_p,}$ $\mathbf{\ x = snd\ x_p\ in\ e_2^+}$
$\mathbf{let\ !x = e_1\ in\ e_2}$	$\rightsquigarrow$	$\mathbf{let\ !x = e_1^+\ in\ e_2^+}$	$\langle\!\langle \mathbf{e} \rangle\!\rangle_\tau$	$\rightsquigarrow$	$C_{\tau\mapsto\tau}(e^+)$

Fig. 8 . Compilers for System F and $\mathbf{L^3}$.

term to System F) and $\langle\!\langle e \rangle\!\rangle_\tau$ (the reverse). Moreover, the conversion rule for foreign types restricts τ to a safe Foreign subset of types, but has no runtime consequences:

$$\frac{\tau \in \textsc{Foreign}}{C_{\langle\tau\rangle\mapsto\tau}, C_{\tau\mapsto\langle\tau\rangle} : \langle\tau\rangle \sim \tau} \qquad \begin{array}{l} C_{\langle\tau\rangle\mapsto\tau}(e) \triangleq e \\ C_{\tau\mapsto\langle\tau\rangle}(e) \triangleq e \end{array}$$

Then, to prove soundness we need to show that the Foreign 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 the case considered here, what we are required to show is that a Foreign type is duplicable (i.e., that none of its values own linear capabilities)—this includes **unit** and **bool**, but also $\mathbf{ptr}\ \zeta$ and any type of the form $!\tau$.

Consider the following example of cross-language instantiation:

$$(\Lambda\alpha.\lambda x{:}\alpha.\lambda y{:}\alpha.y)[\langle\mathbf{bool}\rangle]\ \langle\!\langle \mathbf{true} \rangle\!\rangle_{\langle\mathbf{bool}\rangle}\ \langle\!\langle \mathbf{false} \rangle\!\rangle_{\langle\mathbf{bool}\rangle}$$

The leftmost expression is a polymorphic System F 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 **bool** in $\mathbf{L^3}$ are embedded via the foreign conversion, $\langle\!\langle \cdot \rangle\!\rangle_{\langle\mathbf{bool}\rangle}$, which requires that $\mathbf{bool} \in$ Foreign. Not only does this mechanism allow $\mathbf{L^3}$ programmers to use polymorphic functions, but also System F programmers to use new base types.

Foreign types are just one of the two interoperability mechanisms that the multi-language provides; it also supports conversions. For example, we can define conversions between Church booleans in System F and ordinary booleans in $\mathbf{L^3}$, with which we can write the following program:

$$(\lambda x : \mathtt{BOOL}.x)\langle\!\langle \mathbf{true} \rangle\!\rangle_{\mathtt{BOOL}} \text{ where } \mathtt{BOOL} \triangleq \forall\alpha.\alpha \to \alpha \to \alpha$$

This relies on the following convertibility judgment and conversions:

$$\frac{}{\forall\alpha.\alpha \to \alpha \to \alpha \sim \mathbf{bool}} \qquad \begin{array}{l} C_{\mathtt{BOOL}\mapsto\mathbf{bool}}(e) \triangleq e\ ()\ 0\ 1 \\ C_{\mathbf{bool}\mapsto\mathtt{BOOL}}(e) \triangleq \mathrm{if0}\ e\ \{\Lambda\alpha.\lambda x{:}\alpha.\lambda y{:}\alpha.x\}\ \{\Lambda\alpha.\lambda x{:}\alpha.\lambda y{:}\alpha.y\} \end{array}$$

Semantic Model. To make sense of the above, and to prove that the conversions are sound, we build a binary logical relation, which of course has to line up with our compilers defined in Fig. 8. We present key selections in Fig. 9, referring the reader to our supplementary materials for the full definitions. Since both languages are terminating, our model is not step indexed, but it is binary, so as to be able to express the relational properties from System F.

Our model is similar to that of core $\mathbf{L^3}$ [4], though our relation is binary and theirs was unary. We give a value interpretation of source types $\mathcal{V}[\![\tau]\!]_\rho$ — which says when two heap-fragment-and-value pairs are related — as sets of tuples $(\mathsf{H}_1, \mathsf{v}_1, \mathsf{H}_2, \mathsf{v}_2)$ where the heap fragment H_i paired with value v_i is the portion of the heap *owned* by that value. The relational substitution ρ maps type variables α to arbitrary type interpretations R as is standard for System F and maps location variables ζ to concrete locations (ℓ_1, ℓ_2). Since System F is pure, both the expression relation $\mathcal{E}[\![\tau]\!]_\rho$ and the value relation $\mathcal{V}[\![\tau]\!]_\rho$ have empty $\emptyset$ heap fragments for all terms. 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 to the heap fragment owned by the contents of ℓ_i.

In the expression relation $\mathcal{E}[\![\tau]\!]_\rho$, we run the expressions with an arbitrary disjoint "rest" of the heap (H_{i+}) composed with the owned fragment (H_i). Then, assuming e_1 terminates, we expect that the "rest" heap is unchanged, the owned portion has been transformed into H'_1 and H'_{1+}, and that e_2 terminates in an analogous configuration, where $(\mathsf{H}'_1, \mathsf{v}_1, \mathsf{H}'_2, \mathsf{v}_2) \in \mathcal{V}[\![\tau]\!]_\rho$. The final configuration of heaps is a divergence from [4], and exists because our target language is garbage collected: in $\mathbf{L^3}$, locations can be deallocated. Since removing them from the heap happens by the garbage collector, operationally those locations have to go somewhere: they are not in the unchanged rest and they are no longer owned by the term, so they go into the H'_{i+} portion. Note that while both languages are terminating, capturing that in the relation would require proving normalization as we prove soundness, a complication we wanted to avoid, hence our relations stating the weaker co-termination property.

Note that 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 next two case studies), but it is a possibility and power that the designer of the model has.

Additionally, the reader might note that while System F terms do not own locations, the expression relation is remarkably similar to that of $\mathbf{L^3}$, and indeed, this allows terms in $\mathcal{E}[\![\tau]\!]_\rho$ to allocate and use mutable state, but by the time they reduce to values in $\mathcal{V}[\![\tau]\!]_\rho$, the state must have been freed, as values cannot own mutable state. This amounts to expressing a degree of extensional purity in our System F types. In that way, the state involved in the computation is ephemeral and harmless, as it could, for example, have been alternately encoded in a state monad. Note, of course, that we could have defined even more flexible notions of extensional purity—where, e.g., System F values could own heap locations, but could not depend on their values—but this would have complicated our logical relation, so we elected to avoid it.

At the bottom of the relation, we present syntax ($\Delta; \Gamma; \mathbf{\Delta}; \mathbf{\Gamma} \vdash \mathbf{e_1} \preceq \mathbf{e_2} : \boldsymbol{\tau}$) for expressing pairs of well-typed source terms being in the relation: that is, given closing substitutions (for types, locations, and terms), the compiled, closed terms are in the expression relation.

With the logical relation in hand, we can prove the convertibility soundness lemma:

Lemma 3.1 (Convertibility Soundness).

If $\tau \sim \boldsymbol{\tau}$, *then* $\forall (W, (\mathsf{H}_1, \mathsf{e}_1), (\mathsf{H}_2, \mathsf{e}_2)) \in \mathcal{E}[\![\tau]\!]_\rho.\ (W, (\mathsf{H}_1, C_{\tau \mapsto \boldsymbol{\tau}}(\mathsf{e}_1)), (\mathsf{H}_2, C_{\tau \mapsto \boldsymbol{\tau}}(\mathsf{e}_2))) \in \mathcal{E}[\![\boldsymbol{\tau}]\!]_\rho$

$\forall (W, (\mathsf{H}_1, \mathsf{e}_1), (\mathsf{H}_2, \mathsf{e}_2)) \in \mathcal{E}[\![\boldsymbol{\tau}]\!]_\rho.\ (W, (\mathsf{H}_1, C_{\boldsymbol{\tau} \mapsto \tau}(\mathsf{e}_1)), (\mathsf{H}_2, C_{\boldsymbol{\tau} \mapsto \tau}(\mathsf{e}_2))) \in \mathcal{E}[\![\tau]\!]_\rho$

Proof. By induction on the convertibility relation. See supplementary material. □

$$
\begin{array}{ll}
\mathcal{V}[\![\alpha]\!]_\rho & = \rho.\mathtt{F}(\alpha) \\
\mathcal{V}[\![\tau_1 \to \tau_2]\!]_\rho & = \{(\emptyset, \lambda \mathtt{x}_1.\mathtt{e}_1, \emptyset, \lambda \mathtt{x}_2.\mathtt{e}_2) \mid \\
& \quad \forall (\emptyset, \mathtt{v}_1, \emptyset, \mathtt{v}_2) \in \mathcal{V}[\![\tau_1]\!]_\rho. \\
& \quad (\emptyset, [\mathtt{x}_1 \mapsto \mathtt{v}_1]\mathtt{e}_1, \\
& \quad \emptyset, [\mathtt{x}_2 \mapsto \mathtt{v}_2]\mathtt{e}_2) \in \mathcal{E}[\![\tau_2]\!]_\rho\} \\
\hline
\mathcal{V}[\![\mathbf{unit}]\!]_\rho & = \{(\emptyset, (), \emptyset, ())\} \\
\mathcal{V}[\![\mathbf{bool}]\!]_\rho & = \{(\emptyset, \mathtt{b}, \emptyset, \mathtt{b}) \mid \mathtt{b} \in \{0,1\}\} \\
\mathcal{V}[\![\tau_1 \otimes \tau_2]\!]_\rho & = \{(\mathtt{H}_1^1 \uplus \mathtt{H}_1^2, (\mathtt{v}_1^1, \mathtt{v}_1^2), \mathtt{H}_2^1 \uplus \mathtt{H}_2^2, (\mathtt{v}_2^1, \mathtt{v}_2^2)) \mid \\
& \quad (\mathtt{H}_1^1, \mathtt{v}_1^1, \mathtt{H}_2^1, \mathtt{v}_2^1) \in \mathcal{V}[\![\tau_1]\!]_\rho \wedge \\
& \quad (\mathtt{H}_1^2, \mathtt{v}_1^2, \mathtt{H}_2^2, \mathtt{v}_2^2) \in \mathcal{V}[\![\tau_2]\!]_\rho\} \\
\mathcal{V}[\![!\tau]\!]_\rho & = \{(\emptyset, \mathtt{v}_1, \emptyset, \mathtt{v}_2) \mid (\emptyset, \mathtt{v}_1, \emptyset, \mathtt{v}_2) \in \mathcal{V}[\![\tau]\!]_\rho\} \\
\mathcal{V}[\![\tau_1 \multimap \tau_2]\!]_\rho & = \{(\mathtt{H}_1, \lambda \mathtt{x}_1.\mathtt{e}_1, \mathtt{H}_2, \lambda \mathtt{x}_2.\mathtt{e}_2) \mid \\
& \quad \forall (\mathtt{H}_1^v, \mathtt{v}_1, \mathtt{H}_2^v, \mathtt{v}_2) \in \mathcal{V}[\![\tau_1]\!]_\rho. \\
& \quad (\mathtt{H}_1 \uplus \mathtt{H}_1^v, [\mathtt{x}_1 \mapsto \mathtt{v}_1]\mathtt{e}_1, \\
& \quad \mathtt{H}_2 \uplus \mathtt{H}_2^v, [\mathtt{x}_2 \mapsto \mathtt{v}_2]\mathtt{e}_2) \in \mathcal{E}[\![\tau_2]\!]_\rho\}
\end{array}
$$

$$
\begin{array}{ll}
\mathcal{V}[\![\forall \alpha.\tau]\!]_\rho & = \{(\emptyset, \lambda\_.\mathtt{e}_1, \emptyset, \lambda\_.\mathtt{e}_2) \mid \forall R \in \mathit{RelT}. \\
& \quad (\emptyset, \mathtt{e}_1, \emptyset, \mathtt{e}_2) \in \mathcal{E}[\![\tau]\!]_{\rho[\mathtt{F}(\alpha) \mapsto R]}\} \\
\mathcal{V}[\![\langle \tau \rangle]\!]_\rho & = \{(\emptyset, \mathtt{v}_1, \emptyset, \mathtt{v}_2) \mid (\emptyset, \mathtt{v}_1, \emptyset, \mathtt{v}_2) \in \mathcal{V}[\![\tau]\!]_\rho\} \\
\hline
\mathcal{V}[\![\mathbf{ptr}\ \zeta]\!]_\rho & = \{(\emptyset, \ell_1, \emptyset, \ell_2) \mid \rho.\mathbf{L3}(\zeta) = (\ell_1, \ell_2)\} \\
\mathcal{V}[\![\mathbf{cap}\ \zeta\ \tau]\!]_\rho & = \{(\mathtt{H}_1 \uplus \{\ell_1 \mapsto \mathtt{v}_1\}, (), \mathtt{H}_2 \uplus \{\ell_2 \mapsto \mathtt{v}_2\}, ()) \mid \\
& \quad \rho.\mathbf{L3}(\zeta) = (\ell_1, \ell_2) \wedge \\
& \quad (\mathtt{H}_1, \mathtt{v}_1, \mathtt{H}_2, \mathtt{v}_2) \in \mathcal{V}[\![\tau]\!]_\rho\} \\
\mathcal{V}[\![\forall \zeta.\tau]\!]_\rho & = \{(\mathtt{H}_1, \lambda \mathtt{x}_1.\mathtt{e}_1, \mathtt{H}_2, \lambda \mathtt{x}_2.\mathtt{e}_2) \mid \\
& \quad (\mathtt{H}_1, [\mathtt{x}_1 \mapsto \ell_1]\mathtt{e}_1, \mathtt{H}_2, [\mathtt{x}_2 \mapsto \ell_2]\mathtt{e}_2) \\
& \quad \in \mathcal{E}[\![\tau]\!]_{\rho[\mathbf{L3}(\zeta) \mapsto (\ell_1, \ell_2)]}\} \\
\mathcal{V}[\![\exists \zeta.\tau]\!]_\rho & = \{(\mathtt{H}_1, (\ell_1, \mathtt{v}_1), \mathtt{H}_2, (\ell_2, \mathtt{v}_2) \mid \\
& \quad (\mathtt{H}_1, \mathtt{v}_1, \mathtt{H}_2, \mathtt{v}_2) \in \mathcal{V}[\![\tau]\!]_{\rho[\mathbf{L3}(\zeta) \mapsto (\ell_1, \ell_2)]}\}
\end{array}
$$

$$
\begin{array}{l}
\mathcal{E}[\![\tau]\!]_\rho = \{(\emptyset, \mathtt{e}_1, \emptyset, \mathtt{e}_2) \mid \forall \mathtt{v}_1, \mathtt{H}_{1+}, \mathtt{H}_{1*}. \\
\quad (\mathtt{H}_{1+}, \mathtt{e}_1) \xrightarrow{*} (\mathtt{H}_{1*}, \mathtt{v}_1) \nrightarrow \\
\quad \Longrightarrow \exists \mathtt{H}'_{1+}.\mathtt{H}_{1*} = \mathtt{H}_{1+} \uplus \mathtt{H}'_{1+} \wedge \\
\quad \exists \mathtt{v}_2.(\emptyset, \mathtt{v}_1, \emptyset, \mathtt{v}_2) \in \mathcal{V}[\![\tau]\!]_\rho \wedge \\
\quad \forall \mathtt{H}_{2+}.\exists \mathtt{H}'_{2+}.(\mathtt{H}_2, \mathtt{e}_2) \xrightarrow{*} (\mathtt{H}_{2+} \uplus \mathtt{H}'_{2+}, \mathtt{v}_2) \nrightarrow\}
\end{array}
$$

$$
\begin{array}{l}
\mathcal{E}[\![\tau]\!]_\rho = \{(\mathtt{H}_1, \mathtt{e}_1, \mathtt{H}_2, \mathtt{e}_2) \mid \forall \mathtt{v}_1, \mathtt{H}_{1+}, \mathtt{H}_{1*}. \\
\quad (\mathtt{H}_1 \uplus \mathtt{H}_{1+}, \mathtt{e}_1) \xrightarrow{*} (\mathtt{H}_{1*}, \mathtt{v}_1) \nrightarrow \\
\quad \Longrightarrow \exists \mathtt{H}'_1, \mathtt{H}'_{1+}.\mathtt{H}_{1*} = \mathtt{H}'_1 \uplus \mathtt{H}_{1+} \uplus \mathtt{H}'_{1+} \wedge \\
\quad \forall \mathtt{H}_{2+}.\exists \mathtt{v}_2, \mathtt{H}'_2, \mathtt{H}'_{2+}. \\
\quad (\mathtt{H}_2 \uplus \mathtt{H}_{2+}, \mathtt{e}_2) \xrightarrow{*} (\mathtt{H}'_2 \uplus \mathtt{H}_{2+} \uplus \mathtt{H}'_{2+}, \mathtt{v}_2) \nrightarrow \\
\quad \wedge (\mathtt{H}'_1, \mathtt{v}_1, \mathtt{H}'_2, \mathtt{v}_2) \in \mathcal{V}[\![\tau]\!]_\rho\}
\end{array}
$$

$$\gamma_{\mathrm{locs}}(\rho) \quad \equiv \quad \{\mathtt{x}_\zeta \mapsto (\ell_1, \ell_2) \mid \zeta \mapsto (\ell_1, \ell_2) \in \rho\}$$

$$
\begin{array}{l}
\Delta; \Gamma; \Delta; \Gamma \vdash \mathtt{e}_1 \preceq \mathtt{e}_2 : \tau \equiv \\
\quad \forall \rho, \gamma_{\mathbf{L}}, \gamma_\Gamma.\rho.\mathbf{L3} \in \mathcal{D}[\![\Delta]\!] \wedge \rho.\mathtt{F} \in \mathcal{D}[\![\Delta]\!] \wedge (\emptyset, \emptyset, \gamma_{\mathbf{L}}.\Gamma) \in \mathcal{G}[\![\Gamma]\!]_\rho \wedge \gamma_\Gamma \in \mathcal{G}[\![\Gamma]\!]_\rho \wedge\ \gamma_{\mathbf{L}}.\Delta = \gamma_{\mathrm{locs}}(\rho.\mathbf{L3}) \\
\qquad \Longrightarrow (\emptyset, \gamma_{\mathbf{L}}^1(\gamma_\Gamma^1(\mathtt{e}_1{}^+)), \emptyset, \gamma_{\mathbf{L}}^2(\gamma_\Gamma^2(\mathtt{e}_2{}^+))) \in \mathcal{E}[\![\tau]\!]_\rho \\
\Delta; \Gamma; \Delta; \Gamma \vdash \mathbf{e}_1 \preceq \mathbf{e}_2 : \boldsymbol{\tau} \equiv \\
\quad \forall \rho, \gamma_\Gamma, \gamma_{\mathbf{L}}, \mathtt{H}_1, \mathtt{H}_2.\rho.\mathtt{F} \in \mathcal{D}[\![\Delta]\!] \wedge \rho.\mathbf{L3} \in \mathcal{D}[\![\Delta]\!] \wedge \gamma_\Gamma \in \mathcal{G}[\![\Gamma]\!]_\rho \wedge (\mathtt{H}_1, \mathtt{H}_2, \gamma_{\mathbf{L}}.\Gamma) \in \mathcal{G}[\![\Gamma]\!]_\rho \wedge\ \gamma_{\mathbf{L}}.\Delta = \gamma_{\mathrm{locs}}(\rho.\mathbf{L3}) \\
\qquad \Longrightarrow (\mathtt{H}_1, \gamma_\Gamma^1(\gamma_{\mathbf{L}}^1(\mathbf{e}_1{}^+)), \mathtt{H}_2, \gamma_\Gamma^2(\gamma_{\mathbf{L}}^2(\mathbf{e}_2{}^+))) \in \mathcal{E}[\![\boldsymbol{\tau}]\!]_\rho
\end{array}
$$

Fig. 9 . Logical Relation for System F and $\mathbf{L}^3$.

In particular, we prove that our conversions above between $\mathbf{L}^3$ booleans and System F Church booleans (described above) are sound. We also show that $\tau_1 \to \tau_2 \sim \mathbf{!(!}\boldsymbol{\tau}_1 \multimap \boldsymbol{\tau}_2\mathbf{)}$ assuming $\tau_1 \sim \boldsymbol{\tau}_1$ and $\tau_2 \sim \boldsymbol{\tau}_2$. As before, we then prove compatibility lemmas for the all of the typing rules of the multi-language, after which we can prove the fundamental property and type safety:

Theorem 3.2 (Fundamental Property).
If $\Delta; \Gamma; \Delta; \Gamma \vdash \mathtt{e} : \tau$ *then* $\Delta; \Gamma; \Delta; \Gamma \vdash \mathtt{e} \preceq \mathtt{e} : \tau$ *and if* $\Delta; \Gamma; \Delta; \Gamma \vdash \mathbf{e} : \boldsymbol{\tau}$ *then* $\Delta; \Gamma; \Delta; \Gamma \vdash \mathbf{e} \preceq \mathbf{e} : \boldsymbol{\tau}$.

Theorem 3.3 (Type Safety for System F). *If* $\cdot; \cdot; \cdot; \cdot \vdash \mathtt{e} : \tau$, *then for any heap* H, *if* $(\mathtt{H}, \mathtt{e}^+) \xrightarrow{*} (\mathtt{H}', \mathtt{e}')$, *either there exist* $\mathtt{H}'', \mathtt{e}''$ *such that* $(\mathtt{H}', \mathtt{e}') \to (\mathtt{H}'', \mathtt{e}'')$ *or* $\mathtt{e}'$ *is a value.*

Theorem 3.4 (Type Safety for $\mathbf{L}^3$). *If* $\cdot; \cdot; \cdot; \cdot \vdash \mathbf{e} : \boldsymbol{\tau}$, *then for any heap* H, *if* $(\mathtt{H}, \mathbf{e}^+) \xrightarrow{*} (\mathtt{H}', \mathtt{e}')$, *either there exist* $\mathtt{H}'', \mathtt{e}''$ *such that* $(\mathtt{H}', \mathtt{e}') \to (\mathtt{H}'', \mathtt{e}'')$ *or* $\mathtt{e}'$ *is a value.*

Discussion. While we showed how to handle universal types, handling existential types is another question. With our existing "foreign type" mechanism, we can support defining data structures and operations over them and passing both. For example, we could pass an expression of type $\langle \mathit{int} \rangle \times \langle \mathit{int} \rangle \to \langle \mathit{int} \rangle \times \langle \mathit{int} \rangle \to \mathit{int}$, for a counter defined as an integer. That provides some degree of abstraction, but doesn't, for example, disallow passing the $\langle \mathit{int} \rangle$ back to some other code

that expects that type. We could, however, in the language with existential types, pack that to $\exists\alpha.\alpha \times \alpha \to \alpha \times \alpha \to int$.

More interesting is the question when both languages have polymorphism. In that case, if we wanted to convert abstract types, we would need to generalize our convertibility rules to handle open types, i.e., $\Delta \vdash \tau \sim \tau'$. If the interpretation of type variables were the same in both languages (i.e., in our model this would mean that both were drawn from the same relation), this would be sufficient. If, however, the interpretation of type variables were different in the two languages (see, e.g., *UnrTyp* in Fig. 17 for our final case study in §5), we would need, in our source type systems, some form of bounded polymorphism in order to restrict the judgment to variables that were equivalent. Otherwise, it would be impossible to prove convertibility rules sound.

4 AFFINE & UNRESTRICTED

In our third case study, we consider an affine language, Affi, interacting with an unrestricted one, MiniML. In this case, unlike the previous one, the substructural features from Affi can be enforced dynamically, which means we can allow more flexible but sound mixing than would be possible if the types were linear. In particular, we adopt the classic technique, described in [63], where affine resources are protected behind thunks with stateful flags that indicate failure on a second force.

Languages. We present the syntax of MiniML and Affi and the static semantics of Affi in Fig. 10; we elide the static semantics of MiniML, which are standard. As in the previous case studies, we will support open terms across language boundaries, and thus need to carry environments for both languages. However, while with System F and $\mathbf{L}^3$, the only sound option was to disallow linear resources from crossing boundaries, in this case study, we can protect affine resources. That means that our affine environments Ω need to be split, even within MiniML, to respect the affine invariant.

Rather than splitting environments from the other language, we instead adopt a generic method for threading these types of contexts, where each judgment has an input and output context from the foreign language. i.e., a typing rule for Affi will have the shape $\mathfrak{C}; \Gamma; \Omega \vdash \mathbf{e} : \tau \rightsquigarrow \mathfrak{C}'$, where $\mathfrak{C}$ stands for all of the static typing environments needed by MiniML. So, $\mathfrak{C} = \Gamma; \Omega$ and $\mathfrak{C} = \Delta; \Gamma$. Since each judgment has both input and output, and we thread this through subterms, this allows not only unrestricted judgments (which do not change the output) but substructural ones (which do change the output), in a reasonably lightweight way.

Returning to Fig. 10, we can see that Affi maintains an affine environment, introduced by lambda and tensor-destructuring let, that it splits across subterms, but it does not require that all bindings be used, as we can see in the terminal rules for variables (affine **a** and unrestricted **x**), unit, booleans, etc.

Our target language is the same untyped lambda calculus, LCVM (Fig. 7), from our previous case study. Our compilers, presented in Fig. 11, are primarily interesting in the cases that address affine bindings; otherwise, they do standard type erasure for polymorphic types, etc. We use a compiler macro, thunk(·), which expands to a nullary function closing over a freshly-allocated reference—called a *flag*—initialized to 1. When called, this function fails if the flag is set to 0. Otherwise, it sets the flag to 0 and returns the macro's argument. Throughout the paper, we will use the constants UNUSED and USED for 1 and 0. We use thunk(·) when introducing affine bindings, and then we compile uses of affine variable to expressions that force the thunk. Unrestricted Affi variables **x** and variables from MiniML are unaffected by this strategy.

Convertibility. We define convertibility relations and conversions for Affi and MiniML in Fig. 13. In it we define base type conversions between **unit** and unit, **bool** and int, tensors and pairs, and between $\to$ and $\multimap$. The last is most interesting and challenging. Our compiler is designed to support affine code being mixed directly with unrestricted code. Intuitively, an affine function

$$
\begin{array}{llll}
\textsc{Affi} & \text{Type } \tau & ::= & \mathbf{unit} \mid \mathbf{bool} \mid \mathbf{int} \mid \tau \multimap \tau \mid !\tau \mid \tau \& \tau \mid \tau \otimes \tau \\
& \text{Expression } \mathbf{e} & ::= & () \mid \mathbf{true} \mid \mathbf{false} \mid \mathbf{n} \mid \mathbf{x} \mid \mathbf{a} \mid \lambda \mathbf{a} : \tau.\mathbf{e} \mid \mathbf{e}\ \mathbf{e} \mid (\!|\mathbf{e}|\!)_\tau \mid !\mathbf{e} \\
& & & \mid \mathbf{let}\ !\mathbf{x} = \mathbf{e}\ \mathbf{in}\ \mathbf{e}' \mid \langle \mathbf{e}, \mathbf{e}' \rangle \mid \mathbf{e}.1 \mid \mathbf{e}.2 \mid (\mathbf{e}, \mathbf{e}) \mid \mathbf{let}\ (\mathbf{a}, \mathbf{a}') = \mathbf{e}\ \mathbf{in}\ \mathbf{e}' \\
& \text{Value } \mathbf{v} & ::= & () \mid \lambda \mathbf{a} : \tau.\mathbf{e} \mid !\mathbf{e} \mid \langle \mathbf{e}, \mathbf{e}' \rangle \mid (\mathbf{v}, \mathbf{v}') \\
\texttt{MiniML} & \text{Type } \tau & ::= & \texttt{unit} \mid \texttt{int} \mid \tau \times \tau \mid \tau + \tau \mid \tau \to \tau \mid \forall \alpha.\tau \mid \alpha \mid \texttt{ref}\ \tau \\
& \text{Expression } \mathbf{e} & ::= & () \mid \mathbb{Z} \mid \mathbf{x} \mid (\mathbf{e}, \mathbf{e}) \mid \texttt{fst}\ \mathbf{e} \mid \texttt{snd}\ \mathbf{e} \mid \texttt{inl}\ \mathbf{e} \mid \texttt{inr}\ \mathbf{e} \mid \texttt{match}\ \mathbf{e}\ \mathbf{x}\{\mathbf{e}\}\ \mathbf{y}\{\mathbf{e}\} \\
& & & \mid \lambda x : \tau.\mathbf{e} \mid \mathbf{e}\ \mathbf{e} \mid \Lambda \alpha.\mathbf{e} \mid \mathbf{e}[\tau] \mid \texttt{ref}\ \mathbf{e} \mid !\mathbf{e} \mid \mathbf{e} := \mathbf{e} \mid (\!|\mathbf{e}|\!)_\tau
\end{array}
$$

$$
\frac{\mathbf{a} : \tau \in \Omega}{\mathfrak{C}; \Gamma; \Omega \vdash \mathbf{a} : \tau \rightsquigarrow \mathfrak{C}} \qquad \frac{\mathbf{x} : \tau \in \Gamma}{\mathfrak{C}; \Gamma; \Omega \vdash \mathbf{x} : \tau \rightsquigarrow \mathfrak{C}} \qquad \frac{}{\mathfrak{C}; \Gamma; \Omega \vdash \mathbf{n} : \mathbf{int} \rightsquigarrow \mathfrak{C}}
$$

$$
\frac{\Gamma; \Omega; \Delta; \Gamma \vdash \mathbf{e} : \tau \rightsquigarrow \Gamma; \Omega' \quad \_ : \tau \sim \tau}{\Delta; \Gamma; \Gamma; \Omega \vdash (\!|\mathbf{e}|\!)_\tau : \tau \rightsquigarrow \Delta; \Gamma} \qquad \frac{\Omega = \Omega_1 \uplus \Omega_2 \quad \mathfrak{C}_1; \Gamma; \Omega_1 \vdash \mathbf{e}_1 : \tau_1 \multimap \tau_2 \rightsquigarrow \mathfrak{C}_2 \quad \mathfrak{C}_2; \Gamma; \Omega_2 \vdash \mathbf{e}_2 : \tau_1 \rightsquigarrow \mathfrak{C}_3}{\mathfrak{C}_1; \Gamma; \Omega \vdash \mathbf{e}_1\ \mathbf{e}_2 : \tau_2 \rightsquigarrow \mathfrak{C}_3}
$$

$$
\frac{\mathfrak{C}; \Gamma; \cdot \vdash \mathbf{e} : \tau \rightsquigarrow \mathfrak{C}'}{\mathfrak{C}; \Gamma; \cdot \vdash !\mathbf{e} : !\tau \rightsquigarrow \mathfrak{C}'} \qquad \frac{\Omega = \Omega_1 \uplus \Omega_2 \quad \mathfrak{C}_1; \Gamma; \Omega_1 \vdash \mathbf{e} : !\tau \rightsquigarrow \mathfrak{C}_2 \quad \mathfrak{C}_2; \Gamma[\mathbf{x} := \tau]; \Omega_2 \vdash \mathbf{e}' : \tau' \rightsquigarrow \mathfrak{C}_3}{\mathfrak{C}_1; \Gamma; \Omega \vdash \mathbf{let}\ !\mathbf{x} = \mathbf{e}\ \mathbf{in}\ \mathbf{e}' \rightsquigarrow \mathfrak{C}_3}
$$

$$
\frac{\mathfrak{C}_1; \Gamma; \Omega \vdash \mathbf{e}_1 : \tau_1 \rightsquigarrow \mathfrak{C}_2 \quad \mathfrak{C}_2; \Gamma; \Omega \vdash \mathbf{e}_2 : \tau_2 \rightsquigarrow \mathfrak{C}_3}{\mathfrak{C}_1; \Gamma; \Omega \vdash \langle \mathbf{e}_1, \mathbf{e}_2 \rangle : \tau_1 \& \tau_2 \rightsquigarrow \mathfrak{C}_3} \qquad \frac{\mathfrak{C}; \Gamma; \Omega \vdash \mathbf{e} : \tau_1 \& \tau_2 \rightsquigarrow \mathfrak{C}'}{\mathfrak{C}; \Gamma; \Omega \vdash \mathbf{e}.1 : \tau_1 \rightsquigarrow \mathfrak{C}'}
$$

$$
\frac{\mathfrak{C}; \Gamma; \Omega \vdash \mathbf{e} : \tau_1 \& \tau_2 \rightsquigarrow \mathfrak{C}'}{\mathfrak{C}; \Gamma; \Omega \vdash \mathbf{e}.2 : \tau_2 \rightsquigarrow \mathfrak{C}'} \qquad \frac{\mathfrak{C}; \Gamma; \Omega[\mathbf{a} := \tau_1] \vdash \mathbf{e} : \tau_2 \rightsquigarrow \mathfrak{C}'}{\mathfrak{C}; \Gamma; \Omega \vdash \lambda \mathbf{a} : \tau_1.\mathbf{e} : \tau_1 \multimap \tau_2 \rightsquigarrow \mathfrak{C}'}
$$

$$
\frac{\Omega = \Omega_1 \uplus \Omega_2 \quad \mathfrak{C}_1; \Gamma; \Omega_1 \vdash \mathbf{e}_1 : \tau_1 \rightsquigarrow \mathfrak{C}_2 \quad \mathfrak{C}_2; \Gamma; \Omega_2 \vdash \mathbf{e}_2 : \tau_2 \rightsquigarrow \mathfrak{C}_3}{\mathfrak{C}_1; \Gamma; \Omega \vdash (\mathbf{e}_1, \mathbf{e}_2) : \tau_1 \otimes \tau_2 \rightsquigarrow \mathfrak{C}_3}
$$

$$
\frac{\Omega = \Omega_1 \uplus \Omega_2 \quad \mathfrak{C}_1; \Gamma; \Omega_1 \vdash \mathbf{e} : \tau_1 \otimes \tau_2 \rightsquigarrow \mathfrak{C}_2 \quad \mathfrak{C}_2; \Gamma; \Omega_2[\mathbf{a} := \tau_1, \mathbf{a}' := \tau_1] \vdash \mathbf{e}' : \tau' \rightsquigarrow \mathfrak{C}_3}{\mathfrak{C}_1; \Gamma; \Omega \vdash \mathbf{let}\ (\mathbf{a}, \mathbf{a}') = \mathbf{e}\ \mathbf{in}\ \mathbf{e}' : \tau' \rightsquigarrow \mathfrak{C}_3}
$$

Fig. 10 . Syntax for MiniML; syntax & selected statics for Affi.

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 $\tau_1 \to \tau_2$ (not even for some convertible argument/return types), but rather to $(\texttt{unit} \to \tau_1) \to \tau_2$. 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$(\cdot)$.

Semantic Model. To reason about this system, we present our step indexed binary logical relation in Fig. 12. Although there is some overlap with the previous case studies and prior work, the treatment of affine resources is novel. In our worlds Ws, we keep the step index, a standard heap typing Ψ, which maintains a simple bijection between locations of the two programs (which doesn't support sophisticated reasoning about equivalence in the presence of "local state" [3] but suffices for soundness), but also an affine flag store, Θ, which maintains a bijection on flags (locations) that track whether affine variables have been used. This is a subset of the heap, disjoint from Ψ, and restricted by the model to only contain either 0 or 1, which for convenience we write using the constants USED and UNUSED. Our world extension relation, $W \sqsubseteq W'$, shows that flags cannot

$()$	$\rightsquigarrow$	$()$	$()$	$\rightsquigarrow$	$()$
$\mathbb{Z}$	$\rightsquigarrow$	$\mathbb{Z}$	$\mathbf{n}$	$\rightsquigarrow$	n
			$\mathbf{true}$	$\rightsquigarrow$	0
			$\mathbf{false}$	$\rightsquigarrow$	1
$\mathtt{x}$	$\rightsquigarrow$	x	$\mathbf{x}$	$\rightsquigarrow$	x
			$\mathbf{a}$	$\rightsquigarrow$	a ()
$\lambda \mathtt{x} : \tau.\mathtt{e}$	$\rightsquigarrow$	λx.{$\mathtt{e}^+$}	$\boldsymbol{\lambda \mathtt{a} : \tau.\mathtt{e}}$	$\rightsquigarrow$	λa.{$\mathtt{e}^+$}
$\mathtt{e}_1\ \mathtt{e}_2$	$\rightsquigarrow$	$\mathtt{e}_1^+\ \mathtt{e}_2^+$	$\mathbf{e_1\ e_2}$	$\rightsquigarrow$	$\mathbf{e_1^+}$ (let x = $\mathbf{e_2^+}$ in thunk(x))
$(\mathtt{e}_1, \mathtt{e}_2)$	$\rightsquigarrow$	$(\mathtt{e}_1^+, \mathtt{e}_2^+)$	$\mathbf{(e_1, e_2)}$	$\rightsquigarrow$	$(\mathbf{e_1^+}, \mathbf{e_2^+})$
$\mathtt{fst\ e}$	$\rightsquigarrow$	fst $\mathtt{e}^+$	$\mathbf{e.1}$	$\rightsquigarrow$	(fst $\mathbf{e}^+$) ()
$\mathtt{snd\ e}$	$\rightsquigarrow$	snd $\mathtt{e}^+$	$\mathbf{e.2}$	$\rightsquigarrow$	(snd $\mathbf{e}^+$) ()
$\mathtt{inl\ e}$	$\rightsquigarrow$	inl $\mathtt{e}^+$	$\mathbf{\langle e_1, e_2 \rangle}$	$\rightsquigarrow$	$(\lambda\_.\{\mathbf{e_1^+}\}, \lambda\_.\{\mathbf{e_2^+}\})$
$\mathtt{inr\ e}$	$\rightsquigarrow$	inr $\mathtt{e}^+$			
$\mathtt{match\ e}$ $\mathtt{x}\{\mathtt{e}_1\}\ \mathtt{y}\{\mathtt{e}_2\}$	$\rightsquigarrow$	match $\mathtt{e}^+$ $\mathtt{x}\{\mathtt{e}_1^+\}\ \mathtt{y}\{\mathtt{e}_2^+\}$	$\mathbf{let\ (a_1, a_2) = e_1}$ $\mathbf{in\ e_2}$	$\rightsquigarrow$	let $\mathtt{x}_{\mathrm{fresh}} = \mathbf{e_1^+}$, $\mathtt{x}'_{\mathrm{fresh}}$ = fst $\mathtt{x}_{\mathrm{fresh}}$, $\mathtt{x}''_{\mathrm{fresh}}$ = snd $\mathtt{x}_{\mathrm{fresh}}$, $\mathtt{a}_1$ = thunk($\mathtt{x}'_{\mathrm{fresh}}$), $\mathtt{a}_2$ = thunk($\mathtt{x}''_{\mathrm{fresh}}$) in $\mathbf{e_2^+}$
$\Lambda\alpha.\mathtt{e}$	$\rightsquigarrow$	$\lambda\_.\{\mathtt{e}^+\}$			
$\mathtt{e}[\tau]$	$\rightsquigarrow$	$\mathtt{e}^+$ ()	$\mathbf{!e}$	$\rightsquigarrow$	$\mathbf{e}^+$
$\mathtt{ref\ e}$	$\rightsquigarrow$	ref $\mathtt{e}^+$	$\mathbf{let\ !x = e_1}$ $\mathbf{in\ e_2}$	$\rightsquigarrow$	let x = $\mathbf{e_1^+}$ in $\mathbf{e_2^+}$
$\mathtt{!e}$	$\rightsquigarrow$	$\mathtt{!e}^+$			
$\mathtt{e}_1 := \mathtt{e}_2$	$\rightsquigarrow$	$\mathtt{e}_1^+ := \mathtt{e}_2^+$			
$(\!|\mathtt{e}|\!)_\tau$	$\rightsquigarrow$	$C_{\tau \mapsto \tau}(\mathtt{e}^+)$	$(\!|\mathtt{e}|\!)_\tau$	$\rightsquigarrow$	$C_{\tau \mapsto \tau}(\mathtt{e}^+)$

thunk(e) $\triangleq$ let $\mathtt{r}_{\mathrm{fresh}}$ = ref UNUSED in $\lambda\_.${if !$\mathtt{r}_{\mathrm{fresh}}$ {fail CONV} {$\mathtt{r}_{\mathrm{fresh}}$:= USED; e}} where USED = 0 and UNUSED = 1

Fig. 11 . Compilers for MiniML and **Affi**.

be removed from Θ, and once a flag is marked as USED, it cannot be marked UNUSED. With this setup, our expression relation $\mathcal{E}[\![\tau]\!]_\rho$ is quite ordinary, as the described structure is entirely about characterizing the heaps that programs will run in, not about how they will run. Note that $\mathcal{E}[\![\tau]\!]_\rho$ allows for the possibility of $\mathtt{e}_1$ raising a conversion error (fail CONV) at runtime.

Finally, while most of the cases in our value relation are standard, the affine arrow $\multimap$ is novel. A pair of functions $\lambda\mathtt{a}.\mathtt{e}_1$ and $\lambda\mathtt{a}.\mathtt{e}_2$ are related if, given a pair of arguments $\mathtt{v}_1$ and $\mathtt{v}_2$ related at a future world W', we get related results in W' extended with a new entry in the flag store $W'.\Theta$ for some fresh locations ℓ_1, ℓ_2. Importantly, what we substitute into the body is *not* $\mathtt{v}_1$ and $\mathtt{v}_2$, but rather wrapped forms, guard($\mathtt{v}_1, \ell_1$) and guard($\mathtt{v}_2, \ell_2$), each of which closes over the fresh location in the flag store and thus ensures that the argument is not used more than once. This makes sense, since in the target, we expect affine variables to be thunks, and will force them upon use.

With the logical relation in hand, we can prove the following:

THEOREM 4.1 (CONVERTIBILITY SOUNDNESS).
If $\tau \sim \boldsymbol{\tau}$ *then* $\forall\ (W, \mathtt{e}_1, \mathtt{e}_2) \in \mathcal{E}[\![\tau]\!].\ \Longrightarrow\ (W, C_{\tau \mapsto \boldsymbol{\tau}}(\mathtt{e}_1), C_{\tau \mapsto \boldsymbol{\tau}}(\mathtt{e}_2)) \in \mathcal{E}[\![\boldsymbol{\tau}]\!]$.
$\forall\ (W, \mathtt{e}_1, \mathtt{e}_2) \in \mathcal{E}[\![\boldsymbol{\tau}]\!].\ \Longrightarrow\ (W, C_{\boldsymbol{\tau} \mapsto \tau}(\mathtt{e}_1), C_{\boldsymbol{\tau} \mapsto \tau}(\mathtt{e}_2)) \in \mathcal{E}[\![\tau]\!]$.

THEOREM 4.2 (FUNDAMENTAL PROPERTY).
If $\Gamma; \Omega; \Delta; \Gamma \vdash \mathtt{e} : \tau \rightsquigarrow \Gamma'; \Omega'$ *then* $\Gamma; \Omega; \Delta; \Gamma \vdash \mathtt{e} \preceq \mathtt{e} : \tau \rightsquigarrow \Gamma'; \Omega'$ *and if* $\Delta; \Gamma; \Gamma; \Omega \vdash \mathbf{e} : \boldsymbol{\tau} \rightsquigarrow \Delta'; \Gamma'$ *then* $\Delta; \Gamma; \Gamma; \Omega \vdash \mathbf{e} \preceq \mathbf{e} : \boldsymbol{\tau} \rightsquigarrow \Delta'; \Gamma'$.

THEOREM 4.3 (TYPE SAFETY FOR MiniML).
For any MiniML *term* e *where* $\cdot; \cdot; \cdot; \cdot \vdash \mathtt{e} : \tau \rightsquigarrow \cdot; \cdot$ *and for any heap* H, *if* $\langle \mathtt{H}, \mathtt{e}^+ \rangle \xrightarrow{*} \langle \mathtt{H}', \mathtt{e}' \rangle$, *then either* $\mathtt{e}' = \mathtt{fail}$ CONV, $\mathtt{e}'$ *is a value, or there exist* $\mathtt{H}'', \mathtt{e}''$ *such that* $\langle \mathtt{H}', \mathtt{e}' \rangle \rightarrow \langle \mathtt{H}'', \mathtt{e}'' \rangle$.

THEOREM 4.4 (TYPE SAFETY FOR **Affi**).

$$
\begin{array}{lll}
Atom_n & \triangleq \{(W, e_1, e_2) \mid W \in World_n\} \quad AtomVal_n \triangleq \{(W, v_1, v_2) \in Atom_n\} \quad AtomVal \triangleq \bigcup_n AtomVal_n \\
World_n & \triangleq \{(k, \Psi, \Theta) \mid k < n \wedge \Psi \subset HeapTy_k \wedge \mathrm{dom}(\Psi)\#\mathrm{dom}(\Theta)\} \\
HeapTy_n & \triangleq \{(\ell_1, \ell_2) \mapsto Typ_n, \ldots\} \\
Typ_n & \triangleq \{R \in 2^{AtomVal_n} \mid \forall (W, v_1, v_2) \in R.\, \forall W'.\ W \sqsubseteq W' \implies (W', v_1, v_2) \in R\} \\
Typ & \triangleq \{R \in 2^{AtomVal} \mid \forall k. \lfloor R \rfloor_k \in Typ_k\} \\
\Theta & \triangleq \{(\ell_1, \ell_2) \mapsto \{\textsc{used}, \textsc{unused}\}, \ldots\} \text{ where } \textsc{used} = 0 \text{ and } \textsc{unused} = 1 \\
W \sqsubseteq W' & \triangleq \ W'.k \leq W.k \wedge\ \forall (\ell_1, \ell_2) \in \mathrm{dom}(W.\Psi).\lfloor W.\Psi(\ell_1, \ell_2) \rfloor_{W'.k} = W'.\Psi(\ell_1, \ell_2) \\
 & \quad \wedge \forall (\ell_1, \ell_2) \in \mathrm{dom}(W.\Theta).(\ell_1, \ell_2) \in \mathrm{dom}(W'.\Theta) \wedge (W.\Theta(\ell_1, \ell_2) = \textsc{used} \implies W'.\Theta(\ell_1, \ell_2) = \textsc{used}) \\
H_1, H_2 : W & \triangleq (\forall (\ell_1, \ell_2) \mapsto R \in W.\Psi.\ (\triangleright W, H_1(\ell_1), H_2(\ell_2)) \in R) \wedge\ (\forall (\ell_1, \ell_2) \mapsto b \in W.\Theta. H_1(\ell_1) = H_2(\ell_2) = b)
\end{array}
$$

$$
\begin{array}{lcl}
\mathcal{V}[\![\texttt{unit}]\!]_\rho & = & \{(W, (), ())\} \\
\mathcal{V}[\![\texttt{int}]\!]_\rho & = & \{(W, n, n) \mid n \in \mathbb{Z}\} \\
\mathcal{V}[\![\tau_1 \times \tau_2]\!]_\rho & = & \{(W, (v_{1a}, v_{2a}), (v_{1b}, v_{2b})) \mid (W, v_{1a}, v_{1b}) \in \mathcal{V}[\![\tau_1]\!]_\rho \wedge (W, v_{2a}, v_{2b}) \in \mathcal{V}[\![\tau_2]\!]_\rho\} \\
\mathcal{V}[\![\tau_1 + \tau_2]\!]_\rho & = & \{(W, \mathrm{inl}\ v_1, \mathrm{inl}\ v_2) \mid (W, v_1, v_2) \in \mathcal{V}[\![\tau_1]\!]_\rho\} \cup \{(W, \mathrm{inr}\ v_1, \mathrm{inr}\ v_2) \mid (W, v_1, v_2) \in \mathcal{V}[\![\tau_2]\!]_\rho\} \\
\mathcal{V}[\![\tau_1 \to \tau_2]\!]_\rho & = & \{(W, \lambda x.\{e_1\}, \lambda x.\{e_2\}) \mid \forall v_1\ v_2\ W'. W \sqsubset W' \wedge (W', v_1, v_2) \in \mathcal{V}[\![\tau_1]\!]_\rho \\
 & & \implies (W', [x \mapsto v_1]e_1, [x \mapsto v_2]e_2) \in \mathcal{E}[\![\tau_2]\!]_\rho\} \\
\mathcal{V}[\![\texttt{ref}\ \tau]\!]_\rho & = & \{(W, \ell_1, \ell_2) \mid W.\Psi(\ell_1, \ell_2) = \lfloor \mathcal{V}[\![\tau]\!]_\rho \rfloor_{W.k}\} \\
\mathcal{V}[\![\forall \alpha.\tau]\!]_\rho & = & \{(W, \lambda\_.e_1, \lambda\_.e_2) \mid \forall R \in Typ,\ W'. W \sqsubset W' \implies (W', e_1, e_2) \in \mathcal{E}[\![\tau]\!]_{\rho[\alpha \mapsto R]}\} \\
\mathcal{V}[\![\alpha]\!]_\rho & = & \rho(\alpha) \\
\mathcal{V}[\![\texttt{unit}]\!]. & = & \{(W, (), ())\} \\
\mathcal{V}[\![\texttt{bool}]\!]_\rho & = & \{(W, 0, 0)\} \cup \{(W, n_1, n_2) \mid n_1 \neq 0 \wedge n_2 \neq 0\} \\
\mathcal{V}[\![\texttt{int}]\!]. & = & \{(W, n, n) \mid n \in \mathbb{Z}\} \\
\mathcal{V}[\![\tau_1 \multimap \tau_2]\!]. & = & \{(W, \lambda\ \mathrm{a}.e_1, \lambda\ \mathrm{a}.e_2) \mid \forall v_1\ v_2\ W'\ \ell_1\ \ell_2. \\
 & & W \sqsubset W' \wedge (W', v_1, v_2) \in \mathcal{V}[\![\tau_1]\!]. \wedge (\ell_1, \ell_2) \notin \mathrm{dom}(W'.\Psi) \cup \mathrm{dom}(W'.\Theta) \\
 & & \implies ((W'.k, W'.\Psi, W'.\Theta \uplus (\ell_1, \ell_2) \mapsto \textsc{unused}), \\
 & & [\mathrm{a} \mapsto \mathrm{guard}(v_1, \ell_1)]e_1, [\mathrm{a} \mapsto \mathrm{guard}(v_2, \ell_2)]e_2) \in \mathcal{E}[\![\tau_2]\!].\} \\
\mathcal{V}[\![!\tau]\!]. & = & \{(W, v_1, v_2) \mid (W, v_1, v_2) \in \mathcal{V}[\![\tau]\!].\} \\
\mathcal{V}[\![\tau_1 \otimes \tau_2]\!]. & = & \{(W, (v_{1a}, v_{2a}), (v_{1b}, v_{2b})) \mid (W, v_{1a}, v_{1b}) \in \mathcal{V}[\![\tau_1]\!]. \wedge (W, v_{2a}, v_{2b}) \in \mathcal{V}[\![\tau_2]\!].\} \\
\mathcal{V}[\![\tau_1 \& \tau_2]\!]. & = & \{(W, (\lambda\_.\{e_{1a}\}, \lambda\_.\{e_{2a}\}), (\lambda\_.\{e_{1b}\}, \lambda\_.\{e_{2b}\})) \\
 & & \mid (W, e_{1a}, e_{1b}) \in \mathcal{E}[\![\tau_1]\!]. \wedge (W, e_{2a}, e_{2b}) \in \mathcal{E}[\![\tau_2]\!].\}
\end{array}
$$

$$\mathrm{guard}(e, \ell) \triangleq \lambda\_.\{\mathrm{if}\ !\ell\ \{\mathrm{fail}\ \textsc{Conv}\}\ \{\ell := \textsc{used}; e\}\}$$

$$
\begin{array}{l}
\mathcal{E}[\![\tau]\!]_\rho = \{(W, e_1, e_2) \mid \mathrm{freevars}(e_1) = \mathrm{freevars}(e_2) = \emptyset \wedge \\
\quad \forall H_1, H_2 : W,\ e_1',\ H_1',\ j < W.k.\ \langle H_1, e_1\rangle \xrightarrow{j} \langle H_1', e_1'\rangle \not\to \\
\quad \implies e_1' = \mathrm{fail}\ \textsc{Conv} \vee (\exists v_2 H_2' W'. \langle H_2, e_2\rangle \xrightarrow{*} \langle H_2', v_2\rangle \wedge W \sqsubseteq W' \wedge H_1', H_2' : W' \wedge (W', e_1', v_2) \in \mathcal{V}[\![\tau]\!]_\rho)\}
\end{array}
$$

Fig. 12 . Logical Relation for MiniML and Affi.

For any Affi term e *where* $\cdot;\cdot;\cdot;\cdot \vdash e : \tau \rightsquigarrow \cdot;\cdot$ *and for any heap* H, *if* $\langle H, e^+\rangle \xrightarrow{*} \langle H', e'\rangle$, *then we know from the logical relation that either* $e' = \mathrm{fail}\ \textit{Conv}$, e' *is a value, or there exist* H'', e'' *such that* $\langle H', e'\rangle \to \langle H'', e''\rangle$.

Examples. Using our conversions, we investigate several small examples, presented (with their compilations) in Figure 14. Program $P1$ converts a MiniML function that projects the first element of a pair of integers to Affi and applies it to (true, false), producing true successfully. By contrast, $P1^\dagger$ tries to use the pair twice (sites of errors are highlighted), which once converted to Affi, is a violation of the type invariant, and thus this produces a runtime error, which we can see in the compiled code will occur at the second invocation of x (), which contains the contents of a thunk$(\cdot)$.

Program $P2$ defines an affine function (and immediately applies it) that binds a variable a in Affi, then uses it (inside a closure) in MiniML, returning a pair made up of that variable and a

$$\frac{}{C_{\texttt{unit}\mapsto\texttt{unit}}, C_{\texttt{unit}\mapsto\texttt{unit}} : \texttt{unit} \sim \texttt{unit}} \qquad \frac{}{C_{\texttt{int}\mapsto\texttt{bool}}, C_{\texttt{bool}\mapsto\texttt{int}} : \texttt{int} \sim \texttt{bool}}$$

$$\frac{C_{\tau_1\mapsto\tau_1}, C_{\tau_1\mapsto\tau_1} : \tau_1 \sim \tau_1 \quad C_{\tau_2\mapsto\tau_2}, C_{\tau_2\mapsto\tau_2} : \tau_2 \sim \tau_2}{C_{\tau_1\otimes\tau_2\mapsto\tau_1\times\tau_2}, C_{\tau_1\times\tau_2\mapsto\tau_1\otimes\tau_2} : \tau_1 \otimes \tau_2 \sim \tau_1 \times \tau_2}$$

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

$$C_{\texttt{unit}\mapsto\texttt{unit}}(\texttt{e}) \triangleq C_{\texttt{int}\mapsto\texttt{bool}}(\texttt{e}) \triangleq C_{\texttt{unit}\mapsto\texttt{unit}}(\texttt{e}) \triangleq \texttt{e} \quad C_{\texttt{bool}\mapsto\texttt{int}}(\texttt{e}) \triangleq \texttt{if e 0 1}$$

$$\begin{aligned}
&C_{\tau_1\otimes\tau_2\mapsto\tau_1\times\tau_2}(\texttt{e}) &&\triangleq \texttt{let x = e in } (C_{\tau_1\mapsto\tau_1}(\texttt{fst x}), C_{\tau_2\mapsto\tau_2}(\texttt{snd x}))\\
&C_{\tau_1\times\tau_2\mapsto\tau_1\otimes\tau_2}(\texttt{e}) &&\triangleq \texttt{let x = e in } (C_{\tau_1\mapsto\tau_1}(\texttt{fst x}), C_{\tau_2\mapsto\tau_2}(\texttt{snd x}))\\
&C_{\tau_1\multimap\tau_2\mapsto(\texttt{unit}\to\tau_1)\to\tau_2}(\texttt{e}) &&\triangleq \texttt{let x = e in } \lambda x_{\text{thnk}}.\texttt{let } x_{\text{conv}} = C_{\tau_1\mapsto\tau_1}(x_{\text{thnk}}\ ())\\
&&&\qquad \texttt{in let } x_{\text{acc}} = \text{thunk}(x_{\text{conv}}) \texttt{ in } C_{\tau_2\mapsto\tau_2}(\texttt{x } x_{\text{acc}})\\
&C_{(\texttt{unit}\to\tau_1)\to\tau_2\mapsto\tau_1\multimap\tau_2}(\texttt{e}) &&\triangleq \texttt{let x = e in } \lambda x_{\text{thnk}}.\texttt{let } x_{\text{acc}} = \text{thunk}(C_{\tau_1\mapsto\tau_1}(x_{\text{thnk}}\ ())) \texttt{ in } C_{\tau_2\mapsto\tau_2}(\texttt{x } x_{\text{acc}})
\end{aligned}$$

Fig. 13 . Convertibility for MiniML and Affi.

$$\begin{aligned}
P1 &= (\!|(\lambda x : (\texttt{unit}\to\texttt{int}\times\texttt{int}).\texttt{fst } (\texttt{x } ()))|\!)_{\texttt{bool}\otimes\texttt{bool}\multimap\texttt{bool}}\ \textbf{(true, false)}\\
P1^\dagger &= (\!|(\lambda x : (\texttt{unit}\to\texttt{int}\times\texttt{int}).(\texttt{fst } (\texttt{x } ()), \texttt{snd } (\texttt{ x } ()\ )))|\!)_{\texttt{bool}\otimes\texttt{bool}\multimap\texttt{bool}\otimes\texttt{bool}}\ \textbf{(true, false)}\\
P2 &= \textbf{(}\lambda\texttt{a} : \texttt{bool}.(\!|(\lambda y : \texttt{int}.((\!|\texttt{a}|\!)_{\texttt{int}}, y))\ 0|\!)_{\texttt{bool}\otimes\texttt{bool}}\textbf{)}\ \textbf{true}\\
P2^\dagger &= \textbf{(}\lambda\texttt{a} : \texttt{bool}.(\!|(\lambda y : \texttt{int}.((\!|\texttt{a}|\!)_{\texttt{int}}, (\!|\ \texttt{a}\ |\!)_{\texttt{int}}))\ 0|\!)_{\texttt{bool}\otimes\texttt{bool}}\textbf{)}\ \textbf{true}
\end{aligned}$$

$$\begin{aligned}
\text{compile}^*(P1) &= (\lambda x^1_{\text{thnk}}.(\lambda x.\{\texttt{fst } (\texttt{x } ())\})\ \text{thunk}(\texttt{let } x^2 = (x^1_{\text{thnk}}\ ())\ \texttt{in (if (fst } x^2\texttt{) 0 1, if (snd } x^2\texttt{) 0 1)})\\
&\qquad (\text{thunk}((0, 1)))\\
\text{compile}^*(P1^\dagger) &= (\lambda x^1_{\text{thnk}}.(\texttt{let } x^3 = (\lambda x.\{(\texttt{fst } (\texttt{x } ()), \texttt{snd}(\ \texttt{x } ()\ )\})\\
&\qquad (\text{thunk}(\texttt{let } x^2 = (x^1_{\text{thnk}}\ ())\ \texttt{in (if (fst } x^2\texttt{) 0 1, if (snd } x^2\texttt{) 0 1)}))\\
&\qquad \texttt{in (fst } x^3\texttt{, snd } x^3\texttt{))}\ (\text{thunk}((0, 1)))\\
\text{compile}^*(P2) &= (\lambda\texttt{a}.(\texttt{let } x^2 = ((\lambda y.(\texttt{if (a ()) 0 1, y)})\ 0)\ \texttt{in (fst } x^2\texttt{, snd } x^2\texttt{)))}\ (\texttt{let } x^1 = 0\ \texttt{in}\ \text{thunk}(x^1))\\
\text{compile}^*(P2^\dagger) &= (\lambda\texttt{a}.(\texttt{let } x^2 = ((\lambda y.(\texttt{if (a ()) 0 1, if (}\ \texttt{a ()}\ \texttt{) 0 1)})\ 0)\ \texttt{in (fst } x^2\texttt{, snd } x^2\texttt{)))}\ (\texttt{let } x^1 = 0\ \texttt{in}\ \text{thunk}(x^1))
\end{aligned}$$

Note: compile*(·) performs basic simplifications after compiling.

Fig. 14 . Examples of interoperability for MiniML and Affi, with compilations.

value from MiniML. This works fine, evaluating to **(true, true)**. Program $P2^\dagger$ attempts to do an analogous thing, but uses the variable twice, which is a violation of the affine type and thus results in a runtime failure. We can see that in the invocations of a (), which contain thunk(0).

5 AFFINE & UNRESTRICTED, EFFICIENTLY

While the model described in the previous case study allowed flexible interoperability and sound reasoning, it came at a runtime cost: all affine bindings had to be checked on use. Indeed, an entirely Affi program would have exactly the same runtime checks, even though the source type system guarantees that they would all be redundant!

In order to address this, in this case study we modify the previous one, still using Affi and MiniML but making a distinction between Affi functions (and thus bindings) that may be passed across the boundary (so-called "dynamic" affine arrows $\multimap_\circ$, which bind dynamic affine variables $\texttt{a}_\circ$), and ones that will only ever be used within Affi (so-called "static" affine arrows $\multimap_\bullet$, which bind static affine variables $\texttt{a}_\bullet$).

$$
\begin{array}{llll}
\textsc{Affi} & \text{Type } \tau & ::= & \mathsf{unit} \mid \mathsf{bool} \mid \mathsf{int} \mid \tau \multimap \tau \mid \tau \multimap_\bullet \tau \mid !\tau \mid \tau \& \tau \mid \tau \otimes \tau \\
& \text{Expression } e & ::= & () \mid \mathsf{true} \mid \mathsf{false} \mid n \mid x \mid a_\circ \mid a_\bullet \mid \lambda a_\bullet : \tau.e \mid e\ e \mid (\!|e|\!)_\tau \mid !v \\
& & & \mid \mathsf{let}\ !x = e\ \mathsf{in}\ e' \mid \langle e, e' \rangle \mid e.1 \mid e.2 \mid (e, e) \mid \mathsf{let}\ (a_\bullet, a'_\bullet) = e\ \mathsf{in}\ e' \\
& \text{Value } v & ::= & () \mid \lambda a_\bullet : \tau.e \mid !v \mid \langle e, e' \rangle \mid (v, v') \\
& \text{Mode } \bullet & ::= & \circ \mid \bullet
\end{array}
$$

$$
\frac{a_\bullet : \tau \in \Omega}{\mathfrak{C}; \Gamma; \Omega \vdash a_\bullet : \tau \rightsquigarrow \mathfrak{C}}
\qquad
\frac{\mathfrak{C}; \Gamma; \Omega[a_\circ := \tau_1] \vdash e : \tau_2 \rightsquigarrow \mathfrak{C}' \quad \mathrm{no}_\bullet(\Omega)}{\mathfrak{C}; \Gamma; \Omega \vdash \lambda a_\circ : \tau_1.e : \tau_1 \multimap \tau_2 \rightsquigarrow \mathfrak{C}'}
\qquad
\frac{\mathfrak{C}; \Gamma; \Omega[a_\bullet := \tau_1] \vdash e : \tau_2 \rightsquigarrow \mathfrak{C}'}{\mathfrak{C}; \Gamma; \Omega \vdash \lambda a_\bullet : \tau_1.e : \tau_1 \multimap_\bullet \tau_2 \rightsquigarrow \mathfrak{C}'}
$$

$$
\frac{\Omega = \Omega_1 \uplus \Omega_2 \quad \mathfrak{C}_1; \Gamma; \Omega_1 \vdash e_1 : \tau_1 \multimap_\bullet \tau_2 \rightsquigarrow \mathfrak{C}_2 \quad \mathfrak{C}_2; \Gamma; \Omega_2 \vdash e_2 : \tau_1 \rightsquigarrow \mathfrak{C}_3}{\mathfrak{C}_1; \Gamma; \Omega \vdash e_1\ e_2 : \tau_2 \rightsquigarrow \mathfrak{C}_3}
$$

$$
\frac{\Omega = \Omega_1 \uplus \Omega_2 \quad \mathfrak{C}_1; \Gamma; \Omega_1 \vdash e : \tau_1 \otimes \tau_2 \rightsquigarrow \mathfrak{C}_2 \quad \mathfrak{C}_2; \Gamma; \Omega_2[a := \tau_1, a' := \tau_1] \vdash e' : \tau' \rightsquigarrow \mathfrak{C}_3}{\mathfrak{C}_1; \Gamma; \Omega \vdash \mathsf{let}\ (a_\bullet, a'_\bullet) = e\ \mathsf{in}\ e' : \tau' \rightsquigarrow \mathfrak{C}_3}
$$

$$
\frac{\Omega = \Omega_e \uplus \Omega' \quad \mathrm{no}_\bullet(\Omega_e) \quad \Delta; \Gamma; \Gamma; \Omega_e \vdash e : \tau \rightsquigarrow \Delta; \Gamma \quad \_ : \tau \sim \tau}{\Gamma; \Omega; \Delta; \Gamma \vdash (\!|e|\!)_\tau : \tau \rightsquigarrow \Gamma'; \Omega'}
$$

Fig. 15 . Syntax and selected statics for efficient Affi, along with the new typing rule for efficient MiniML.

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

Languages. The syntax and static semantics for MiniML are nearly identical to the previous case study, except that the typing rule for $(\!|e|\!)_\tau$ now requires that e have no free static variables. We present syntax and selected static semantics for Affi as well as the new typing rule for MiniML in Fig. 15 — most of the typing rules are the same as in the previous case study, so we refer the reader to Fig. 10. We write dynamic bindings and arrows with a hollow circle $\circ$ and static ones with a solid circle $\bullet$.

Note that we do not allow a dynamic function to close over static resources, as it may be duplicated if passed to MiniML, and 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 closure is called at most once. Once called, any static resources in its body will surely be used safely because the static closure typechecked.

Our compilers are quite similar to the previous case – identical for MiniML, and only different for Affi in that we compile variables, functions, and application differently if they are static. That compiler is in Fig. 16, where you can see, as expected, the static features do not introduce the dynamic overhead of thunk(·)s.

Semantic Model. The most interesting part of this case study, and what we want to focus on the rest of this section is the logical relation. While the values that can cross the language boundary have (intentionally) not changed, the static bindings within Affi are challenging to deal with since they are a resource that must be kept isolated from parts of the program that would be allowed to use them more than once. To reason about this in our model, we first define an augmented target operational semantics. This augmented semantics 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. First, we extend our machine configurations to keep track

$$\text{thunk}(e) \triangleq \text{let } r_{\text{fresh}} = \text{ref } 1 \text{ in } \lambda\_.\{\text{if } !r_{\text{fresh}} \{\text{fail CONV}\} \{r_{\text{fresh}} := 0; e\}\}$$

$$\begin{array}{llll} a_\circ & \rightsquigarrow & a_\circ\ () & \text{let } (a_\bullet, a'_\bullet) = e_1 \text{ in } e_2 \rightsquigarrow \text{let } x_{\text{fresh}} = e^+, \\ a_\bullet & \rightsquigarrow & a_\bullet & \quad a_\bullet = \text{fst } x_{\text{fresh}}, \\ \lambda a_\circ : \tau.e & \rightsquigarrow & \lambda a_\circ.\{e^+\} & \quad a'_\bullet = \text{snd } x_{\text{fresh}} \text{ in } e'^+ \\ \lambda a_\bullet : \tau.e & \rightsquigarrow & \lambda a_\bullet.\{e^+\} & \\ (e_1 : \tau_1 \multimap \tau_2)\ e_2 & \rightsquigarrow & e_1^+ \ (\text{let } x = e_2^+ \text{ in thunk}(x)) & (e_1 : \tau_1 \multimap_\bullet \tau_2)\ e_2 \rightsquigarrow e_1{}^+\ e_2{}^+ \end{array}$$

Fig. 16 . Selected cases for compiler for efficient AFFI.

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, which uses the aforementioned phantom flags when it reduces:

$$\text{Expressions e} \quad ::= \quad \ldots \text{protect}(e, f) \qquad \langle \Phi \uplus \{f\}, H, \text{protect}(e, f) \rangle \dashrightarrow \langle \Phi, H, e \rangle$$

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):

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

Note that we write $\dashrightarrow$ for a step in this augmented semantics, to distinguish it from the true operational step $\longrightarrow$. The augmented operational semantics and the phantom flags play a similar role in protecting affine resources to mutable reference flags in the dynamic case. However, the critical difference is that in the augmented semantics, a protect$(\cdot)$ed resource for which there is no phantom flag will get stuck, and thus be prohibited 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.

Although this particular case study was largely motivated by efficiency concerns, more broadly, we demonstrate 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, but rather that the operational behavior of the target is exactly what the type interpretations characterize. Thus, realizability models like these allow us to move more of our reasoning into the model and out of the actual runtime that we deploy. And, from a very pragmatic angle, such models may allow one to reason post-hoc about compilers for existing systems that were not designed with static reasoning in mind.

We present the full logical relation, with only minor bits left to our supplementary materials, in Fig. 17. First, we have worlds W which are composed of three parts: a step index k, a heap typing Ψ, and an affine store Θ. The heap typing is standard, and describes, for each pair of locations, a relation (from Typ) that characterizes the types of (related) values stored at those locations. The affine store Θ, which has locations disjoint from those in the heap typing, is novel, and modified from the previous case study. It maps pairs of flag locations (ℓ_1, ℓ_2) to either USED or to a pair of static phantom flag sets Φ_1 and Φ_2 *owned* by ℓ_1 and ℓ_2. These capture the static phantom flags that are protected behind the dynamic resource that the locations protect. The final invariant on worlds

$W \in World_n$ is that the static phantom flag sets (Φ_1, Φ_2) stored at different dynamic reference flags must all be disjoint.

We interpret types as sets of tuples of the form $(W, (\Phi_1, e_1), (\Phi_2, e_2))$ from $Atom$, which requires that the phantom flag sets that are owned by the terms be disjoint from the flags that exist in the affine store of the world, the judgment written $\Phi_1, \Phi_2 : W$. We define Typ, the set of valid type interpretations R as subsets of $Atom$ that are closed under world extension. In addition to Typ, we also define $UnrTyp$, which are the set of relations from Typ that have empty phantom flag sets – essentially, terms that will inhabit the MiniML language cannot own static phantom flags, and thus this definition will be used for defining the interpretation of polymorphic types.

World extension is novel, written $W_1 \sqsubseteq_{\Phi_1,\Phi_2} W_2$, which says that W_2 is a future world of W_1, but neither world contains the flags in the phantom flag stores Φ_1 and Φ_2. Otherwise, we define world extension in a standard way: the step index can decrease, heap typing can gain mappings but cannot overwrite or remove mappings, and analogously, the affine store cannot unmark something as USED, but can change a binding from (Φ_1, Φ_2) to USED.

With those preliminary definitions in mind, we can define the expression relation, $\mathcal{E}[\![\tau]\!]_\rho$. It is made up of worlds W and pairs of phantom flag stores and terms, which each flag store represents the phantom variables owned by the expression. 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 the world and the owned portions), the term e_1 will either: (i) run longer than the step index accounts for (ii) fail CONV 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 world has evolved to W'. 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 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 left alone. Note that, over the course of running, some phantom flags may have moved into the world, which has changed, but cannot have used what was in the rest.

Our value relation cases, are now mostly standard, so we will focus only on the interesting cases: $\multimap$ and $\multimap_\bullet$. $\mathcal{V}[\![\tau_1 \multimap \tau_2]\!].$ is defined to take arbitrary arguments from $\mathcal{V}[\![\tau_1]\!].$, 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 thunk that prevent multiple uses, but also store the phantom flags in the affine store. The idea is that a function $\lambda a_\circ : \_.e$ can be applied to an expression that closes over static phantom flags, like $\mathbf{let}\ (b_\bullet, c_\bullet) = (1, 2)\ \mathbf{in}\ \lambda a_\bullet.b_\bullet$ – the latter will have phantom flags for both $b_\bullet$ and $c_\bullet$. 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 there are no longer coming out in flags(W). That means, for the reduction to be well-formed, they 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 be used by protect$(\cdot)$ expressions, as they are no longer subject to the restrictions, in the definition of world extension, that phantom flag sets stored at locations are immutable (can only be changed when they are removed because the locations are set to USED).

The static function, $\mathcal{V}[\![\tau_1 \multimap_\bullet \tau_2]\!].$, 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.

With the logical relation in hand, we can prove the following:

$$Atom_n = \{(W, (\Phi_1, e_1), (\Phi_2, e_2)) \mid W \in World_n \wedge \Phi_1, \Phi_2 : W\} \qquad \Phi_1, \Phi_2 : W \triangleq \forall i \in \{1,2\}.\Phi_i\#\text{flags}(W, i)$$

$$AtomVal_n = \{(W, (\Phi_1, v_1), (\Phi_2, v_2)) \in Atom_n\} \qquad AtomVal = \bigcup_n AtomVal_n$$

$$World_n = \{(k, \Psi, \Theta) \mid k < n \wedge \Psi \subset HeapTy_k \wedge \text{dom}(\Psi)\#\text{dom}(\Theta) \\ \wedge (\forall (\ell_1, \ell_2) \mapsto (\Phi_1, \Phi_2), (\ell_1', \ell_2') \mapsto (\Phi_1', \Phi_2') \in \Theta.(\ell_1, \ell_2) \neq (\ell_1', \ell_2') \implies \Phi_1 \cap \Phi_1' = \Phi_2 \cap \Phi_2' = \emptyset)\}$$

$$\Theta = \{(\ell_1, \ell_2) \mapsto \textsc{used}\} \cup \{(\ell_1, \ell_2) \mapsto (\Phi_1, \Phi_2)\} \qquad \Phi = \{f\} \qquad \text{flags}(W, i) = \bigcup_{(\ell_1,\ell_2)\mapsto(\Phi_1,\Phi_2)\in W.\Theta} \Phi i$$

$$Typ_n = \{R \in 2^{AtomVal_n} \mid \forall (W, (\Phi_1, v_1), (\Phi_2, v_2)) \in R.\, \forall W'.\ W \sqsubseteq_{\Phi_1,\Phi_2} W' \implies (W', (\Phi_1, v_1), (\Phi_2, v_2)) \in R\}$$

$$Typ = \{R \in 2^{AtomVal} \mid \forall k. \lfloor R \rfloor_k \in Typ_k\} \qquad UnrTyp = \{R \in Typ \mid \forall (W, (\Phi_1, v_1), (\Phi_2, v_2)) \in R.\, \Phi_1 = \Phi_2 = \emptyset\}$$

$$(k, \Psi, \Theta) \sqsubseteq_{\Phi_1,\Phi_2} (j, \Psi', \Theta') \triangleq (j, \Psi', \Theta') \in World_j \ \wedge\ j \leq k \wedge\ \Phi_1, \Phi_2 : (k, \Psi, \Theta) \ \wedge\ \Phi_1, \Phi_2 : (j, \Psi', \Theta') \\ \wedge \forall (\ell_1, \ell_2) \in \text{dom}(\Psi).\lfloor \Psi(\ell_1, \ell_2) \rfloor_j = \Psi'(\ell_1, \ell_2) \ \wedge\ \forall (\ell_1, \ell_2) \in \text{dom}(\Theta).(\ell_1, \ell_2) \in \text{dom}(\Theta') \wedge \\ (\Theta(\ell_1, \ell_2) = \textsc{used} \implies \Theta'(\ell_1, \ell_2) = \textsc{used}) \ \wedge\ (\Theta(\ell_1, \ell_2) = (\Phi_1, \Phi_2) \implies \Theta'(\ell_1, \ell_2) = (\textsc{used} \vee (\Phi_1, \Phi_2)))$$

$$H_1, H_2 : W \ \triangleq\ (\forall (\ell_1, \ell_2) \mapsto R \in W.\Psi.\ (\triangleright W, H_1(\ell_1), H_2(\ell_2)) \in R) \\ \wedge\ (\forall (\ell_1, \ell_2) \mapsto \textsc{used} \in W.\Theta.\forall i \in \{1,2\}.\ H_i(\ell_i) = \textsc{used}) \\ \wedge\ (\forall (\ell_1, \ell_2) \mapsto (\Phi_1, \Phi_2) \in W.\Theta.\forall i \in \{1,2\}.\ H_i(\ell_i) = \textsc{unused})$$

$$\begin{aligned}
\mathcal{V}[\![\texttt{unit}]\!]_\rho &= \{(W, (\emptyset, ()), (\emptyset, ()))\} \\
\mathcal{V}[\![\texttt{int}]\!]_\rho &= \{(W, (\emptyset, n), (\emptyset, n)) \mid n \in \mathbb{Z}\} \\
\mathcal{V}[\![\tau_1 \times \tau_2]\!]_\rho &= \{(W, (\emptyset, (v_{1a}, v_{2a})), (\emptyset, (v_{1b}, v_{2b}))) \\
&\quad \mid (W, (\emptyset, v_{1a}), (\emptyset, v_{1b})) \in \mathcal{V}[\![\tau_1]\!]_\rho \wedge (W, (\emptyset, v_{2a}), (\emptyset, v_{2b})) \in \mathcal{V}[\![\tau_2]\!]_\rho\} \\
\mathcal{V}[\![\tau_1 + \tau_2]\!]_\rho &= \{(W, (\emptyset, \text{inl } v_1), (\emptyset, \text{inl } v_2)) \mid (W, (\emptyset, v_1), (\emptyset, v_2)) \in \mathcal{V}[\![\tau_1]\!]_\rho\} \\
&\quad \cup \{(W, (\emptyset, \text{inr } v_1), (\emptyset, \text{inr } v_2)) \mid (W, (\emptyset, v_1), (\emptyset, v_2)) \in \mathcal{V}[\![\tau_2]\!]_\rho\} \\
\mathcal{V}[\![\tau_1 \to \tau_2]\!]_\rho &= \{(W, (\emptyset, \lambda x.\{e_1\}), (\emptyset, \lambda x.\{e_2\})) \mid \forall v_1\ v_2\ W'.W \sqsubset_{\emptyset,\emptyset} W' \\
&\quad \wedge (W', (\emptyset, v_1), (\emptyset, v_2)) \in \mathcal{V}[\![\tau_1]\!]_\rho \implies (W', (\emptyset, [x{\mapsto}v_1]e_1), (\emptyset, [x{\mapsto}v_2]e_2)) \in \mathcal{E}[\![\tau_2]\!]_\rho\} \\
\mathcal{V}[\![\texttt{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}[\![\forall \alpha.\tau]\!]_\rho &= \{(W, (\emptyset, \lambda.e_1), (\emptyset, \lambda.e_2)) \mid \forall R \in UnrTyp,\ W'.W \sqsubset_{\emptyset,\emptyset} W' \implies (W', (\emptyset, e_1), (\emptyset, e_2)) \in \mathcal{E}[\![\tau]\!]_{\rho[\alpha\mapsto R]}\} \\
\mathcal{V}[\![\alpha]\!]_\rho &= \rho(\alpha) \\
\mathcal{V}[\![\texttt{unit}]\!]. &= \{(W, (\emptyset, ()), (\emptyset, ()))\} \\
\mathcal{V}[\![\texttt{bool}]\!]_\rho &= \{(W, (\emptyset, 0), (\emptyset, 0))\} \cup \{(W, (\emptyset, n_1), (\emptyset, n_2)) \mid n_1 \neq 0 \wedge n_2 \neq 0\} \\
\mathcal{V}[\![\texttt{int}]\!]. &= \{(W, (\emptyset, n), (\emptyset, n)) \mid n \in \mathbb{Z}\} \\
\mathcal{V}[\![\tau_1 \multimap \tau_2]\!]. &= \{(W, (\emptyset, \lambda\, x\{e_1\}), (\emptyset, \lambda\, x\{e_2\})) \mid \forall \Phi_1\ v_1\ \Phi_2\ v_2\ W'.W \sqsubset_{\emptyset,\emptyset} W' \wedge (W', (\Phi_1, v_1), (\Phi_2, v_2)) \in \mathcal{V}[\![\tau_1]\!]. \\
&\quad \implies ((W'.k, W'.\Psi, W'.\Theta \uplus (\ell_1, \ell_2) \mapsto (\Phi_1, \Phi_2)), \\
&\qquad (\emptyset, [x{\mapsto}\text{guard}(v_1, \ell_1)]e_1), (\emptyset, [x{\mapsto}\text{guard}(v_2, \ell_2)]e_2)) \in \mathcal{E}[\![\tau_2]\!].\} \\
\mathcal{V}[\![\tau_1 \multimap \tau_2]\!]. &= \{(W, (\Phi_1, \lambda\, a_\bullet.\{e_1\}), (\Phi_2, \lambda\, a_\bullet.\{e_2\})) \mid \forall \Phi_1'\ \Phi_2'\ f_1\ f_2\ v_1\ v_2\ W'.W \sqsubset_{\Phi_1,\Phi_2} W' \\
&\quad \wedge (W', (\Phi_1', v_1), (\Phi_2', v_2)) \in \mathcal{V}[\![\tau_1]\!]. \wedge \Phi_1 \cap \Phi_1' = \Phi_2 \cap \Phi_2' = \emptyset \\
&\quad \wedge f_1 \notin \Phi_1 \uplus \Phi_1' \uplus \text{flags}(W', 1) \wedge f_2 \notin \Phi_2 \uplus \Phi_2' \uplus \text{flags}(W', 2) \\
&\quad \implies (W', (\Phi_1 \uplus \Phi_1' \uplus \{f_1\}, [a_\bullet{\mapsto}\text{protect}(v_1, f_1)]e_1), \\
&\qquad (\Phi_2 \uplus \Phi_2' \uplus \{f_2\}, [a_\bullet{\mapsto}\text{protect}(v_2, f_2)]e_2)) \in \mathcal{E}[\![\tau_2]\!].\} \\
\mathcal{V}[\![!\tau]\!]. &= \{(W, (\emptyset, v_1), (\emptyset, v_2)) \mid (W, (\emptyset, v_1), (\emptyset, v_2)) \in \mathcal{V}[\![\tau]\!].\} \\
\mathcal{V}[\![\tau_1 \otimes \tau_2]\!]. &= \{(W, (\Phi_1 \uplus \Phi_1', (v_{1a}, v_{2a})), (\Phi_2 \uplus \Phi_2', (v_{1b}, v_{2b}))) \\
&\quad \mid (W, (\Phi_1, v_{1a}), (\Phi_2, v_{1b})) \in \mathcal{V}[\![\tau_1]\!]. \wedge (W, (\Phi_1', v_{2a}), (\Phi_2', v_{2b})) \in \mathcal{V}[\![\tau_2]\!].\} \\
\mathcal{V}[\![\tau_1 \& \tau_2]\!]. &= \{(W, (\Phi_1, (\lambda_.\{e_{1a}\}, \lambda_.\{e_{2a}\})), (\Phi_2, (\lambda_.\{e_{1b}\}, \lambda_.\{e_{2b}\}))) \\
&\quad \mid (W, (\Phi_1, e_{1a}), (\Phi_2, e_{1b})) \in \mathcal{E}[\![\tau_1]\!]. \wedge (W, (\Phi_1, e_{2a}), (\Phi_2, e_{2b})) \in \mathcal{E}[\![\tau_2]\!].\}
\end{aligned}$$

$$\begin{aligned}
\mathcal{E}[\![\tau]\!]_\rho = \{(W, (\Phi_1, e_1), (\Phi_2, e_2)) \mid\ & \text{freevars}(e_1) = \text{freevars}(e_2) = \emptyset \wedge \\
& \forall \Phi_{r1}, \Phi_{r2}, H_1, H_2{:}W,\ e_1',\ H_1',\ j < W.k.\ \Phi_{r1}\#\Phi_1 \wedge \Phi_{r2}\#\Phi_2 \wedge \Phi_{r1} \uplus \Phi_1, \Phi_{r2} \uplus \Phi_2 : W \wedge \\
& \langle \Phi_{r1} \uplus \text{flags}(W, 1) \uplus \Phi_1, H_1, e_1\rangle \xrightarrow{j} \langle \Phi_1', H_1', e_1'\rangle \nrightarrow \implies e_1' = \text{fail Conv} \vee (\exists \Phi_{f1}\ \Phi_{g1}\ \Phi_{f2}\ \Phi_{g2}\ v_2 H_2'W'. \\
& \quad \langle \Phi_{r2} \uplus \text{flags}(W, 2) \uplus \Phi_2, H_2, e_2\rangle \xrightarrow{*} \langle \Phi_{r2} \uplus \text{flags}(W', 2) \uplus \Phi_{f2} \uplus \Phi_{g2}, H_2', v_2\rangle \nrightarrow \\
& \quad \wedge\ \Phi_1' = \Phi_{r1} \uplus \text{flags}(W', 1) \uplus \Phi_{f1} \uplus \Phi_{g1} \wedge \\
& \quad \wedge\ W \sqsubseteq_{\Phi_{r1},\Phi_{r2}} W' \wedge\ H_1, H_2' : W' \ \wedge\ (W', (\Phi_{f1}, e_1'), (\Phi_{f2}, v_2)) \in \mathcal{V}[\![\tau]\!]_\rho)\}
\end{aligned}$$

$$\text{guard}(e, \ell) \triangleq \lambda\_.\{\text{if } !\ell\ \{\text{fail Conv}\}\ \{\ell := \textsc{used}; e\}\}$$

Fig. 17 . Logical Relation for efficient MiniML and Affi.

Theorem 5.1 (Convertibility Soundness).
If $\tau \sim \tau$ *then* $\forall\, (W, (\Phi_1, e_1), (\Phi_2, e_2)) \in \mathcal{E}[\![\tau]\!]. \implies (W, (\Phi_1, C_{\tau\mapsto\tau}(e_1)), (\Phi_2, C_{\tau\mapsto\tau}(e_2))) \in \mathcal{E}[\![\tau]\!]$.
$\forall\, (W, (\Phi_1, e_1), (\Phi_2, e_2)) \in \mathcal{E}[\![\tau]\!]. \implies (W, (\Phi_1, C_{\tau\mapsto\tau}(e_1)), (\Phi_2, C_{\tau\mapsto\tau}(e_2))) \in \mathcal{E}[\![\tau]\!]$.

Theorem 5.2 (Fundamental Property). *Same statement as Theorem 4.2.*

Theorem 5.3 (Type Safety for MiniML). *Same statement as Theorem 4.3.*

Theorem 5.4 (Type Safety for Affi). *Same statement as Theorem 4.4.*

Note that to prove Theorems 5.3 and 5.4, we require a lemma which states that, if $\langle H, e\rangle \xrightarrow{*} \langle H', e'\rangle \not\to$, then for any Φ, $\langle \Phi, H, e\rangle \dashrightarrow^{*} \langle \Phi'_1, H'_1, e'_1\rangle \not\to$. This lemma is necessary because the given assumption of the type safety theorem is that the configuration $\langle H, e\rangle$ 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.

6 RELATED WORK

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

Multi-language semantics. Matthews and Findler [43] studied the question of language interoperability from a source perspective, developing the idea of a syntactic multi-language with *boundary terms* (c.f., contracts [20, 21]) that mediate between the two languages. They consider interactions between a statically typed and a dynamically typed language, but similar techniques have been applied to a variety of languages (e.g., object-oriented [27, 28], affine and unrestricted [63], simple and dependently typed [52], functional language and assembly [53], linear and unrestricted [57]) and used to prove compiler properties (e.g., correctness [54], full abstraction [2, 48]). More recently, there has been an effort to generalize the multi-language construction and apply techniques from denotational [17] and categorical [16] semantics.

Barrett et al. [7] take a slightly different path, directly mixing languages, in their case PHP and Python, and allowing bindings from one to be used in the other. As discussed earlier, our approach differs since it is focused on what target code that realizes boundaries has to do, rather than on a source-level (perhaps idealized) version of that interaction.

Interoperability via typed targets. Shao and Trifonov [58, 64] studied interoperability much earlier, and much closer to our context: they explicitly consider interoperability mediated by translation to a common target. Like us, they expect type-safe source languages. They tackle the problem that one language has access to control effects and the other does not. Their approach, however, is quite different: it relies upon a target language with an effect-based type system that is sufficient to capture the safety invariants and support interoperation between code compiled from both source languages. This fits our framework, but our work is broader in that our approach accounts for both richly typed intermediate languages and untyped (or poorly typed) intermediate languages that may use runtime checks to enforce invariants. While typed intermediate languages obviously offer real benefits, there are also unaddressed problems, foremost of which is designing a usable type system that is sufficiently general to allow (efficient) compilation from all the languages you want to support.

Proving particular FFIs sound. There has been significant work on how to augment existing unsafe FFIs in order to make them safe, primarily by adding type systems, inference, static analysis, etc. For example, Furr and Foster [24] study interactions between OCaml and C via the C FFI, and specifically, how to ensure safety by doing type inference on the C code as it operates over a

structural representation of OCaml types. There is some faint similarity to our work, if you assume the type systems could be made sound and take the two interacting languages to not be OCaml and C but rather those languages augmented with the richer types and inference systems.

Along the same lines, Tan et al. [62] study safe interoperability between Java and C via the JNI. They do this by first ensuring safety for C via CCured [46], and then extending that with static and dynamic checks to ensure that the invariants of Java pointers and APIs can not be violated in C. This is a bit more explicitly a case of two now more-or-less typesafe languages interacting. Hirzel and Grimm [30] also build a system for safe interoperability between Java and C, though their system, Jeanie, relies on a novel syntax that embeds both languages and is responsible for analyzing both together before compiling to Java and C with appropriate JNI usage. There has been plenty of other research studying how to make the JNI safer by analyzing C for various properties (e.g., looking for exception behavior in [41]).

Rich FFIs. There has been lots of work exploring how to make existing FFIs safer, usually by extending the annotations that are written down so that there is less hand-written (and thus error-prone) code to write. Some of this was done in the context of the Haskell FFI, including work by Chakravarty [18], Finne et al. [22], Jones et al. [34]. While they were certainly intending to preserve type invariants from Haskell, or wanted to express type invariants via different mechanisms, and obviously were concerned about soundness, it's not clear from these papers whether any formal soundness properties were proved. Similar work has also been done in other languages, for example, for Standard ML Blume [15] embedded C types into ML such that ML programs could safely operate over low-level C representations. This approach fits well with our semantic framework (though, given weak types in C would be unsound), as they have ML types that have the same interpretation as corresponding C types (to minimize copying/conversions), realized by minimal wrapper code.

Another approach to having rich FFIs is to co-design both languages, as has been done in the much more recent verification project Everest [14], where a low-level C-like language Low* has been designed [55] to interoperate with an embedding of a subset of assembly suitable for cryptography [23]. By embedding both languages into the verification framework F*, they are able to prove rich properties about the interactions between the two languages.

An abstract framework for unsafe FFIs. Turcotte et al. [65] advocate a framework where an abstract version of the foreign language is incorporated, 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 blaming all unsound behavior on C. While this approach seems promising in the context of unsound languages whose behavior can be collapsed, it is less clear how it applies to sound languages interacting.

Modeling FFIs via State Machines. Lee et al. [39] specify the type (and other) constraints that exist in both the JNI and Python/C FFI via state machines and use that to generate runtime checks to enforce these at runtime. While this is practical work and so they do not prove properties about their system, Jinn, there are many similarities between their approach and ours. In particular, the idea that invariants that cannot be expressed via the languages themselves and should instead be checked via inserted code. We would expect that if their approach were applied to safe languages, we would be able to prove that the code that they inserted satisfied semantic interpretations of the respective types.

Semantic Models and Realizability Models The use of semantic models to prove type soundness has a long history [44]. We make use of step-indexed models [5, 6], 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 [35]: 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") [9]. Such models were used to prove type soundness of standalone languages, specifically, Benton and Zarfaty [13] proved an imperative while language sound and Benton and Tabareau [12] 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 [9]. Krishnaswami et al. [38] 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. [4]. Such realizability models have also been used by Jensen et al. [33] to verify low-level code using a high-level separation logic, and by Benton and Hur [10] to verify compiler correctness.

Finally, New et al. [47, 49, 50] make use of realizability models in their work on semantic foundations of gradual typing, work that we have drawn inspiration from, given gradual typing may be thought of as a special instance of language interoperability. They compile type casts in a surface gradual language to a target Call-By-Push-Value [40] 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.

7 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. We have shown how to define a source-level convertibility relation, $\tau_A \sim \tau_B$, that describes what types in a language A are convertible with types in a language B, and how to prove that the corresponding target-level glue code soundly implements those conversions. We prove soundness by defining realizability models that give interpretations of source types as sets of (or relations on) target terms, compatible with the implementation choices of the compilers. These models give us powerful reasoning tools, including the ability to encode static invariants that are otherwise impossible to encode 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.

We have demonstrated the flexibility of this framework through a series of case studies, but we hope that the framework can enable further exploration of the interoperability design space. There are interesting unanswered questions about interactions between lazy and strict languages (compilation to Call-By-Push-Value [40] may illuminate conversions), between single-threaded and concurrent languages (session types [31, 32, 61] may help guide interoperability with process calculi like the π-calculus [45]), about control effects, not to mention further explorations of polymorphism as described at the end of §3. In addition to language features, there is work to be done to better understand how the framework tolerates extensions over time, whether through adding source languages or changing those already proved sound. While the semantic model does allow more flexibility than a syntactic multi-language, since new source languages that do not require changes

to the world structure will not require changes to existing proofs, there are open questions about migration of the proofs in cases where additional static reasoning is necessary.

In future work, we would like to challenge the framework by applying it to a larger and more realistic case study. There are many possibilities of target language choices — from common bytecodes like JVM and .NET CIL, to compiler IRs like LLVM — and each would come with different appropriate source languages. Our immediate plan is to apply our framework to WebAssembly and Interface Types [26] to establish the soundness of conversions between the source language types that the Interface Types proposal supports (via composed lift and lower functions). We think it will also be a good demonstration of the realizability model, as the underlying lazy representation of conversions, and the corresponding performance guarantees that it comes with, are a critical element of the proposal.

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

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

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

[6] 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

[7] 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

[8] 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

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

[10] 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

[11] 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.

[12] 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.

[13] 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

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

Informatik, 1:1–1:12. https://doi.org/10.4230/LIPIcs.SNAPL.2017.1
[15] 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.
[16] 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.
[17] Samuele Buro and Isabella Mastroeni. 2019. On the Multi-Language Construction.. In *ESOP*. 293–321.
[18] Manuel MT Chakravarty. 1999. C->HASKELL, or Yet Another Interfacing Tool. In *Symposium on Implementation and Application of Functional Languages*. Springer, 131–148.
[19] Christos Dimoulas, Sam Tobin-Hochstadt, and Matthias Felleisen. 2012. Complete Monitors for Behavioral Contracts. In *European Symposium on Programming (ESOP)*.
[20] Robert Bruce Findler and Matthias Blume. 2006. Contracts as pairs of projections. In *International Symposium on Functional and Logic Programming*. Springer, 226–241.
[21] 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.
[22] Sigbjorn Finne, Daan Leijen, Erik Meijer, and Simon Peyton Jones. 1998. H/Direct: a binary foreign language interface for Haskell. In *Proceedings of the third ACM SIGPLAN international conference on Functional programming*. 153–162.
[23] Aymeric Fromherz, Nick Giannarakis, Chris Hawblitzel, Bryan Parno, Aseem Rastogi, and Nikhil Swamy. 2019. A verified, efficient embedding of a verifiable assembly language. *PACMPL* 3, POPL (2019), 63:1–63:30. https://doi.org/10.1145/3290376
[24] Michael Furr and Jeffrey S. Foster. 2005. Checking Type Safety of Foreign Function Calls. *SIGPLAN Not.* 40, 6 (June 2005), 62–72. https://doi.org/10.1145/1064978.1065019
[25] Jean-Yves Girard. 1971. Une Extension De L'Interpretation De Gödel a L'Analyse, Et Son Application a L'Elimination Des Coupures Dans L'Analyse Et La Theorie Des Types. In *Proceedings of the Second Scandinavian Logic Symposium*, J.E. Fenstad (Ed.). Studies in Logic and the Foundations of Mathematics, Vol. 63. Elsevier, 63–92. https://doi.org/10.1016/S0049-237X(08)70843-7
[26] WebAssembly GitHub. 2020. Interface Types Proposal. https://github.com/WebAssembly/interface-types/blob/master/proposals/interface-types/Explainer.md
[27] Kathryn E Gray. 2008. Safe cross-language inheritance. In *European Conference on Object-Oriented Programming*. Springer, 52–75.
[28] 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.
[29] Rich Hickey. 2020. A history of Clojure. *Proceedings of the ACM on programming languages* 4, HOPL (2020), 1–46.
[30] Martin Hirzel and Robert Grimm. 2007. Jeannie: granting java native interface developers their wishes. In *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2007, October 21-25, 2007, Montreal, Quebec, Canada*, Richard P. Gabriel, David F. Bacon, Cristina Videira Lopes, and Guy L. Steele Jr. (Eds.). ACM, 19–38. https://doi.org/10.1145/1297027.1297030
[31] Kohei Honda. 1993. Types for dyadic interaction. In *International Conference on Concurrency Theory*. Springer, 509–523.
[32] 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.
[33] 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
[34] Simon Peyton Jones, Thomas Nordin, and Alastair Reid. 1997. GreenCard: a foreign-language interface for Haskell. In *Proc. Haskell Workshop*.
[35] 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)*.
[36] 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
[37] Robert Kleffner. 2017. *A Foundation for Typed Concatenative Languages*. Master's thesis. Northeastern University.
[38] 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
[39] Byeongcheol Lee, Ben Wiedermann, Martin Hirzel, Robert Grimm, and Kathryn S. McKinley. 2010. Jinn: synthesizing dynamic bug detectors for foreign language interfaces. In *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2010, Toronto, Ontario, Canada, June 5-10, 2010*, Benjamin G.

Zorn and Alexander Aiken (Eds.). ACM, 36–49. https://doi.org/10.1145/1806596.1806601
[40] Paul Blain Levy. 2001. *Call-by-Push-Value.* Ph. D. Dissertation. Queen Mary, University of London, London, UK.
[41] Siliang Li and Gang Tan. 2014. Exception analysis in the java native interface. *Science of Computer Programming* 89 (2014), 273–297.
[42] 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).*
[43] 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
[44] Robin Milner. 1978. A theory of type polymorphism in programming. *J. Comput. Syst. Sci.* 17 (1978), 348–375.
[45] Robin Milner, Joachim Parrow, and David Walker. 1992. A calculus of mobile processes, i. *Information and computation* 100, 1 (1992), 1–40.
[46] George C Necula, Scott McPeak, and Westley Weimer. 2002. CCured: Type-safe retrofitting of legacy code. In *Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages.* 128–139.
[47] 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.
[48] 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
[49] 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.
[50] 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.
[51] 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.
[52] 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
[53] 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
[54] 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
[55] Jonathan Protzenko, Jean Karim Zinzindohoué, Aseem Rastogi, Tahina Ramananandro, Peng Wang, Santiago Zanella Béguelin, Antoine Delignat-Lavaud, Catalin Hritcu, Karthikeyan Bhargavan, Cédric Fournet, and Nikhil Swamy. 2017. Verified low-level programming embedded in F. *PACMPL* 1, ICFP (2017), 17:1–17:29. https://doi.org/10.1145/3110261
[56] John C. Reynolds. 1974. Towards a theory of type structure. In *Programming Symposium*, B. Robinet (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 408–425.
[57] 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
[58] Zhong Shao and Valery Trifonov. 1998. Type-directed continuation allocation. In *International Workshop on Types in Compilation.* Springer, 116–135.
[59] 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
[60] 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.
[61] 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.

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

[63] 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).

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

[65] 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

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

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